

CUBRID 2008 R3.0 사용자 매뉴얼

목차

매뉴얼 소개	1
CUBRID 소개	3
시스템 구조	4
시스템 구성	4
데이터베이스 볼륨 구조	5
데이터베이스 서버	8
브로커	9
인터페이스 모듈	11
CUBRID의 특징	12
CUBRID 시작	15
설치와 실행	16
Linux에서의 설치와 실행	16
Windows에서의 설치와 실행	18
환경 변수 설정 및 CUBRID 서비스 시작	20
환경 변수 설정	20
언어 설정	22
CUBRID 서비스 시작	22
CSQL 인터프리터 사용	24
CSQL 인터프리터 시작	24
CSQL에서 SQL 실행	25
CUBRID 매니저 사용	27
CUBRID 매니저 시작	27
질의 편집기 사용	31
JDBC를 이용한 프로그램 작성	33
JDBC 환경 설정	33
JDBC 예제 프로그램	35
PHP를 이용한 프로그램 작성	38
PHP 모듈 설치	38
PHP 예제 프로그램	39
ODBC와 ASP를 이용한 프로그램 작성	41
ODBC와 ASP 환경 설정	41
ASP 예제 프로그램	42

CCI 를 이용한 프로그램 작성	47
CCI 라이브러리	47
CCI 설치 및 구성	47
CCI 사용	47
CCI 예제 프로그램	49
 CSQL 인터프리터	 53
CSQL 인터프리터 소개	54
CSQL 실행	55
CSQL 실행 모드	55
CSQL 사용 방법	55
CSQL 시작 옵션	56
세션 명령어	60
 CUBRID SQL 설명서	 69
용어 정리	70
주석	71
식별자	72
예약어	74
데이터 타입	79
문자 스트링 데이터 타입	79
비트 스트링 데이터 타입	84
수치형 데이터 타입	86
날짜/시간 데이터 타입	91
집합형 데이터 타입	95
테이블 정의	99
CREATE TABLE	99
ALTER TABLE	113
DROP TABLE	120
RENAME TABLE	121
인덱스 정의	122
CREATE INDEX	122
ALTER INDEX	123
DROP INDEX	123
뷰 정의	125
CREATE VIEW	125
ALTER VIEW	127

DROP VIEW	130
RENAME VIEW	130
시리얼	131
CREATE SERIAL	131
ALTER SERIAL	133
DROP SERIAL	134
시리얼 사용	134
연산자와 함수	136
논리 연산자	136
비교 연산자	136
산술 연산자	139
집합 연산자	143
포함 연산자	147
비트 함수와 연산자	154
스tring 함수와 연산자	156
수치 연산 함수	178
날짜/시간 함수와 연산자	193
데이터 타입 변환 함수와 연산자	204
집계 함수	219
클릭 카운터 함수	223
ROWNUM 함수	225
정보 함수	229
조건 연산식과 함수	232
조건식	239
데이터 조회 및 조작	247
SELECT	247
조인 질의	258
부질의	261
계층적 질의	263
INSERT	272
UPDATE	275
REPLACE	277
DELETE	278
TRUNCATE	279
DO	280
PREPARED STATEMENT	280
트랜잭션과 잠금	283
개요	283
데이터베이스 트랜잭션	283

데이터베이스 동시성	287
잠금 프로토콜	289
트랜잭션 격리 수준	298
트랜잭션 종료와 복구	316
데이터베이스 사용자 권한	319
데이터베이스 사용자	319
사용자 관리	319
권한 부여	320
권한 해지	321
사용자 권한 관리 메소드	322
질의 최적화	325
통계 정보 갱신	325
SQL 힌트 사용	325
쿼리 실행 계획 보기	327
트리거	329
CREATE TRIGGER	329
ALTER TRIGGER	337
DROP TRIGGER	337
RENAME TRIGGER	338
지연된 트리거	338
트리거 디버깅	340
트리거를 이용한 응용	341
Java 저장 함수/프로시저	344
개요	344
Java 저장 함수/프로시저 사용을 위한 환경 설정	344
Java 저장 함수/프로시저 작성 단계	346
서버 내부 JDBC 드라이버 사용	347
다른 데이터베이스 연결	348
loadjava 유틸리티	349
로딩한 Java 클래스 등록	349
Java 저장 함수/프로시저 호출	352
주의 사항	355
메소드(METHOD)	358
개요	358
메소드 타입	358
메소드 호출	358
분할	361
분할이란?	361
영역 분할	361

해시 분할	364
리스트 분할	366
분할 관리	368
클래스 상속	375
개요	375
클래스 속성과 클래스 메소드	376
상속을 위한 순서 규칙	376
INHERIT 절	377
ADD SUPERCLASS 절	377
DROP SUPERCLASS 절	378
클래스 충돌 해결	379
개요	379
해결 지시자	379
수퍼클래스 충돌	380
서브클래스 충돌	381
스키마 불변성	382
스키마 변경 규칙	383
멀티미디어 데이터 관리	387
glo 클래스	387
CUBRID 시스템 카탈로그	394
개요	394
시스템 카탈로그 클래스	394
시스템 카탈로그 가상 클래스	407
카탈로그 클래스/가상 클래스 사용 권한	426
카탈로그 정보의 일관성	426
카탈로그에 대한 질의	426
관리자 안내서	429427
CUBRID 유틸리티	430428
CUBRID 제어	433431
CUBRID 유틸리티 사용법(구문)	433431
CUBRID 서비스	434432
데이터베이스 서버	437435
브로커	438436
CUBRID 매니저 서버	447445
데이터베이스 관리	448446
CUBRID 관리 유틸리티 사용법(구문)	448446
데이터베이스 사용자	448446

databases.txt 파일	449447
데이터베이스 생성	450448
데이터베이스 볼륨 추가	455453
데이터베이스 삭제	458456
데이터베이스 변경	459457
데이터베이스 등록	460458
사용 공간 확인	461459
사용 공간 정리	462460
통계 정보 갱신	464462
데이터베이스 서버 실행 통계 정보 출력	464462
잠금(Lock) 상태 확인	468466
데이터베이스 일관성 확인	468466
데이터베이스 트랜잭션 제거	469467
질의 수행 계획 캐시 확인	470468
데이터베이스 로그 응급 복구	471469
데이터베이스 내부 정보 출력	472470
백업 및 복구	473471
내보내기와 가져오기	473471
서버/클라이언트에서 사용하는 파라미터 출력	473471
데이터베이스 마이그레이션	475473
데이터베이스 마이그레이션	475473
데이터베이스 내보내기(unload)	476474
데이터베이스 가져오기(load)	479477
가져오기용 파일 작성 방법	484482
데이터베이스 백업 및 복구	487485
데이터베이스 백업	487485
백업 정책 및 방식	490488
백업 파일 관리	493491
데이터베이스 복구	494492
복구 정책과 절차	497495
다른 서버로의 데이터베이스 복구	498496
데이터베이스 복제	501499
복제 개념	501499
복제 설정	508506
복제 내부 구현	524522
복제 제약 사항	528526
복제 주의 사항	529527
CUBRID HA	530528
개요	530528

용어 정의	530528
HA 제약 사항	533531
HA 환경 설정	534532
HA 모드 구동	537535
HA 유틸리티	538536
HA 관련 JDBC 환경 설정	540538
HA 관련 시스템 카탈로그	541539

성능 튜닝 **543541**

데이터베이스 서버 설정	544542
데이터베이스 서버 설정이 미치는 범위	544542
cubrid.conf 설정 파일과 기본 제공 파라미터	544542
접속 관련 파라미터	548546
메모리 관련 파라미터	549547
디스크 관련 파라미터	551549
오류 메시지 관련 파라미터	552550
동시성/잠금 파라미터	554552
로깅 관련 파라미터	556554
트랜잭션 처리 관련 파라미터	559557
구문/타입 관련 파라미터	560558
쿼리 캐시 관련 파라미터	561559
유틸리티 관련 파라미터	563561
기타 파라미터	564562
데이터베이스 서버 설정값 변경	567565
브로커 설정	569567
cubrid_broker.conf 설정 파일과 기본 제공 파라미터	569567
공통 적용 파라미터	571569
브로커별 파라미터	571569
CUBRID 매니저 서버 설정	577575
cm.conf 설정 파일과 기본 제공 파라미터	577575
파라미터 설명	578576

API 레퍼런스 **581579**

JDBC API	582580
JDBC 프로그래밍	582580
CUBRIDConnection	591589
CUBRIDOID	592590

CUBRIDPreparedStatement	600598
CUBRIDResultSet	602600
CUBRIDResultSetMetaData	603601
CUBRIDStatement	604602
ODBC API	605603
ODBC 프로그래밍	605603
OLE DB API	608606
OLE DB 프로그래밍	608606
PHP API	612610
PHP 프로그래밍	612610
cubrid_affected_rows	615613
cubrid_bind	616614
cubrid_close_request	617615
cubrid_col_get	618616
cubrid_col_size	618616
cubrid_column_names	619617
cubrid_column_types	619617
cubrid_commit	620618
cubrid_connect	621619
cubrid_current_oid	621619
cubrid_data_seek	623620
cubrid_disconnect	624621
cubrid_drop	624621
cubrid_error_code	625622
cubrid_error_code_facility	626623
cubrid_error_msg	626623
cubrid_execute	627624
cubrid_fetch	628625
cubrid_fetch_assoc	629626
cubrid_fetch_field	630627
cubrid_fetch_lengths	631628
cubrid_fetch_object	631629
cubrid_fetch_row	632629
cubrid_field_flags	633630
cubrid_field_len	633630
cubrid_field_name	634631
cubrid_field_seek	634631
cubrid_field_table	635632
cubrid_field_type	635632

cubrid_free_result	636633
cubrid_get	636634
cubrid_get_charset	637634
cubrid_get_class_name	637635
cubrid_get_client_info	638635
cubrid_get_db_parameter	638635
cubrid_get_server_info	639636
cubrid_insert_id	639636
cubrid_is_instance	639637
cubrid_list_dbs	640637
cubrid_load_from_glo	640638
cubrid_lock_read	641638
cubrid_lock_write	642639
cubrid_move_cursor	642639
cubrid_new_glo	643640
cubrid_num_cols	644641
cubrid_num_fields	644642
cubrid_num_rows	645642
cubrid_prepare	646643
cubrid_put	646644
cubrid_real_escape_string	647645
cubrid_result	648645
cubrid_rollback	649646
cubrid_save_to_glo	649647
cubrid_schema	650647
cubrid_send_glo	653650
cubrid_seq_drop	653651
cubrid_seq_insert	654651
cubrid_seq_put	655652
cubrid_set_add	655653
cubrid_set_drop	656653
cubrid_unbuffered_query	657654
cubrid_version	657655
CCI API	659656
CCI 개요	659656
cci_bind_param	669666
cci_bind_param_array	671668
cci_bind_param_array_size	672669
cci_close_req_handle	673670

cci_col_get	673670
cci_col_seq_drop	674671
cci_col_seq_insert	674671
cci_col_seq_put	675672
cci_col_set_add	676673
cci_col_set_drop	676673
cci_col_size	677674
cci_connect	677674
cci_connect_with_url	678675
cci_cursor	679676
cci_cursor_update	680677
cci_disconnect	681678
cci_end_tran	682679
cci_execute	683680
cci_execute_array	685682
cci_execute_batch	687684
cci_execute_result	688685
cci_fetch	689686
cci_fetch_buffer_clear	689686
cci_fetch_sensitive	690687
cci_fetch_size	690687
cci_get_bind_num	691688
cci_get_class_num_objs	691688
CCI_GET_COLLECTION_DOMAIN	692689
cci_get_cur_oid	692689
cci_get_data	693690
cci_get_db_parameter	694691
cci_get_db_version	695692
cci_get_result_info	695692
CCI_GET_RESULT_INFO_ATTR_NAME	696693
CCI_GET_RESULT_INFO_CLASS_NAME	697694
CCI_GET_RESULT_INFO_IS_NON_NULL	697694
CCI_GET_RESULT_INFO_NAME	697694
CCI_GET_RESULT_INFO_PRECISION	698695
CCI_GET_RESULT_INFO_SCALE	698695
CCI_GET_RESULT_INFO_TYPE	699696
cci_glo_append_data	699696
cci_glo_compress_data	699696
cci_glo_data_size	700697

cci_glo_delete_data	700697
cci_glo_destroy_data	701698
cci_glo_insert_data	701698
cci_glo_load	701698
cci_glo_new	702699
cci_glo_read_data	705702
cci_glo_save	706703
cci_glo_truncate_data	706703
cci_glo_write_data	707704
CCI_IS_SET_TYPE, CCI_IS_MULTISSET_TYPE, CCI_IS_SEQUENCE_TYPE, CCI_IS_COLLECTION_TYPE	707704
cci_is_updatable	708705
cci_next_result	708705
cci_oid	709706
cci_oid_get	710707
cci_oid_get_class_name	710707
cci_oid_put	711708
cci_oid_put2	711708
cci_prepare	713710
CCI_QUERY_RESULT_ERR_MSG	714711
cci_query_result_free	714711
CCI_QUERY_RESULT_RESULT	715712
CCI_QUERY_RESULT_STMT_TYPE	715712
cci_savepoint	715712
cci_schema_info	716713
cci_set_db_parameter	719716
cci_set_element_type	720717
cci_set_free	720717
cci_set_get	721718
cci_set_isolation_level	721718
cci_set_make	722719
cci_set_max_row	723720
cci_set_size	723720

CUBRID 매니저 **725721**

CUBRID 매니저 소개	726722
CUBRID 매니저 구조	726722
CUBRID 매니저 클라이언트	727723
CUBRID 매니저 실행	733729

CUBRID 매니저 서버 환경 설정	733729
CUBRID 매니저 서버 실행	734730
CUBRID 매니저 클라이언트 실행	735731
호스트 관리	737733
기본 호스트 정보	737733
호스트 연결 및 연결 해제	738734
호스트 추가 및 삭제	738734
서비스 관리	740736
사용자 관리	742738
사용자 권한 설정	742738
사용자 추가, 편집 및 삭제	743739
속성 관리	745741
질의 편집기	748744
질의 편집기 구조	748744
툴바 기능	749745
질의 편집 창	751747
질의 결과 창	754750
질의 편집기 옵션	756752
질의 실행 계획 보기	758754
데이터베이스	763759
데이터베이스 기능 구조	763759
데이터베이스 연결	764760
데이터베이스 시작	766762
데이터베이스 생성	766762
데이터베이스 설정	770766
사용자	772768
테이블	774770
뷰	781777
트리거	783779
시리얼	785781
저장 프로시저	785781
작업 자동화	788784
저장 공간	791787
데이터베이스 로드	794790
데이터베이스 언로드	795791
데이터베이스 백업	797793
데이터베이스 복구	798794
데이터베이스 이름 변경	800796
데이터베이스 복사	801797

데이터베이스 최적화	802798
데이터베이스 공간 정리	803799
데이터베이스 검사	804800
데이터베이스 잠금 정보	804800
데이터베이스 트랜잭션 정보	806802
데이터베이스 삭제	807803
OID 탐색기	808804
다중 호스트 환경에서의 백그라운드 운영	809805
사용 중인 매개 변수 정보	809805
질의 수행 계획 캐쉬 정보	811807
브로커	813809
브로커 구조	813809
브로커 메뉴	813809
개별 브로커 기능	816812
SQL 로그	817813
상태 모니터	821817
로그	826822

매뉴얼 소개

매뉴얼 구성

CUBRID 데이터베이스 관리 시스템(Database Management System, DBMS) 매뉴얼의 구성은 다음과 같다.

- [CUBRID 시작](#) : CUBRID를 처음 시작하는데 참고할 수 있는 내용을 제공한다. 시스템 설치 및 실행 방법, CSQL 인터프리터와 CUBRID 매니저의 간단한 사용 방법을 찾아볼 수 있다. 또한, JDBC나 PHP, ODBC, CCI 등을 이용한 응용 프로그램 작성 예제를 제공한다.
- [CUBRID 소개](#) : CUBRID 데이터베이스 관리 시스템의 구조 및 특징을 설명한다.
- [CSQL 인터프리터](#) : CSQL은 CUBRID에서 명령어 방식으로 SQL 문을 사용할 수 있는 프로그램이다. CSQL 인터프리터의 간단한 사용법과 관련 명령어들을 설명한다.
- [관리자 안내서](#) : 데이터베이스 생성 및 삭제, 백업 및 복구, 마이그레이션, 복제 작업 등을 수행하는 방법을 설명한다. CUBRID 서버, 브로커 및 매니저 서버 등의 구동과 정지 등을 수행하는 CUBRID 유틸리티의 사용법도 포함되어 있다.
- [성능 튜닝](#) : 성능에 영향을 미칠 수 있는 시스템 파라미터의 설정 방법을 설명한다. 서버, 브로커, CUBRID 매니저 서버 각각의 설정 파일 사용법과 개별 파라미터의 의미를 설명한다.
- [CUBRID SQL 설명서](#) : 데이터 타입, 함수와 연산자, 데이터 조회나 테이블 조작 등, CUBRID에서 사용할 수 있는 SQL 구문에 대해 설명한다. 인덱스나 트리거, 분할, 시리얼 및 사용자 정보 변경 등의 작업을 위한 SQL 구문도 찾아볼 수 있다.
- [CUBRID 매니저](#) : GUI (Graphic User Interface) 방식의 데이터베이스 관리 및 질의 도구인 "CUBRID 매니저"의 사용 방법을 설명한다. "CUBRID 매니저"는 다양한 관리 작업을 손쉽게 하도록 도와주며, 접속한 데이터베이스에 SQL 문을 실행할 수 있는 "질의 편집기" 기능을 제공한다.
- [API 레퍼런스](#) : JDBC API, ODBC API, OLE DB API, PHP API 및 CCI API에 대해 설명한다.

매뉴얼 규약

다음 표는 CUBRID 데이터베이스 관리 시스템 제품 매뉴얼에서 '문장', '명령어', 그리고 '텍스트 안에서의 참조'를 식별하는데 사용되는 문서 규약이다.

규약	설명	예제
<i>기울임꼴</i>	변수를 나타낸다.	<i>persistent. stringValueName</i>
굵게	상수, CUBRID 키워드 등 정해진 값을	fetch () member function

	나타낸다.
고정폭 글꼴	구문, 코드 예제 또는 명령어의 실행 및 csql database_name 결과를 나타낸다.
대문자	CUBRID 키워드를 나타내는 데 사용된 SELECT 다(굵게 참조).
작은 따옴표(' ')	중괄호, 대괄호와 함께 사용되면 문법 <code>{'const_list'}</code> 의 필요한 부분을 나타낸다. 스트링을 감쌀 때도 사용된다.
대괄호([])	파라미터나 키워드가 선택적임을 나타 [ONLY] 낸다.
세로줄()	여럿 중의 하나만을 선택할 수 있음을 [COLUMN ATTRIBUTE] 나타낸다.
중괄호({ })로 쓴 파라미터	여럿 중에서 하나만을 반드시 선택해야 CREATE { TABLE CLASS } 함을 나타낸다.
중괄호({ })로 쓴 값	집합을 구성하는 원소를 나타낸다. {2, 4, 6}
중괄호 뒤의 줄임표({ }...)	파라미터 지정이 반복될 수 있음을 나 { <i>class_name</i> }... 타낸다.
산괄호(< >)	단일 키 또는 일련의 키 입력을 나타낸 <Ctrl+n> 다.

CUBRID 소개

CUBRID의 구조 및 특징을 설명한다. CUBRID는 객체 관계형 데이터베이스 관리 시스템으로서, 데이터베이스 서버, 브로커, CUBRID 매니저로 구성된다. CUBRID는 인터넷 데이터 서비스에 최적화된 데이터베이스 시스템이며, 사용자가 편리하게 사용할 수 있는 다양한 기능을 제공한다.

이 장에서는 설명하는 주요 내용은 다음과 같다.

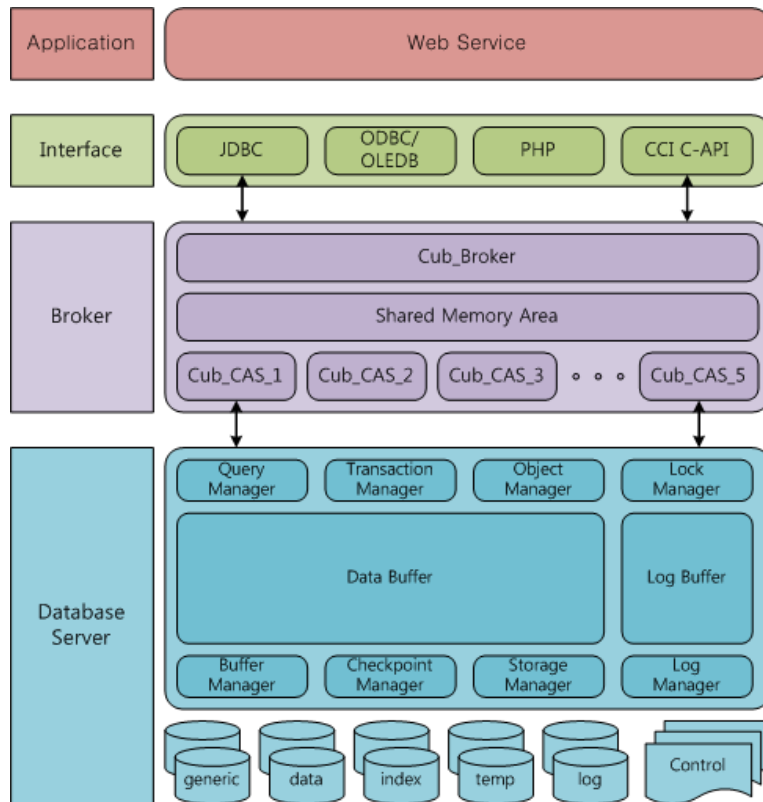
- 시스템 구조
- CUBRID의 특징

시스템 구조

시스템 구성

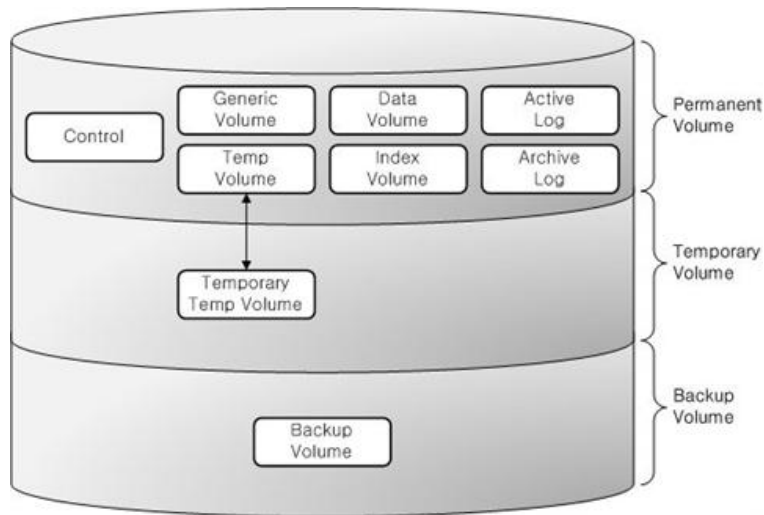
CUBRID는 객체 관계형 데이터베이스 관리 시스템으로서, 데이터베이스 서버, 브로커, CUBRID 매니저로 구성된다.

- 데이터베이스 서버는 CUBRID 데이터베이스 관리 시스템의 핵심 구성 요소로 데이터를 저장 및 관리하는 기능을 수행하며, 멀티스레드 기반 클라이언트/서버 방식으로 동작한다. 데이터베이스 서버는 사용자가 입력한 질의를 처리하고, 데이터베이스 내의 객체를 관리한다. CUBRID 데이터베이스 서버는 잠금 기법과 로깅 기법을 이용해 다수 사용자가 동시에 사용하는 환경에서도 완벽한 트랜잭션을 지원하며, 운영에 필요한 데이터베이스 백업과 복구 기능을 지원한다.
- 브로커는 서버와 외부 응용 프로그램 간의 통신을 중계하는 CUBRID 전용 미들웨어로서, 커넥션 풀링, 모니터링, 로그 추적 및 분석 기능을 제공한다.
- CUBRID 매니저는 데이터베이스와 브로커를 원격에서 관리할 수 있는 GUI 툴이다. 또한, CUBRID 매니저는 사용자가 데이터베이스 서버에 SQL 질의를 수행할 수 있는 편리한 기능의 질의 편집기를 제공한다. CUBRID 매니저에 대한 자세한 내용은 [CUBRID 매니저 안내서](#)를 참조한다.



데이터베이스 볼륨 구조

아래 그림은 CUBRID 데이터베이스 볼륨의 구조를 도식화한 구성도이다. 데이터베이스 볼륨을 크게 영구 볼륨, 임시 볼륨, 백업 볼륨으로 분류하고, 아래 구성도를 참고하여 각각에 속하는 볼륨 및 특징을 살펴보기로 한다.



영구 볼륨

영구 볼륨은 한번 생성되면 영구적으로 존재하는 데이터베이스 볼륨으로서, 볼륨 타입으로는 범용(generic), 데이터(data), 임시(temp), 인덱스(index), 제어(control), 활성 로그(active log), 보관 로그(archive log)가 있다.

범용 볼륨

사용자는 데이터베이스에 추가할 볼륨 타입을 데이터(data), 임시(temp), 인덱스(index) 중 하나의 용도로 지정하여 효율적으로 관리할 수 있는데, 별도로 데이터 용도를 지정하지 않는 경우에는 범용 볼륨으로 지정된다.

데이터 볼륨

데이터 볼륨이란, 인스턴스, 테이블, 멀티미디어 데이터 등과 같은 데이터를 저장하기 위한 볼륨이다.

임시 볼륨

임시 볼륨이란, 질의 처리 및 정렬(sorting)을 수행할 때 일시적으로 사용되는 볼륨이다. 다만, 임시 볼륨은 저장 공간이 임시적으로 생성 및 소멸되는 볼륨이 아니라 영구적으로 공간을 확보한 영구 볼륨 중 하나로서, 데이터가 임시적으로 저장 및 소멸되는 것을 의미한다. 따라서, CUBRID가 재시작하면 임시 볼륨 공간 내의 데이터는 초기화되고, 이에 관해 로그 정보는 남기지 않는다.

인덱스 볼륨

인덱스 볼륨이란, 신속한 질의 처리 또는 무결성 제약 조건(integrity constraints)의 신속한 검증을 위하여 인덱스 정보를 유지하는 볼륨이다.

제어 파일

제어 파일이란, 데이터베이스 내 존재하는 볼륨의 정보, 백업 정보, 및 로그의 정보를 저장한다.

- **볼륨 정보** : 데이터베이스 내 모든 볼륨의 이름과 위치, 그리고 내부 볼륨 식별자를 포함하는 정보로서, 데이터베이스가 재시작될 때 CUBRID는 볼륨 정보 제어 파일을 판독하며, 새로운 데이터베이스 볼륨이 추가될 때에 새로운 엔트리를 볼륨 정보 제어 파일에 기록한다.
- **백업 정보** : 정보 볼륨에 대한 모든 백업의 위치는 백업 정보 제어 파일에 기록된다. 이 제어 파일은 로그 파일이 관리되는 곳에 유지된다.
- **로그 정보** : 모든 활성 로그와 보관 로그의 이름을 포함하며, 사용자는 로그 정보 제어 파일을 통해 보관 로그의 정보를 확인할 수 있다. 이러한 로그 정보 제어 파일은 로그 파일과 동일한 위치에서 생성 및 관리된다.

이와 같이 각각의 제어 파일은 데이터베이스 볼륨의 위치, 백업 정보, 로그 정보를 포함하며, 데이터베이스가 재시작하면서 읽는 파일이므로 사용자 임의로 변경해서는 안 된다.

활성 로그

활성 로그(active log)란 데이터베이스의 최근 변경 사항을 포함하는 로그를 의미하며, 데이터베이스에 문제가 발생하는 경우 활성 로그 및 보관 로그를 이용하여 고장 발생 전의 커밋된 시점으로 완전하게 데이터베이스를 복구할 수 있다.

보관 로그

보관 로그는 최근의 변경 사항을 포함하고 있는 활성 로그(active log) 공간이 모두 사용된 후에 지속적으로 생성되는 로그를 보관하기 위한 볼륨이다. 마치 영구적 임시 볼륨 공간이 소진된 후에 일시적 임시 볼륨이 이용되는 것처럼, 활성 로그 볼륨의 공간이 소진된 후에 비로소 보관 로그 볼륨이 이용된다. 그러나, 일시적 임시 볼륨은 서버 프로세스가 종료될 때 자동 소멸하나 보관 로그 볼륨은 자동 소멸하지 않으므로 **DBA**가 불필요한 보관 로그에 대해서는 수동으로 삭제해야 한다는 차이점이 있다. **DBA**의 판단 하에 보관 로그 볼륨은 어느 때든 삭제할 수 있다.

일시 볼륨

일시 볼륨이란, 영구 볼륨과 반대되는 의미이다. 즉, 사용자가 영구 볼륨으로 지정한 공간을 초과하여 데이터가 축적되는 경우에만 일시적으로 마련되는 저장 공간을 일시 볼륨이라 하며, 이는 서버 프로세스가 종료됨에 따라 소멸된다. 이처럼 일시적으로 생성 및 소멸되는 볼륨으로는 일시적 임시 볼륨이 있다.

일시적 임시 볼륨

영구 볼륨에 속하는 임시 볼륨은 영구적으로 공간을 확보하는 볼륨이며, 일시적 임시 볼륨은 영구적 임시 볼륨으로 지정된 공간이 모두 사용된 후에 시스템에 의해 일시적으로 생성하는 임시 볼륨이다. 따라서, **DBA**는 데이터베이스 운영 상황을 고려하여 적절한 크기의 영구적 임시 볼륨을 확보해야 한다.

일시적 임시 볼륨은 조인(join)이나 정렬이 수행하거나 인덱스를 생성할 때 필요한 디스크 공간을 확보하는 경우에 생성되며, 일시적 임시 볼륨을 생성하는 대형 질의의 예로는 1) **GROUP BY**나 **ORDER BY**가 포함된

질의 2) 부질의(subquery)가 포함된 질의, 3) 정렬-병합(sort-merge) 조인이 수행되는 질의, 4) **CREATE INDEX** 문장이 포함된 질의 등이 있다.

- **일시적 임시 볼륨의 파일명** : CUBRID의 일시적 임시 볼륨의 파일명은 `db_name_tnum` 형식의 이름을 갖는다. 여기서 `db_name`은 데이터베이스 이름이고, `num`은 볼륨 식별자이다. 볼륨 식별자는 32766에서부터 1씩 감소한다.
- **일시적 임시 볼륨의 크기 설정** : 일시적 임시 볼륨이 생성되는 개수는 트랜잭션 처리에 필요한 공간의 크기에 따라 시스템이 결정한다. 그러나, 일시적 임시 볼륨의 크기는 사용자가 데이터베이스 파라미터 설정 파일(`cubrid.conf`)의 `temp_file_max_size_in_pages` 파라미터의 값을 설정함으로써 제한할 수 있다. 만약, `temp_file_max_size_in_pages`의 값이 0으로 설정되면 영구적 임시 볼륨이 소진되어도 일시적 임시 볼륨이 생성되지 않을 것이다.
- **일시적 임시 볼륨의 저장 위치 설정** : 일시적 임시 볼륨은 기본적으로 첫 번째 데이터베이스 볼륨이 생성된 위치에 만들어진다. 그러나, 사용자가 `temp_volume_path` 파라미터 값을 설정하여 일시적 임시 볼륨이 저장될 다른 디렉터리를 지정할 수 있다.
- **일시적 임시 볼륨의 삭제** : 일시적 임시 볼륨은 데이터베이스가 구동 중일때만 일시적으로 존재하며, 서버가 운영 중일때는 일시적 임시 볼륨을 삭제해서는 안 된다. 또한, 데이터베이스 서버가 독립 모드(`standalone`)로 운영되는 경우에는 서버 클라이언트의 접속이 종료될 때 일시적 임시 볼륨이 삭제되고, 데이터베이스 서버가 클라이언트/서버 모드로 운영되는 경우에는 `cubrid` 유틸리티에 의해 서버 프로세스가 정상적으로 종료될 때 일시적 임시 볼륨이 삭제된다. 만약, 데이터베이스 서버가 비정상적으로 종료된 경우라면 서버가 재시작할 때 일시적 임시 볼륨이 삭제된다.

백업 볼륨

백업 볼륨은 데이터베이스에 대한 스냅샷으로서, 이러한 백업 볼륨과 로그 볼륨을 기반으로 특정 시점까지 발생한 트랜잭션을 복구할 수 있다.

사용자는 `cubrid backuppdb` 유틸리티를 통해 데이터베이스 복구를 위해 필요한 모든 데이터를 복사할 수 있으며, 데이터베이스 환경 설정 파일(`cubrid.conf`)의 `backup_volume_max_size_bytes` 파라미터 값을 설정하여 백업 볼륨의 크기를 조정할 수 있다.

데이터베이스 서버

DB 서버 프로세스

데이터베이스별로 한개의 서버 프로세스가 존재한다. 서버 프로세스는 CUBRID 데이터베이스 서버를 구성하는 핵심 프로세스로 데이터베이스 파일 및 로그 파일 등에 직접 접근하여, 사용자의 요청을 처리한다. 클라이언트 프로세스는 서버 프로세스와 TCP/IP 통신을 통해 접속하며, 하나의 서버 프로세스는 스레드를 생성해서 다수의 클라이언트 프로세스의 요청 작업을 처리한다. 데이터베이스별, 즉 서버 프로세스별로 시스템 파라미터 설정을 지정할 수 있으면 서버 프로세스는 `max_clients` 파라미터 값으로 지정된 수 만큼의 클라이언트 프로세스의 접속이 가능하다.

마스터 프로세스

마스터 프로세스는 클라이언트 프로세스가 서버 프로세스에 접속하여 통신할 수 있게 하는 중개 프로세스로서, 호스트별로 한 개씩 동작한다. (정확히는 시스템 파라미터 파일인 **cubrid.conf**에 지정되는 접속 포트 번호별로 하나씩의 마스터 프로세스가 존재한다.) 마스터 프로세스는 지정된 TCP/IP 포트에 대기하고 있고, 클라이언트 프로세스는 해당 TCP/IP 포트에 마스터 프로세스에 접속한 후 마스터 프로세스가 지정된 데이터베이스 명에 따라 해당 서버 프로세스로 소켓 포트를 변경하여 접속을 처리한다.

실행 모드

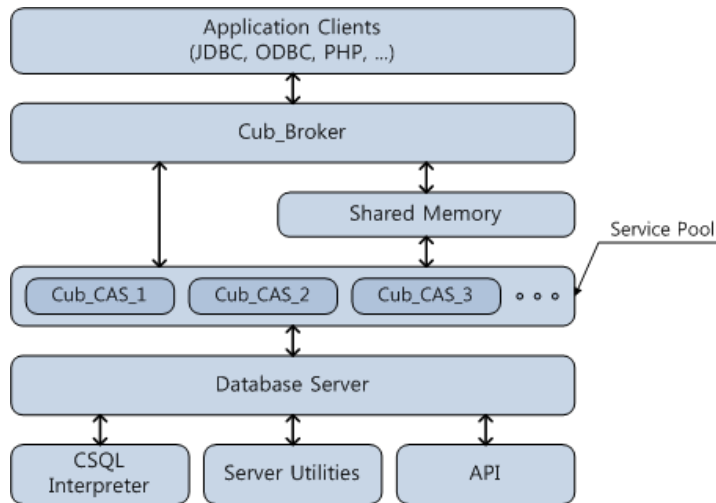
서버 프로세스를 제외한 CUBRID의 프로그램들은 종류에 따라 두 가지 실행 모드가 있다. 실행 모드는 클라이언트/서버 모드(client/server mode)와 독자 실행 모드(standalone mode)로 나뉜다.

- 클라이언트/서버 모드는 해당 프로그램이 클라이언트 프로세스로서 동작하여 서버 프로세스에 접속하는 방식이다.
- 독자 실행 모드는 해당 프로그램이 서버 프로세스의 기능을 포함하고 있어 직접 데이터베이스 파일에 접근하여 수행하는 방식이다.

예를 들어, 데이터베이스 생성 유틸리티나 복구 유틸리티 등은 다수 사용자가 데이터베이스에 접근하는 것을 막고 해당 프로그램만이 온전히 점유해서 작업할 수 있도록 독자 실행 모드로 실행된다. 또 다른 예로, CSQL 인터프리터는 클라이언트/서버 모드로 동작하여 서버 프로세스에 접속할 수도 있고, 독자 실행 모드로 동작하여 데이터베이스에 접근하여 SQL 문을 실행할 수도 있다. 참고로, 하나의 데이터베이스에 서버 프로세스와 독자 실행 프로그램이 모두 접근할 수는 없다.

브로커

브로커는 다양한 응용 클라이언트가 데이터베이스 서버에 연결할 수 있도록 중계하는 미들웨어이다. 브로커를 포함하는 큐브리드 시스템은 아래 그림과 같이, 응용 클라이언트, cub_broker, cub_cas, 데이터베이스 서버를 포함한 다중 계층 구조를 가진다.



응용 클라이언트

응용 클라이언트에서 사용할 수 있는 인터페이스는 C-API, ODBC, JDBC, PHP, Tcl/Tk, Python, Ruby, OLEDB, ADO.NET 등이 있다.

cub_cas

cub_cas(Cubrid Common Application Server)는 연결을 요청하는 모든 종류의 응용 클라이언트가 사용하는 공용 응용 서버 역할을 한다. 또한, cub_cas는 데이터베이스 서버의 클라이언트로 동작하여 클라이언트의 요청에 의해 데이터베이스 서버와 연결을 제공한다. 서비스 풀(service pool) 내에서 구동되는 cub_cas의 개수는 설정 파일에 지정할 수 있으며, cub_broker에 의해 동적으로 조정된다.

cub_cas는 CUBRID 데이터베이스 서버의 클라이언트 라이브러리와 링크되는 프로그램으로 서버 프로세스에는 클라이언트 모듈로 동작하며, 쿼리 파싱이나 최적화, 실행 계획 생성 등의 작업이 클라이언트 모듈에서 수행된다.

cub_broker

cub_broker는 응용 클라이언트와 cub_cas 사이의 연결을 중계하는 기능을 수행한다. 즉, 응용 클라이언트가 접근을 요청하면, cub_broker는 공유 메모리(shared memory)를 통해 cub_cas의 상태를 파악하여 접근 가능한 cub_cas에게 요청을 전달하고, 해당 cub_cas로부터 전달 받은 요청에 대한 처리 결과를 응용 클라이언트에게 반환한다.

또한, cub_broker는 서비스 풀 내의 cub_cas 개수를 조정하여 서버 부하를 관리하고, cub_cas의 구동 상태를 모니터링 및 관리한다. 만약, 응용 클라이언트의 요청을 cub_cas 1에게 전달하였는데, 비정상적인 종료로 인해 cub_cas 1과의 연결이 실패하면, cub_broker는 응용 클라이언트에게 연결 실패에 관한 에러 메시지를

전송하고 cub_cas 1을 재구동한다. 새롭게 구동된 cub_cas 1은 정상적인 대기 상태가 되어, 새로운 응용 클라이언트의 요청에 의해 재연결된다.

공유 메모리

공유 메모리에는 cub_cas의 상태 정보가 저장되며, cub_broker는 공유 메모리에 저장된 cub_cas의 상태 정보를 참조하여 응용 클라이언트와의 연결을 중계한다. 공유 메모리에 저장된 cub_cas의 상태 정보를 통해 시스템 관리자는 어떤 cub_cas가 현재 작업을 수행중인지, 어떤 응용 클라이언트의 요청이 처리 중인지를 확인할 수 있다.

인터페이스 모듈

CUBRID는 다양한 응용 프로그래밍 인터페이스(API : Application Programming Interface)를 제공한다.

지원되는 API는 다음과 같으며, CUBRID는 각 인터페이스를 제공하기 위한 인터페이스 모듈들을 제공한다.

- JDBC : Java 환경에서 데이터베이스 응용 프로그램을 작성하는 표준 API로 CUBRID는 인터페이스 모듈로 JDBC 드라이버를 제공한다.
- ODBC : Windows 환경에서 데이터베이스 응용 프로그램을 작성하는 표준 API로 CUBRID는 인터페이스 모듈로 ODBC 드라이버를 제공한다.
- OLE DB : Windows 환경에서 COM 방식으로 데이터베이스 응용 프로그램을 작성하는 API로 CUBRID는 인터페이스 모듈로 OLE DB 프로바이더를 제공한다.
- PHP : PHP 환경에서 데이터베이스 응용 프로그램을 작성할 수 있는 PHP 인터페이스 모듈을 제공한다. PHP 모듈은 CCI 라이브러리를 기반으로 작성되었다.
- CCI : CCI는 CUBRID에서 제공하는 C 언어 인터페이스로 인터페이스 모듈은 C 라이브러리 형태로 제공된다.

각 인터페이스 모듈들은 모두 브로커를 통해서 데이터베이스 서버에 접근하게 된다. 브로커는 다양한 응용 클라이언트가 데이터베이스 서버에 연결할 수 있도록 중계하는 미들웨어로 각 인터페이스 모듈의 요청을 받아서 데이터베이스 서버의 클라이언트 라이브러리에서 제공하는 native-C API를 호출하게 된다.

CUBRID의 특징

완벽한 트랜잭션 지원

트랜잭션의 원자성(atomicity), 일관성(consistency), 격리성(isolation), 지속성(durability)을 완벽하게 보장하기 위해 CUBRID는 다음의 기능을 충실하게 지원한다.

- 트랜잭션 단위의 commit, rollback, savepoint 지원
- 시스템이나 데이터베이스의 장애 시 트랜잭션 일관성 보장
- 복제간 트랜잭션 일관성 보장
- 데이터베이스, 테이블, 레코드 등 다중 단위 잠금(multiple granularity locking) 지원
- 교착 상태(deadlock) 자동 해결
- 분산 트랜잭션(two phase commit) 지원

데이터베이스 백업 및 복구

데이터베이스 백업은 CUBRID 데이터베이스 볼륨, 제어 파일, 로그 파일을 저장하는 작업이고, 데이터베이스 복구는 백업 작업에 의해 생성된 백업 파일, 활성 로그, 보관 로그를 이용하여 특정 시점의 데이터베이스로 복구하는 작업이다. 이 때, 복구 환경은 백업 환경과 동일한 운영체제 및 동일 버전의 CUBRID가 설치되어야 한다.

CUBRID가 지원하는 백업 방식으로는 온라인 백업, 오프라인 백업, 증분 백업이 있고, 복구 방식으로는 증분 백업에 의한 복구, 부분 복구, 전체 복구가 있다.

테이블 분할 - 파티션

분할 기법(partitioning)은 하나의 테이블을 여러 개의 독립적인 논리적 단위로 분할하는 기법을 가리킨다. 각 논리적 단위를 분할(partition)이라 부르며, 각 분할을 서로 다른 물리적 공간에 나누어 저장하도록 하여 레코드를 검색할 때 해당 분할 만으로의 접근을 통해 성능 향상을 기대할 수 있다. CUBRID가 제공하는 분할 기법은 다음과 같다.

- 레인지 분할 기법 : 컬럼 값의 범위를 기준으로 테이블을 분할하는 기법
- 해시 분할 기법 : 컬럼의 해시값을 기준으로 분할하는 기법
- 리스트 분할 기법 : 컬럼 값의 목록을 기준으로 분할하는 기법

HA 기능

CUBRID는 하드웨어, 소프트웨어, 네트워크 등에 장애가 발생해도 지속적인 서비스가 가능하게 하는 HA(High Availability) 기능을 제공한다. CUBRID의 HA 기능은 shared-nothing 구조이며, CUBRID Heartbeat을 이용하여 시스템과 CUBRID의 상태를 실시간으로 감시하고 장애 발생 시 절체(failover)를

수행한다. CUBRID HA 환경에서 마스터 데이터베이스 서버로부터 슬레이브 데이터베이스 서버로의 데이터 동기화를 위해 다음 두 단계를 수행한다.

- 마스터 데이터베이스 서버에서 생성되는 트랜잭션 로그를 실시간으로 다른 node에 복제하는 트랜잭션 로그 다중화 단계
- 실시간으로 복제되는 트랜잭션 로그를 분석하여 슬레이브 데이터베이스 서버로 데이터를 반영하는 트랜잭션 로그 반영 단계

Java 저장 프로시저

저장 프로시저는 미들웨어에서 실행되는 로직과 데이터베이스에서 실행되는 로직을 분리하여 응용 프로그램의 복잡성을 줄이고, 재사용성, 보안성, 성능을 향상시킬 수 있는 기법이다. CUBRID는 범용 언어인 Java로 작성되고, Java 가상 머신(JVM, Java Virtual Machine)에서 구동되는 Java 저장 프로시저를 제공한다. CUBRID에서 Java 저장 프로시저를 실행하기 위해서는 다음과 같은 절차가 수행되어야 한다.

- Java 가상 머신 설치 및 환경 설정
- Java 소스 파일 작성
- 컴파일 및 Java 리소스 로딩
- 로딩된 Java 클래스를 데이터베이스에서 호출할 수 있도록 등록
- Java 저장 프로시저 호출

클릭 카운터

인터넷 환경에서의 데이터 검색 시 검색이력을 남기기 위한 조회수 등과 같은 카운터를 데이터베이스에 유지하는 시나리오가 보편적 이다.

일반적으로 위의 시나리오는 **SELECT** 문을 이용하여 데이터를 검색하고, 검색한 질의에 대한 조회수를 증가시키기 위해 다시 **UPDATE** 문을 통해 구현하는 것이 일반적인 방식이었다.

이 방식은 한 데이터에 **SELECT**가 집중될 때 **UPDATE**에 대한 잠금(Lock) 경쟁이 가중되어 급격한 성능 저하가 발생하는 단점이 존재한다.

이에 CUBRID는 인터넷 환경에서 사용자 편의성 및 성능 측면에서 최적화된 기능을 제공하기 위해 클릭카운터(Click Counter) 라는 새로운 개념을 도입하고, 이를 위해 **INCR** 함수 및 **WITH INCREMENT FOR** 구문을 제공한다.

관계형 데이터 모델 확장

컬렉션

관계형 데이터베이스에서는 한 컬럼이 여러 개의 값을 가지는 것을 허용하지 않지만, CUBRID는 한 컬럼이 여러 개의 값을 가지도록 정의할 수 있다. 이를 위해 CUBRID에서는 컬렉션(collection)이라는 데이터 타입을 제공하는데, 컬렉션 타입은 집합 원소의 중복 허용 여부와 순서 유지 여부에 따라 크게 **SET**, **MULTISET**, **LIST**의 세 종류로 구분할 수 있다.

- **SET** : 각 원소의 중복을 허용하지 않는 집합으로서, 원소의 나열 순서와 무관하게 중복없이 정렬되어 저장된다.
- **MULTISET** : 각 원소의 중복을 허용하는 집합으로서, 원소의 나열 순서와 무관하다.
- **LIST** : 각 원소의 중복을 허용하는 집합으로서, **SET**, **MULTISET**과 달리 원소의 순서를 유지한다.

상속

상속은 부모 테이블에서 생성된 컬럼과 메소드들을 자식 테이블에서 재사용할 수 있게 하는 개념으로서, CUBRID는 상속을 지원함으로써 재사용성을 제공한다. CUBRID에서 제공하는 상속 기능을 이용하여 공통의 컬럼을 가지는 부모 테이블을 생성하고, 부모 테이블을 상속받아 고유한 컬럼을 추가한 자식 테이블을 생성함으로써, 필요한 컬럼 수를 최소화한 데이터베이스 모델링이 가능해진다.

컴포지션

관계형 데이터베이스에서는 참조하는 테이블이 참조되는 테이블의 기본 키를 외래 키로 가짐으로써 테이블 간 참조 관계가 정의되는데, 참조되는 테이블의 기본 키가 다수의 컬럼이거나 기본 키의 크기가 매우 큰 경우에는 테이블 간 조인 연산의 성능이 저하되는 문제가 있다. 그러나, CUBRID는 참조되는 테이블의 레코드가 위치하는 물리적인 주소값(OID)을 직접 이용할 수 있으므로 조인 연산 없이도 참조 관계를 정의할 수 있다.

즉, 객체형 데이터베이스에서는 참조되는 테이블의 기본 키 컬럼을 참조하는 대신에, 참조되는 테이블을 도메인(타입)으로 하는 컬럼을 통하여, 한 레코드가 다른 레코드의 참조값을 가지는 컴포지션 관계(composition relation)를 구성할 수 있다.

CUBRID 시작

CUBRID를 처음 사용하는데 참고할 수 있는 간략한 사용법을 설명한다. 시스템 설치 및 실행 방법, CSQL 인터프리터와 CUBRID 매니저의 간단한 사용법을 찾아볼 수 있다. 또한, JDBC나 PHP, ODBC, 그리고 CCI 등을 이용한 응용 프로그램 작성 예제도 포함되어 있다.

이 장에서는 설명하는 주요 내용은 다음과 같다.

- 설치와 실행
- 꼭 알아 두어야 할 것
- CSQL 인터프리터 사용
- CUBRID 매니저 사용
- JDBC를 이용한 프로그램 작성
- PHP를 이용한 프로그램 작성
- ODBC와 ASP를 이용한 프로그램 작성
- CCI를 이용한 프로그램 작성

설치와 실행

Linux에서의 설치와 실행

설치 시 확인 사항

Linux 버전의 CUBRID 데이터베이스를 설치하기 전에 다음 사항을 점검한다.

구분	내용
운영체제 버전	운영체제 버전에 상관 없이 glibc 2.3.4 버전 이상만 지원한다. glibc 버전은 다음과 같은 방법으로 확인한다. %rpm -q glibc
64 비트 여부	CUBRID 2008 R2.0 버전부터 32 비트 버전과 64 비트 버전을 각각 지원한다. Linux 버전은 다음과 같은 방법으로 확인한다. % uname -a Linux host_name 2.6.18-53.1.14.el5xen #1 SMP Wed Mar 5 12:08:17 EST 2008 x86_64 x86_64 x86_64 GNU/Linux 32 비트 Linux에서는 CUBRID 32 비트 버전을, 64 비트 Linux에서는 CUBRID 64 비트 버전을 설치한다. 설치할 추가 라이브러리는 다음과 같다. Curses Library (rpm -q ncurses) gcrypt Library (rpm -q libgcrypt) stdc++ Library (rpm -q libstdc++)

CUBRID 설치

설치 프로그램은 바이너리 셸로 되어 있어 자동으로 설치할 수 있다. 설치를 진행하려면 다운로드한 설치 파일을 바이너리 모드로 Linux 서버의 FTP로 전송한다.

다음은 리눅스에서 "CUBRID-8.3.0.0312-linux.x86_64.sh" 파일을 이용하여 CUBRID를 설치하는 예제이다.

```
[cub user@cubrid ~]$ sh CUBRID-8.3.0.0312-linux.x86_64.sh
Do you agree to the above license terms? (yes or no) : yes
Do you want to install this software(CUBRID) to the default(/home1/cub user/CUBRID)
directory? (yes or no) [Default: yes] : yes
Install CUBRID to '/home1/cub user/CUBRID' ...
In case a different version of the CUBRID product is being used in other machines, please
note that the CUBRID 2008 R3.0 servers are only compatible with the CUBRID 2008 R3.0
clients and vice versa.
Do you want to continue? (yes or no) [Default: yes] : yes
Copying old .cubrid.sh to .cubrid.sh.bak ...

CUBRID has been successfully installed.

demodb has been successfully created.

If you want to use CUBRID, run the following commands
% . /home1/cub_user/.cubrid.sh
% cubrid service start
```

위의 예제와 같이 다운로드한 파일(CUBRID-8.3.0.0312-linux.x86_64.sh)을 설치한 후, CUBRID 데이터베이스를 사용하기 위해서는 CUBRID 관련 환경 정보를 설정해야 한다. 이 설정은 해당 터미널에 로그인할 때 자동 설정되도록 지정되어 있으므로 설치 후 최초 한 번만 수행하면 된다.

```
[cub_user@cubrid ~]$ . /home1/cub_user/.cubrid.sh
```

CUBRID가 설치 완료되면 CUBRID 매니저 서버와 브로커를 다음과 같이 구동시킬 수 있다.

```
[cub_user@cubrid ~]$ cubrid service start
```

cubrid service를 구동시킨 후 정상적으로 구동되었는지 확인하려면 다음과 같이 grep으로 cub_* 프로세스들이 구동되어 있는지를 확인한다.

```
[cub_user@cubrid ~]$ ps -ef | grep cub_
cub_user 15200 1 0 18:57 ? 00:00:00 cub_master
cub_user 15205 1 0 18:57 pts/17 00:00:00 cub_broker
cub_user 15210 1 0 18:57 pts/17 00:00:00 query_editor cub_cas_1
cub_user 15211 1 0 18:57 pts/17 00:00:00 query_editor cub_cas_2
cub_user 15212 1 0 18:57 pts/17 00:00:00 query_editor cub_cas_3
cub_user 15213 1 0 18:57 pts/17 00:00:00 query_editor cub_cas_4
cub_user 15214 1 0 18:57 pts/17 00:00:00 query_editor cub_cas_5
cub_user 15217 1 0 18:57 pts/17 00:00:00 cub_broker
cub_user 15222 1 0 18:57 pts/17 00:00:00 broker1 cub_cas_1
cub_user 15223 1 0 18:57 pts/17 00:00:00 broker1 cub_cas_2
cub_user 15224 1 0 18:57 pts/17 00:00:00 broker1 cub_cas_3
cub_user 15225 1 0 18:57 pts/17 00:00:00 broker1 cub_cas_4
cub_user 15226 1 0 18:57 pts/17 00:00:00 broker1 cub_cas_5
cub_user 15229 1 0 18:57 ? 00:00:00 cub_auto_start
cub_user 15232 1 0 18:57 ? 00:00:00 cub_js_start
```

CUBRID 업그레이드

다른 버전의 CUBRID가 설치된 디렉터리를 CUBRID를 설치할 디렉터리로 지정하면, 해당 디렉터리가 존재하는 것을 알리고 덮어쓸 것인지 확인한다. **no**를 입력하면 설치가 중단된다.

```
Directory '/home1/cub_user/CUBRID' exist!
If a CUBRID service is running on this directory, it may be terminated abnormally.
And if you don't have right access permission on this directory(subdirectories or files),
install operation will be failed.
Overwrite anyway? (yes or no) [Default: no] : yes
```

CUBRID를 설치하고 설정 파일을 구성할 때 기존의 설정 파일을 그대로 사용할 것인지, 새 설정 파일을 사용할 것인지 확인한다. **yes**를 입력하면 기존의 설정 파일을 확장자가 .bak인 백업 파일로 보관한다.

```
The configuration file (.conf or .pass) already exists. Do you want to overwrite it? (yes or no) : yes
```

환경 설정

서비스 포트 등 사용자 환경에 맞춰 설정을 변경하려면 \$CUBRID/conf 디렉터리에서 설정 파일의 파라미터를 수정한다. 자세한 사항은 [환경 설정](#)을 참고한다.

참고 - RPM으로 설치할 때에는 의존성을 꼭 확인해야 한다. 의존성을 무시(--nodeps)하고 설치하면 실행되지 않을 수 있다.

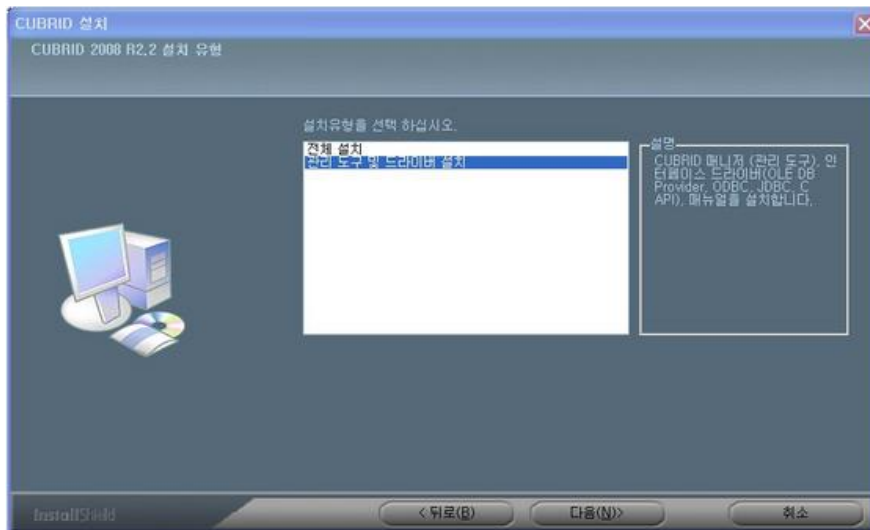
Windows 에서의 설치와 실행

설치 시 확인 사항

CUBRID 2008 R2.0 버전부터 32비트 버전과 64비트 버전을 각각 지원한다. [내 컴퓨터] > [시스템 등록 정보] 창을 활성화하여 Windows 버전 비트를 확인할 수 있다. 32비트 Windows에서는 CUBRID 32비트 버전을 설치하고, 64비트 Windows에서는 CUBRID 64비트 버전을 설치한다.

CUBRID 매니저 및 자바 저장 프로시저를 사용하려면 Java Runtime Environment (JRE) 1.5 이상 버전이 설치되어 있어야 한다. 그리고 Windows 버전에 한하여 Visual C++ 2008 재배포 패키지와 .NET Framework 2.0 이상 버전이 설치되어 있어야 한다. 그렇지 않으면 CUBRID Service가 Windows 서비스에 등록되지 않는다는 오류 메시지와 CUBRID_Service_Tray.exe를 시작할 수 없다는 오류 메시지가 출력된다.

설치 유형 선택



- **전체 설치** : CUBRID 서버, 커맨드라인 도구인 CSQL, GUI 관리 도구인 CUBRID 매니저, CUBRID 매뉴얼 및 인터페이스 드라이버(OLEDB Provider, ODBC, JDBC, C API)가 모두 설치된다.
- **관리 도구 및 드라이버 설치** : CUBRID 매니저, CUBRID 매뉴얼 및 인터페이스 드라이버(OLEDB Provider, ODBC, JDBC, C API)만 설치된다. CUBRID 데이터베이스 서버가 설치된 컴퓨터에 원격 접근하여 개발하거나 운영하는 경우, 이 설치 유형을 선택할 수 있다.

CUBRID 업그레이드

이전 버전의 CUBRID가 이미 설치된 환경에 새로운 버전의 CUBRID를 설치하는 경우, 시스템 트레이에서 [CUBRID Service Tray] > [Exit]를 선택하여 운영 중인 서비스를 종료한 후 이전 버전의 CUBRID를 제거해야

한다. "데이터베이스와 설정 파일을 모두 삭제하겠습니까?"라고 묻는 대화 상자가 나타나면, 이전 버전의 데이터베이스가 삭제되지 않도록 [아니오]를 클릭한다.

이전 버전의 데이터베이스를 새로운 버전으로 마이그레이션하는 작업은 [데이터베이스 마이그레이션](#)을 참고한다.

환경 설정

서비스 포트 등 사용자 환경에 맞춰 설정을 변경하려면 **%CUBRID%Wconf** 디렉터리에서 다음 파일의 파라미터 값을 변경한다.

구분	내용
cm.conf	CUBRID 매니저용 설정 파일로 기본 포트는 8001로 설정되어 있다. 설정된 포트와 설정된 포트 번호+1 두 개의 포트가 사용됨. 즉, 8001 포트를 설정하면 8001, 8002 두 개의 포트가 사용된다.
cubrid.conf	서버 설정용 파일로 운영하려는 데이터베이스의 메모리, 동시 사용자 수에 따른 스레드 수, 브로커와 서버 사이의 통신 포트 등을 설정한다. 자세한 내용은 데이터베이스 파라미터 를 참조한다.
cubrid_broker.conf	브로커 설정용 파일로 운영하려는 브로커가 사용하는 포트, 응용서버(CAS) 수, SQL LOG 등을 설정한다. 실제 JDBC와 같은 드라이버에서 보는 포트는 해당 브로커의 포트이다. 자세한 내용은 브로커별 파라미터 를 참조한다.

환경 변수 설정 및 CUBRID 서비스 시작

환경 변수 설정

CUBRID를 사용하기 위해서는 다음의 환경 변수들이 설정되어 있어야 한다. 필요한 환경 변수들은 CUBRID 시스템을 설치하면 자동으로 설정되나 필요에 의해서 사용자가 적절히 변경할 수도 있다.

CUBRID 환경 변수

- CUBRID : CUBRID 시스템이 설치된 위치를 지정하는 기본 환경 변수이다. CUBRID 시스템에 포함된 모든 프로그램은 이 환경 변수를 참조하므로 정확히 설정되어 있어야 한다.
- CUBRID_DATABASES : 데이터베이스 위치 정보 파일의 위치를 지정하는 환경 변수이다. CUBRID 시스템은 \$CUBRID_DATABASES/databases.txt 파일에 사용되는 데이터베이스 볼륨들의 절대 경로를 저장 관리한다. [databases.txt 파일](#)을 참고한다.
- CUBRID_LANG : CUBRID 시스템이 사용할 언어를 지정하는 환경 변수이다. 현재 CUBRID는 영어(en_US)와 한국어(ko_KR.euckr과 ko_KR.utf8)를 지원한다. 필수 설정은 아니며 설정되지 않은 경우 LANG 환경 변수를 참조하거나 기본값인 en_US를 사용한다. [언어 설정](#)을 참고한다.

이들 환경변수는 CUBRID를 설치하면서 이미 설정되었으나, 설정을 확인하기 위해서는 다음 명령을 사용할 수 있다.

Linux에서:

```
% printenv CUBRID
% printenv CUBRID_DATABASES
% printenv CUBRID_LANG
```

Windows에서:

```
C:\> set CUBRID
```

OS 환경 변수 및 Java 환경 변수

- PATH : Linux 환경에서 PATH 환경 변수에는 CUBRID 시스템의 실행 파일이 있는 디렉터리인 \$CUBRID/bin이 포함되어 있어야 한다.
- LD_LIBRARY_PATH : Linux 환경에서는 LD_LIBRARY_PATH (혹은 SHLIB_PATH나 LIBPATH) 환경 변수에 CUBRID 시스템의 동적 라이브러리 파일(libjvm.so)이 있는 디렉터리인 \$CUBRID/lib이 포함되어 있어야 한다.
- Path : Windows 환경에서 Path 환경 변수에는 CUBRID 시스템의 실행 파일이 있는 디렉터리인 \$CUBRID/bin이 포함되어 있어야 한다.
- JAVA_HOME : CUBRID 시스템에서 자바 저장 프로시저 기능을 사용하기 위해서는 자바 런타임 환경(JRE) 1.5이상 버전이 설치되어야 하고 JAVA_HOME 환경 변수에 해당 디렉터리가 지정되어야 한다. [Java 저장 함수/프로시저 사용을 위한 환경 설정](#)을 참고한다.

환경 변수 설정

Windows 환경인 경우

Windows 환경에서 CUBRID 시스템을 설치한 경우는 설치 프로그램이 필요한 환경 변수를 자동으로 설정한다. [시스템 등록 정보] 대화 상자의 [고급] 탭에서 [환경 변수]를 클릭하면 나타나는 [환경 변수] 대화 상자에서 확인할 수 있으며, [편집] 버튼을 통해 변경할 수 있다. Windows 환경에서 환경 변수를 변경하는 방법에 대한 상세한 정보는 Windows 도움말을 참고한다.



Linux 환경인 경우

Linux 환경에서 CUBRID 시스템을 설치한 경우는 설치 프로그램이 .cubrid.sh 혹은 .cubrid.csh 파일을 자동으로 생성하고 설치 계정의 셸 로그인 스크립트에서 자동으로 호출하도록 구성한다. 다음은 sh이나 bash 등을 사용하는 환경에서 생성된 .cubrid.sh 환경 변수 설정 파일의 내용이다.

```
CUBRID=/home1/cub user/CUBRID
CUBRID_DATABASES=/home1/cub user/CUBRID/databases
CUBRID_LANG=en US
ld lib path='printenv LD_LIBRARY_PATH`
if [ "$ld lib path" = "" ]
then
LD_LIBRARY_PATH=$CUBRID/lib
else
LD_LIBRARY_PATH=$CUBRID/lib:$LD_LIBRARY_PATH
fi
SHLIB_PATH=$LD_LIBRARY_PATH
LIBPATH=$LD_LIBRARY_PATH
PATH=$CUBRID/bin:$CUBRID/cubridmanager:$PATH
export CUBRID
export CUBRID_DATABASES
export CUBRID_LANG
export LD_LIBRARY_PATH
```

```
export SHLIB_PATH
export LIBPATH
export PATH
```

언어 설정

CUBRID 데이터베이스 관리 시스템은 사용할 언어를 **CUBRID_LANG** 환경 변수로 지정한다. 현재 **CUBRID_LANG** 환경 변수에 설정될 수 있는 값은 다음과 같다.

- **en_US** : 영어(디폴트 값)
- **ko_KR.euckr** : 한국어 EUC-KR 인코딩
- **ko_KR.utf8** : 한국어 UTF-8 인코딩

CUBRID 시스템에서 언어 설정은 저장되는 데이터의 문자 세트(character set)를 의미하지는 않는다. 즉, **CUBRID_LANG**이 ko_KR.utf8로 설정되었다고 해서 저장되는 데이터가 해당 인코딩으로 변환되거나 보장되는 것은 아니다. CUBRID의 언어 설정은 프로그램들이 출력하는 메시지들에 영향을 미치며, 날짜/시간 데이터 타입 상수를 표현하는 방식에 영향을 미친다.

CUBRID_LANG 환경 변수가 설정되지 않은 경우에는 LANG 환경 변수의 값을 사용한다. **CUBRID_LANG** 혹은 **LANG** 값이 지원되지 않는 값으로 설정된 경우에는 디폴트 값인 en_US로 설정한 것과 같이 동작한다.

CUBRID 서비스 시작

환경 변수 및 언어 설정을 완료한 후, CUBRID 서비스를 시작한다. 이에 대한 자세한 설명은 [서비스 등록 및 서비스 구동 및 종료](#)를 참고한다.

셸 명령어

Linux 환경 또는 Windows 환경에서 아래와 같은 셸 명령어로 CUBRID 서비스를 시작하고, 설치 패키지에 포함된 demodb를 구동할 수 있다.

```
% cubrid service start

@ cubrid master start
++ cubrid master start: success
@ cubrid broker start
++ cubrid broker start: success
@ cubrid manager server start
++ cubrid manager server start: success

% cubrid server start demodb
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0

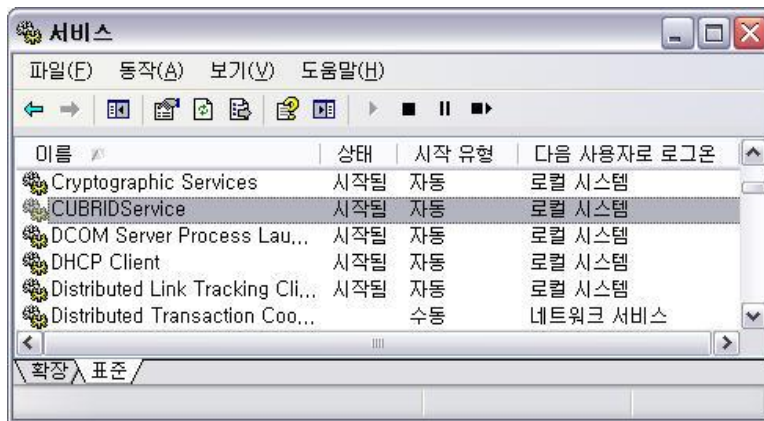
++ cubrid server start: success

@ cubrid server status
Server demodb (rel 8.3, pid 31322)
```

CUBRIDService 또는 CUBRID Service Tray

Windows 환경에서는 다음과 같은 방법으로 CUBRID 서비스를 시작하거나 중지할 수 있다.

- [제어판] > [성능 및 유지 관리] > [관리도구] > [서비스]에 등록된 CUBRIDService를 선택하여 시작하거나 중지한다.



- 시스템 트레이에서 CUBRID Service Tray를 클릭한 후, CUBRID를 시작하려면 [CUBRID Server] > [Start]를 선택하고 중지하려면 [CUBRID Server] > [Stop]을 선택한다. CUBRID가 실행 중일 때 CUBRID 서비스 트레이에서 [Exit]를 선택하면, 해당 서버에서 실행 중인 모든 서비스와 프로세스가 중지되므로 주의한다.



참고 CUBRID 서비스 트레이를 통해 CUBRID 관련 프로세스를 시작/종료하는 작업은 관리자 권한(SYSTEM)으로 수행되고, 셸 명령어로 시작/종료하는 작업은 로그인한 사용자 권한으로 수행된다. Windows 환경에서 셸 명령어로 CUBRID 프로세스가 제어되지 않는 경우, 관리자 권한으로 로그인하거나 CUBRID 서비스 트레이를 이용해서 해당 작업을 수행할 수 있다. CUBRID 서버 프로세스가 모두 중단되면, CUBRID Service Tray 아이콘이 붉은 색으로 변한다.

CSQL 인터프리터 사용

CSQL 인터프리터 시작

CSQL 인터프리터는 CUBRID에서 명령어 방식으로 SQL 질의를 수행하고 수행 결과를 조회할 수 있는 프로그램이다. 입력된 SQL 문장과 그 결과는 나중에 사용하기 위해서 파일에 저장될 수도 있다. 자세한 내용은 [CSQL 인터프리터 소개](#) 및 [CSQL 실행](#)을 참고한다.

CUBRID는 CSQL 인터프리터 이외에도 편리한 GUI 방식의 "CUBRID 매니저" 프로그램을 제공하며, CUBRID 매니저의 질의 편집기에서도 모든 SQL 문을 수행하고 결과를 조회할 수 있다. 자세한 내용은 [질의 편집기 구조](#)를 참고한다.

Windows 환경에서는 명령어 방식의 CSQL 인터프리터보다는 CUBRID 매니저의 사용을 더 권장한다. 따라서, 본 장에서는 Linux 환경에서 CSQL 인터프리터를 사용하는 경우를 설명한다.

CSQL 인터프리터 시작

csql 프로그램은 셸에서 다음과 같이 시작할 수 있다.

```
% csql demodb
      CUBRID SQL 인터프리터

도움말 메시지를 보려면 ';'help' 명령어를 입력하세요.
csql> ;help

=== <도움말: 세션 명령어 요약> ===

모든 세션 명령어는 ';'로 시작하며 앞에는 공백/탭 문자 이외에는 올 수 없습니다.
대문자로 표시된 부분은 해당 명령어를 수행하기 위해 지정되어야 할 최소한의 약어이다.

;REAd  [<file-name>]      - 명령어 버퍼로 파일 내용을 읽어 들임.
;Write [<file-name>]      - 명령어 버퍼 내용을 파일에 저장.
;APpend [<file-name>]    - 명령어 버퍼 내용을 파일에 덧붙임.
;PRINT                                - 명령어 버퍼 내용을 프린트.
;SHELL                                - 셸 수행.
;CD                                       - 현재 작업 디렉터리 변경.
;EXit                                  - 종료.

;CLeAr                                - 명령어 버퍼 내용을 지움.
;EDIT                                  - 명령어 버퍼 편집.
;List                                  - 명령어 버퍼 내용 출력.

;RUn                                   - 명령어 버퍼 실행.
;Xrun                                  - 명령어 버퍼 실행 후 버퍼 내용을 지움.
;Commit                               - 진행 중인 트랜잭션 커밋.
;ROllback                             - 진행 중인 트랜잭션 롤백.
;AUtocommit [ON|OFF]                - 자동 커밋 모드 설정|해제.
```

```

;REStart                - 데이터베이스에 재접속.

;SHELL_Cmd  [shell-cmd]  - 내장된 셸을 설정하거나 보여줌.
;EDITOR_Cmd [editor-cmd] - 내장된 편집기를 설정하거나 보여줌.
;PRINT_Cmd  [print-cmd]  - 내장된 프린트 명령어를 설정하거나 보여줌.
;Pager_cmd  [pager-cmd]  - 내장된 페이지 구분기를 설정하거나 보여줌.

;DATE                - 지역 날짜, 시간 출력.
;DATAbase            - 접근 중인 데이터베이스 이름 출력.
;SCHEMA class-name   - 클래스의 스키마 정보 출력.
;SYNTAX [sql-cmd-name] - 명령어 문법 출력.
;TRIGGER ['*'|trigger-name] - 트리거 정의 출력.
;GET system_parameter - 시스템 파라미터 보기.
;SET system_parameter=value - 시스템 파라미터 설정.
;PLAN [simple|detail|off] - 쿼리 플랜 보기 설정.
;INFO <command>      - 내부 정보 출력.
;TIME [ON/OFF]        - 쿼리 수행 시간 보기 설정|해제
;HISTORYList          - 수행된 쿼리 리스트 보기.
;HISTORYRead <history_num> - 히스토리 번호에 해당되는 쿼리를 명령어 버퍼에서 읽음.
;HELP                - 도움말 메시지 출력.

csql>

```

CSQL 에서 SQL 실행

csql을 실행하고 난 후에는 csql> 프롬프트에서 원하는 SQL문을 입력해서 실행할 수 있다. 하나의 SQL 문은 세미콜론(;)으로 끝나도록 입력하며, 여러 개의 SQL문을 한 줄에 입력할 수도 있다. 입력된 SQL 문들을 실행하기 위해서는 ;x 세션 명령어를 사용한다. 세션 명령어는 ;help 명령으로 간단한 사용법을 찾아 볼 수 있으며 상세한 내용은 [세션 명령어](#)를 참고한다.

```

% csql demodb
      CUBRID SQL 인터프리터

도움말 메시지를 보려면 ';help' 명령어를 입력하세요.
csql> select * from olympic;
csql> ;x
=== <SELECT의 결과, 명령어 라인 1> ===

      host year  host nation      host city      opening date  closing
_date mascot      slogan      introduction
=====
      2004  'Greece'      'Athens'      08/13/2004    08/29/2
004  'Athena Phevos'      'Welcome Home'      'In 2004 the Olympic Games returned to
Greece, the home of both the ancient Olympics and the first modern Olympics...'

<중간 생략>
25 개의 열이 조회 되었습니다.

진행 중인 트랜잭션이 커밋 되었습니다.

1 개의 명령어가 실행되었습니다.
csql> SELECT SUM(n) FROM (SELECT gold FROM participant WHERE nation_code='KOR'
csql> UNION ALL SELECT silver FROM participant WHERE nation_code='JPN') AS t(n);

```

```
csql> ;x
=== <SELECT의 결과, 명령어 라인 1> ===
      sum(n)
=====
        82

1 개의 열이 조회 되었습니다.

진행 중인 트랜잭션이 커밋 되었습니다.

1 개의 명령어가 실행되었습니다.
csql> ;exit
```

CUBRID 매니저 사용

CUBRID 매니저 시작

GUI 방식의 데이터베이스 관리 및 질의 도구인 "CUBRID 매니저"는 다양한 관리 작업을 손쉽게 하도록 도와주며, 접속한 데이터베이스에 SQL 문을 실행할 수 있는 "질의 편집기" 기능을 제공한다.

CUBRID 매니저는 Java Runtime Environment (JRE) 1.5 이상 버전이 설치된 환경에서 정상 실행된다.

JRE는 Developer Resources for Java Technology 사이트(<http://java.sun.com>)에서 다운로드할 수 있다.

CUBRID 매니저에 대한 보다 자세한 내용은 [CUBRID 매니저 소개](#) 및 [CUBRID 매니저 서버 실행](#)을 참고한다.

CUBRID 매니저 실행

Windows 환경에서 실행

- CUBRID 서비스 트레이를 클릭하고 [Tools] > [CUBRID Manager]를 선택한다.



Linux 환경에서 실행

셸에서 다음 중 하나의 명령을 입력하여 CUBRID 매니저 클라이언트를 실행할 수 있다.

- % cubridmanager
- % \$CUBRID/cubridmanager/cubridmanager

실행 오류 시 확인 사항

CUBRID 매니저가 정상적으로 구동되지 않으면, "서버에 접속할 수 없습니다. CUBRID 매니저 서버의 구동 및 기타 접속 환경을 점검하시기 바랍니다."라는 오류 메시지가 나타난다. 이 때, 사용자는 다음 사항을 확인하여 문제를 해결할 수 있다.

- CUBRID 매니저 서버가 구동 중인지 확인한다.
- CUBRID 매니저 서버의 환경 설정 파일을 열고, **cm_port** 파라미터의 값이 등록된 통신 포트와 같은지 확인한다. [CUBRID 매니저 서버 환경 설정](#)을 참고한다.
- 매니저 클라이언트가 구동되는 현재 시스템에 방화벽이 설정되어 있다면, 매니저 클라이언트 접속을 위한 통신 포트(**cm_port**, **cm_port+1**)를 모두 연다. 예를 들어, **cm_port**가 8001이면, 8001 포트와 8002 포트를 열어야 한다.

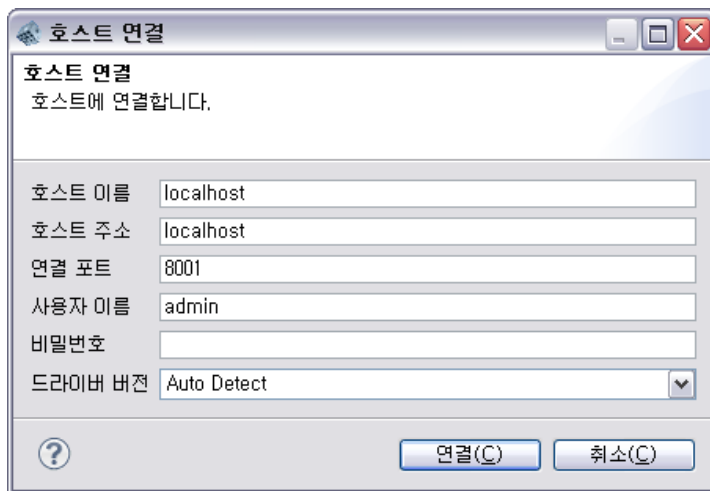
- "Cannot execute the current operation because the previous operation is already running."이라는 메시지가 나타나면, 서버가 이미 동일한 작업을 수행하고 있는 것이다. 이 메시지가 나타나면 해당 작업을 재시도한다.

호스트 등록 및 추가

매니저 클라이언트에서 접속할 호스트 사이트를 등록한다. 즉, CUBRID 매니저 서버가 구동되는 사이트 정보를 입력해야 한다. 기본으로 등록된 호스트는 로컬 호스트이고, 통신포트는 8001로 설정되며, 사용자 아이디와 비밀번호는 모두 "admin"으로 설정되어 있다. 처음 접속할 때 **admin** 아이디의 비밀번호를 변경해야 접속할 수 있으며, "admin"으로 재설정할 수는 없다.



호스트 탭에서 마우스 오른쪽 버튼 클릭하여 [호스트 추가]를 선택하면 호스트를 추가할 수 있다. [비밀번호 변경]을 선택하면 호스트 계정(예: admin)의 비밀번호를 변경할 수 있다.

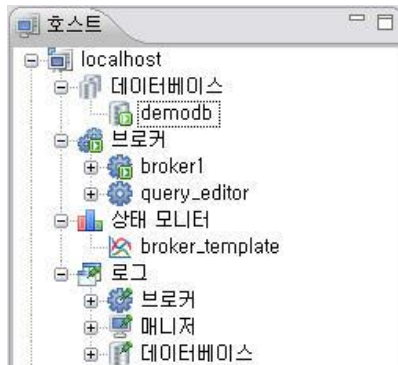
호스트 등록에 대한 자세한 내용은 [호스트 관리](#)를 참조한다.



호스트 연결

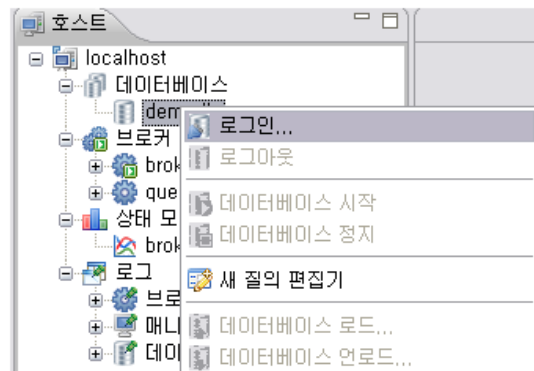
호스트 탐색 트리에 등록된 호스트를 더블 클릭하면 해당 호스트로 연결된다. 또는 해당 호스트를 마우스 오른쪽 버튼 클릭하여 [연결]을 선택하고, 사용자 이름을 확인한 후 [연결]을 클릭하면 호스트에 연결된다.

연결이 성공하면 호스트 아이콘이 에서 로 변경된다. 아래는 호스트(localhost)에 연결한 후 데이터베이스(demodb)를 구동 중인 상태의 트리 메뉴이다.

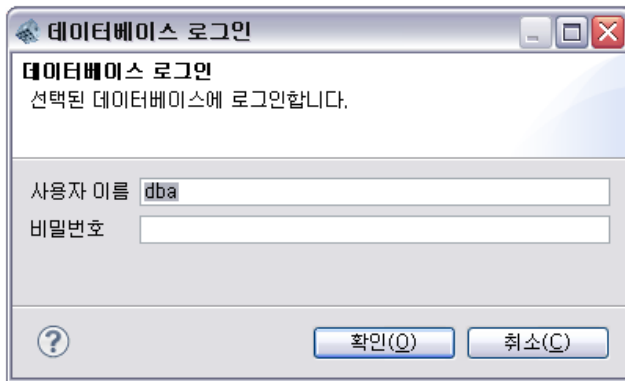


데이터베이스 연결

호스트 계정(예: admin)으로 호스트에 접속한 후, 해당 호스트에 존재하는 데이터베이스를 구동하려면 해당 데이터베이스에 로그인해야 한다. 호스트 탐색 트리에서 접속하려는 데이터베이스 이름을 마우스 오른쪽 버튼 클릭하고 [로그인]을 선택하거나, 더블 클릭한다.



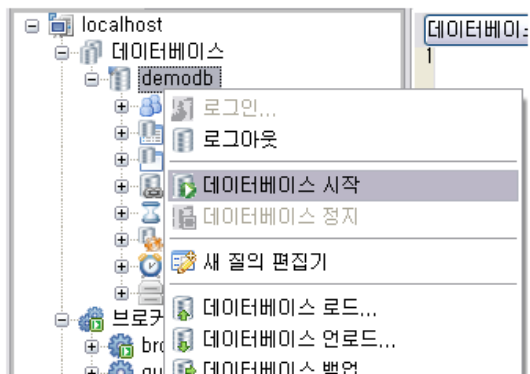
[데이터베이스 로그인] 창에서 데이터베이스 사용자 이름과 비밀번호를 입력한다. 기본 설정된 사용자 이름은 **dba**이며, 비밀번호는 없으므로 [확인]을 클릭한다. 데이터베이스 로그인에 성공하면 해당 사용자 권한에 맞는 트리 메뉴가 나타난다. 데이터베이스가 구동된 상태에서 [사용자]를 클릭하면 새로운 사용자를 추가하거나 기존 사용자(예: dba)의 비밀번호를 변경할 수 있다.



데이터베이스 시작

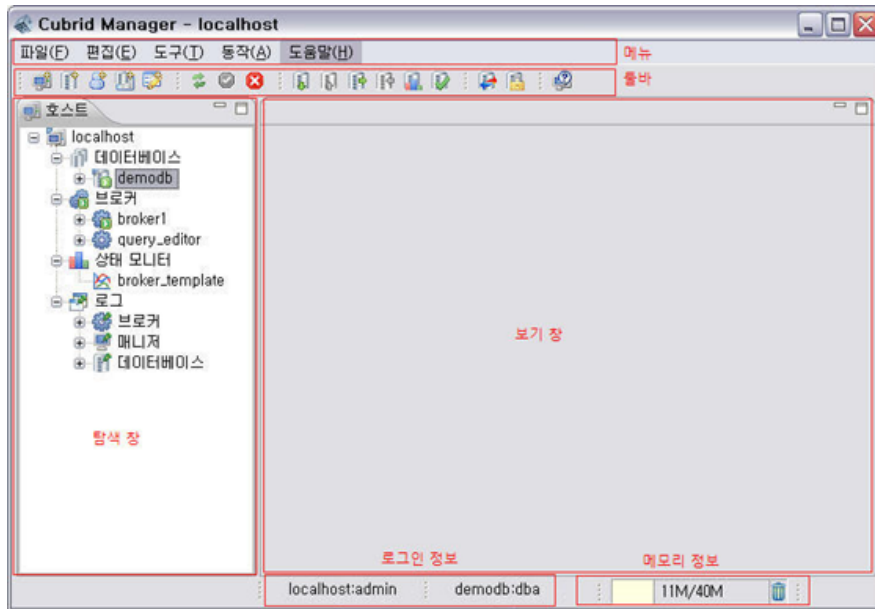
데이터베이스 로그인에 성공하면, 데이터베이스를 구동하여 질의를 수행할 수 있다.

해당 데이터베이스를 구동하기 위해서는 왼쪽 탐색 트리에서 데이터베이스 이름을 마우스 오른쪽 버튼 클릭하고 [데이터베이스 시작]을 선택한다.



CUBRID 매니저 화면 구성


CUBRID 매니저는 데이터베이스를 보다 편리하고 효율적으로 관리할 수 있는 도구이다. CUBRID 매니저 화면은 메뉴바, 툴바, 탐색 창, 보기 창, 로그인 정보, 메모리 정보로 구성된다. 자세한 내용은 [RCP 응용 프로그램](#)을 참고한다.



질의 편집기 사용

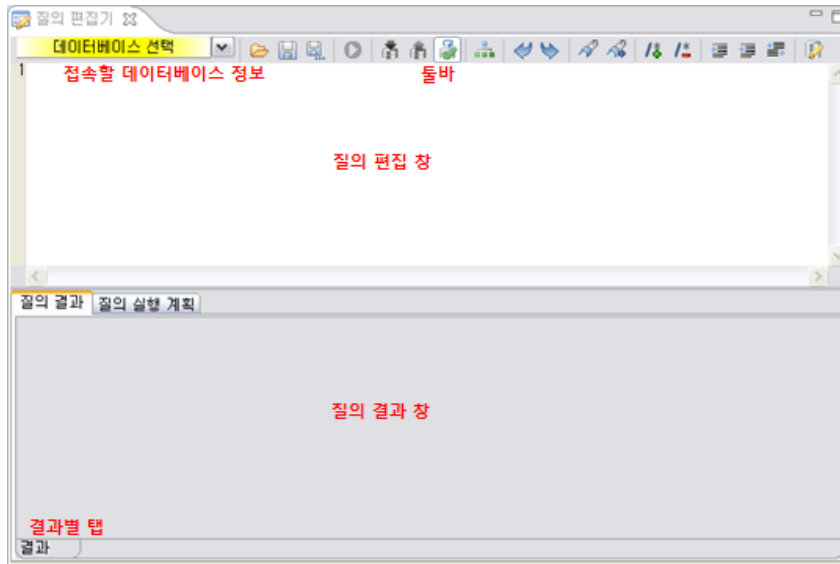
질의 편집기 실행

CUBRID 매니저 질의 편집기는 모든 **DML**, **DDL**, **DCL** 문의 수행을 지원하는 질의 도구로서, 보다 쉽고 편리하게 질의를 편집하고 실행할 수 있다.

질의 편집기를 실행하려면 메뉴에서 [도구] > [새 질의 편집기]를 선택하거나, 툴바에서 [새 질의 편집기] 를 클릭한다. 또는 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하여 [새 질의 편집기]를 선택한다.

질의 편집기 화면 구성

질의 편집기 화면은 위쪽의 질의 편집 창과 아래쪽의 질의 결과 창으로 구분할 수 있다. 질의 편집 창에서는 수행할 질의를 입력하고 편집하며, 질의 편집기에서 주로 사용하는 기능을 아이콘으로 보여주는 툴바를 포함한다. 질의 결과 창에서는 수행된 질의별 결과를 탭 형식으로 보여주며, 수행한 질의 및 해당 질의의 수행 시간을 확인할 수 있다. 질의 편집기에 대한 자세한 설명은 [질의 편집기](#)를 참고한다.



JDBC 를 이용한 프로그램 작성

JDBC 환경 설정

기본 환경

- JDK 1.5 이상
- CUBRID 2008 R1.0 이상
- CUBRID JDBC Driver 2008 R1.0 이상

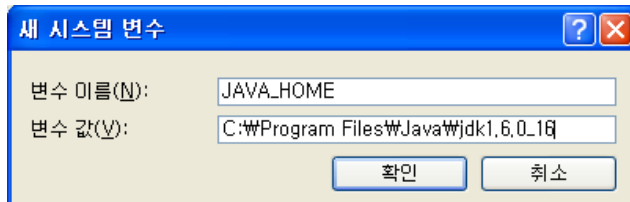
Java 설치 및 환경 변수 설정

시스템에 Java가 설치되어 있고 **JAVA_HOME** 환경 변수가 등록되어 있어야 한다. Java는 Developer Resources for Java Technology 사이트(<http://java.sun.com>)에서 다운로드할 수 있다. 이에 대한 자세한 설명은 [Java 저장 함수/프로시저 사용을 위한 환경 설정](#)을 참고한다.

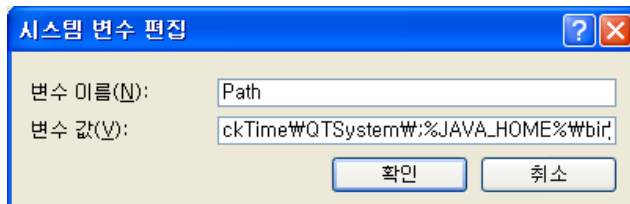
Windows 환경에서 환경 변수 설정

Java 설치 후 [내 컴퓨터]를 마우스 오른쪽 버튼 클릭하여 [속성]을 선택하면 [시스템 등록 정보] 대화 상자가 나타난다. [고급] 탭의 [환경 변수]를 클릭하면 나타나는 [환경 변수] 대화 상자가 나타난다.

[시스템 변수]에서 [새로 만들기]를 선택한다. [변수 이름]에 **JAVA_HOME**을 입력하고, 변수 값으로 Java 설치 경로(예: C:\Program Files\Java\jdk1.6.0_16)를 입력한 후 [확인]을 클릭한다.



[시스템 변수] 중 Path를 선택하고 [편집]을 클릭한다. [변수 값]에 **%JAVA_HOME%\bin**를 추가하고 [확인]을 클릭한다.



위의 방법을 사용하지 않고 다음과 같이 셸에서 **JAVA_HOME**과 **PATH** 환경 변수를 설정할 수도 있다.

```
set JAVA_HOME= C:\Program Files\Java\jdk1.6.0_16
```

```
set PATH=%PATH%;%JAVA_HOME%\bin
```

Linux 환경에서 환경 변수 설정

다음과 같이 Java가 설치된 **JAVA_HOME** 환경 변수로 디렉토리 경로(예: /usr/java/jdk1.6.0_16)를

설정하고, **PATH** 환경 변수에 **\$JAVA_HOME/bin**을 추가한다.

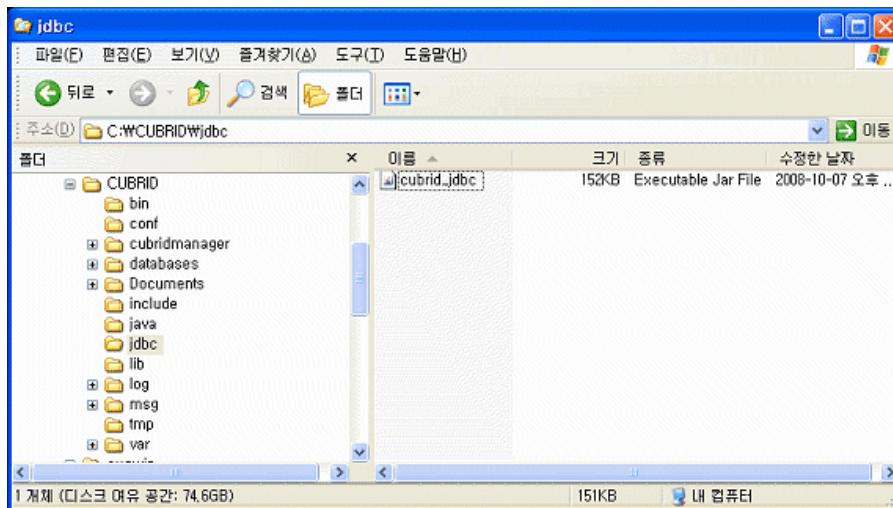
```
export JAVA_HOME=/usr/java/jdk1.6.0_16 //bash
export PATH=$JAVA_HOME/bin:$PATH //bash

setenv JAVA_HOME /usr/java/jdk1.6.0_16 //csh
set path = ($JAVA_HOME/bin $path) //csh
```

JDBC 드라이버 설정

JDBC를 사용하려면 CUBRID JDBC 드라이버가 존재하는 경로를 환경 변수 **CLASSPATH**에 추가해야 한다.

CUBRID JDBC 드라이버(**cuprid_jdbc.jar**)는 CUBRID 설치 디렉터리 아래의 jdbc 디렉터리에 위치한다.



Windows 환경에서 CLASSPATH 환경 변수 설정

```
set CLASSPATH=C:\CUBRID\jdbc\cuprid_jdbc.jar:.
```

Linux 환경에서 CLASSPATH 환경 변수 설정

```
export CLASSPATH=$HOME/CUBRID/jdbc/cuprid_jdbc.jar:.
```

주의 만약 JRE가 설치된 라이브러리 디렉터리(**\$JAVA_HOME/jre/lib/ext**)에 일반 CUBRID JDBC 드라이버가 설치되어 있다면, Java 저장 프로시저에서 사용하는 서버 사이드 JDBC 드라이버보다 먼저 로드되어 Java 저장 프로시저가 비정상적으로 구동될 수 있다. Java 저장 프로시저를 사용하는 환경에서는 JRE가 설치된 라이브러리 디렉터리(**\$JAVA_HOME/jre/lib/ext**)에 일반 CUBRID JDBC 드라이버를 설치하지 않도록 주의한다.

JDBC 예제 프로그램

다음은 JDBC 드라이버를 통해 CUBRID에 접속하여 데이터를 조회, 삽입하는 것을 간단하게 구성한 예제이다. 예제를 실행하려면 먼저 접속하고자 하는 데이터베이스와 CUBRID 브로커가 구동되어 있어야 한다. 예제에서는 설치 시 자동으로 생성되는 demodb 데이터베이스를 사용한다.

JDBC 드라이버 로드

CUBRID에 접속하기 위해서는 **Class**의 **forName()** 메소드를 사용하여 JDBC 드라이버를 로드해야 한다. 자세한 내용은 [CUBRID JDBC 드라이버](#)를 참고한다.

```
Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
```

데이터베이스 연결

JDBC 드라이버를 로드한 후 **DriverManager**의 **getConnection()** 메소드를 사용하여 데이터베이스와 연결한다. **Connection** 객체를 생성하기 위해서는 데이터베이스의 위치를 기술하기 위한 URL, 데이터베이스의 사용자 이름, 암호 등의 정보가 지정되어야 한다. 자세한 내용은 [연결 설정](#)을 참고한다.

```
String url = "jdbc:cubrid:localhost:30000:demodb:::";
String userid = "dba";
String password = "";

Connection conn = DriverManager.getConnection(url,userid,password);
```

데이터베이스 조작(질의 수행 및 ResultSet 처리)

접속된 데이터베이스에 질의문을 전달하고 실행시키기 위하여 **Statement**, **PreparedStatement**, **CallableStatement** 객체를 생성한다. **Statement** 객체가 생성되면, **Statement** 객체의 **executeQuery()** 메소드나 **executeUpdate()** 메소드를 사용하여 질의문을 실행한다. **next()** 메소드를 사용하여 **ExecuteQuery()** 메소드의 결과로 반환된 **ResultSet**의 다음 행을 처리할 수 있다. 보다 자세한 내용은 [JDBC API](#)를 참고한다.

데이터베이스 연결 해제

각 객체에 대해 **close()** 메소드를 수행하여 데이터베이스와의 연결을 해제할 수 있다.

예제 1

아래는 테이블을 생성하고, prepared statement로 질의문을 수행한 후, 질의를 롤백시키는 예제코드이며, **getConnection()** 메소드의 인자값을 적절하게 수정하여 실행할 수 있다.

```
import java.util.*;
import java.sql.*;

public class Basic {
    public static Connection connect() {
        Connection conn = null;
        try {
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
            conn = DriverManager.getConnection("jdbc:cubrid:localhost:30000:::", "dba", "");
            conn.setAutoCommit (false);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return conn;
    }
}
```

```

    } catch ( Exception e ) {
        System.err.println("SQLException : " + e.getMessage());
    }
    return conn;
}

public static void printdata(ResultSet rs) {
    try {
        ResultSetMetaData rsmd = null;

        rsmd = rs.getMetaData();
        int numberOfColumn = rsmd.getColumnCount();

        while (rs.next () ) {
            for(int j=1; j<=numberOfColumn; j++ )
                System.out.print(rs.getString(j) + "  ");
            System.out.println("");
        }
    } catch ( Exception e ) {
        System.err.println("SQLException : " + e.getMessage());
    }
}

public static void main(String[] args) throws Exception {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;
    PreparedStatement preStmt = null;

    try {
        conn = connect();

        stmt = conn.createStatement();
        stmt.executeUpdate("create class xoo ( a int, b int, c char(10))");

        preStmt = conn.prepareStatement("insert into xoo values(?,?,'''100''')");
        preStmt.setInt (1, 1) ;
        preStmt.setInt (2, 1*10) ;
        int rst = preStmt.executeUpdate () ;

        rs = stmt.executeQuery("select a,b,c from xoo" );

        printdata(rs);

        conn.rollback();
        stmt.close();
        conn.close();
    } catch ( Exception e ) {
        conn.rollback();
        System.err.println("SQLException : " + e.getMessage());
    } finally {
        if ( conn != null ) conn.close();
    }
}
}

```

예제 2

다음은 CUBRID 설치 시 기본 제공되는 demodb에 접속하여 **SELECT** 질의를 수행하는 예제이다.

```

import java.sql.*;
public class SelectData {
    public static void main(String[] args) throws Exception {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rs = null;
        try {
            // CUBRID에 Connect
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
            conn =
            DriverManager.getConnection("jdbc:cubrid:localhost:30000:demodb:::", "dba", "");

```



```

String sql = "select name, players from event";
stmt = conn.createStatement();
rs = stmt.executeQuery(sql);
while(rs.next()) {
    String name = rs.getString("name");
    String players = rs.getString("players");
    System.out.println("name ==> " + name);
    System.out.println("Number of players==> " + players);
    System.out.println("\n===== \n");
}
rs.close();
stmt.close();
conn.close();
} catch ( SQLException e ) {
    System.err.println(e.getMessage());
} catch ( Exception e ) {
    System.err.println(e.getMessage());
} finally {
    if ( conn != null ) conn.close();
}
}
}

```

예제 3

다음은 CUBRID 설치 시 기본 제공되는 demodb에 접속하여 **INSERT** 질의를 수행하는 예제이다. 데이터 삭제 및 갱신 방법은 데이터 삽입 방법과 동일하므로 아래 코드에서 질의문만 변경하여 사용할 수 있다.

```

import java.sql.*;
public class insertData {
    public static void main(String[] args) throws Exception {
        Connection conn = null;
        Statement stmt = null;
        try {
            // CUBRID에 Connect
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
            conn =
DriverManager.getConnection("jdbc:cubrid:localhost:30000:demodb::", "dba", "");
            String sql = "insert into olympic(host_year, host_nation, host_city,
opening date,          closing date) values (2008, 'China', 'Beijing', to date('08-08-
2008','mm-dd-yyyy'),          to_date('08-24-2008','mm-dd-yyyy'))";
            stmt = conn.createStatement();
            stmt.executeUpdate(sql);
            System.out.println("데이터가 입력되었습니다.");
            stmt.close();
        } catch ( SQLException e ) {
            System.err.println(e.getMessage());
        } catch ( Exception e ) {
            System.err.println(e.getMessage());
        } finally {
            if ( conn != null ) conn.close();
        }
    }
}

```

PHP 를 이용한 프로그램 작성

PHP 모듈 설치

CUBRID 홈페이지(<http://www.cubrid.com>)에서 PHP 모듈을 다운로드한다.

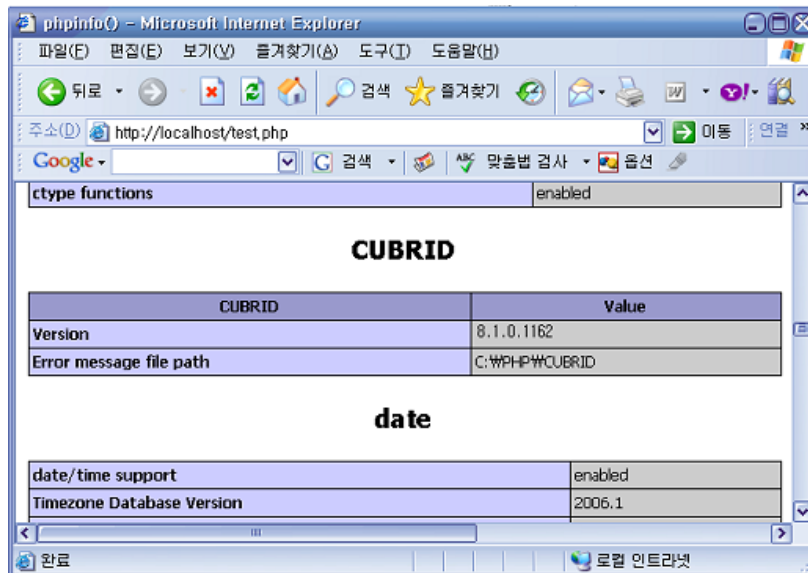
Windows 버전 설치

PHP가 설치되어 있는 디렉터리에 CUBRID라는 디렉터리를 만든 후 다운로드한 cubrid_err.msg 파일과 cubrid_php 버전 .dll 파일을 복사한다.

다운로드한 CUBRID PHP 모듈 버전이 5.1.4라면, 다음 예제와 같이 php.ini를 편집하여 관련 설정 값을 추가한다.

```
extension_dir=C:\PHP\CUBRID
extension=cubrid_php5.1.4.dll
[CUBRID]
cubrid_err_path=C:\PHP\CUBRID
```

설정이 완료되면 웹 서버를 재시작한다. PHP의 phpinfo() 함수를 사용하여 PHP 설정 내용을 확인했을 때 다음과 같이 CUBRID가 보이면 정상적으로 설치가 완료된 것이다.



Linux 버전 설치

Linux 버전은 Windows 버전과는 달리 소스코드가 제공된다. 사용자는 CUBRID

홈페이지(<http://www.cubrid.com>)에서 소스 코드를 다운로드한 다음 압축 파일에 포함되어 있는 INSTALL

파일을 참고하여 설치를 진행한다. Linux 버전도 다음과 같이 php.ini 파일을 편집하고 저장한 후 웹 서버를 재시작한다.

```
; Directory in which the loadable extensions (modules) reside.
extension_dir = /usr/lib/php5/lib/php/ext
extension=cubrid.so
;Windows의 경우 extension=cubrid.dll
[CUBRID]
cubrid.err_path=/home/cubrid/CUBRID/msg
```

phpinfo() 함수를 사용해서 확인한다.

```
[root@localhost]#vi test.php
```

웹 브라우저를 통해서 확인했을 때 Windows 버전과 마찬가지로 CUBRID가 보인다면 정상적으로 설치가 완료된 것이다.

PHP 예제 프로그램

PHP와 CUBRID를 연동하는 것을 간단하게 구현한 예제다. 가장 기본적인 기능과 특별히 주의해야 할 내용만을 다룬다. 이 예제를 수행하기 위해서는 먼저 접속하려는 데이터베이스와 CUBRID 브로커가 구동되어 있어야 한다. 이 예제에서는 설치 시 자동 생성되는 **demodb** 데이터베이스를 사용한다.

조회, 데이터 가져오기 예제

```
<html>
<head>
<meta http-equiv='content-type' content='text/html; charset=euc-kr'>
</head>
<body>
<center>
<table border=2>
<?
    // CUBRID에 접속하기 위한 서버 정보를 설정한다. host_ip는 CUBRID 브로커가 설치된 곳의 IP
    address(여기서는 localhost)이며, host_port는 CUBRID 브로커의 port 번호이다. port 번호는 설치 시
    디폴트 값이다. 자세한 내용은 관리자 안내서를 참조한다.
    $host_ip = "localhost";
    $host_port = 30000;
    $db_name = "demodb";
    // CUBRID 서버에 접속한다. 그러나 실제로 접속하지는 않고, 접속 정보만 보관한다. 실제로 접속하지 않는
    것은 3-tier 구조이므로 transaction 처리를 효율적으로 수행하기 위해서다.
    $cubrid_con = @cubrid_connect($host_ip, $host_port, $db_name);
    if (!$cubrid_con) {
        echo "DB 접속 오류";
        exit;
    }
?>
<?
    $sql = "select sports, count(players) as players from event group by sports";
    // SQL 문장에 대하여 CUBRID 서버로 결과를 요청한다. 이 때 실제로 CUBRID 서버로 접속한다.
    $result = cubrid_execute($cubrid_con, $sql);
    if ($result) {
        // SQL 처리 결과로 만들어진 결과 집합에 대하여 컬럼 이름을 얻어온다.
        $columns = cubrid_column_names($result);
        // SQL 처리 결과로 만들어진 결과 집합에 대하여 컬럼개수를 얻어온다.
        $num_fields = cubrid_num_cols($result);
        // 화면에 결과 집합의 각 컬럼 이름을 보여준다.
        echo("<tr>");
```

```

        while (list($key, $colname) = each($columns)) {
            echo("<td align=center>$colname</td>");
        }
        echo("</tr>");
        // 결과 집합에서 결과를 가져온다.
        while ($row = cubrid_fetch($result)) {
            echo("<tr>");
            for ($i = 0; $i < $num_fields; $i++) {
                echo("<td align=center>");
                echo($row[$i]);
                echo("</td>");
            }
            echo("</tr>");
        }
    }

    // CUBRID의 PHP 모듈은 3-tier 구조로 동작하며, transaction 처리를 위하여 SELECT를 하더라도
transaction의 일부로 처리한다. 따라서 원활한 동작과 성능을 위하여 SELECT 처리를 했더라도, commit(또는
rollback) 처리를 하여 transaction을 정리해야 한다.
    cubrid_commit($cubrid_con);
    cubrid_disconnect($cubrid_con);
?>
</body></html>

```

데이터 삽입 예제

```

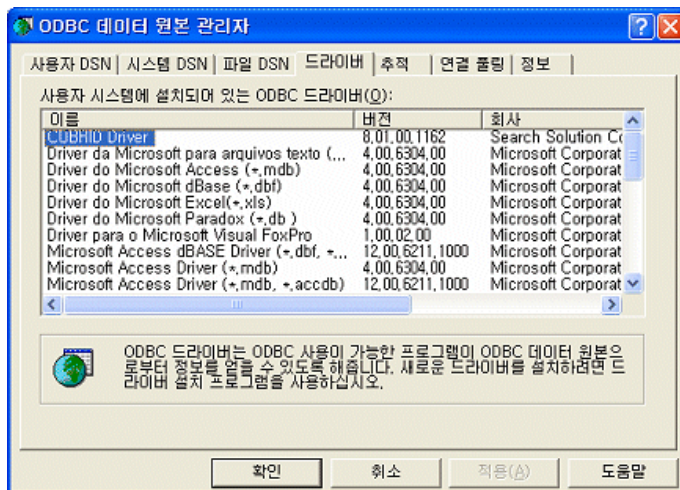
<html>
<head>
<meta http-equiv='content-type' content='text/html; charset=euc-kr'>
</head>
<body>
<center>
<table border=2>
<?
    $host_ip = "localhost";
    $host_port = 30000;
    $db_name = "demodb";
    $cubrid_con = @cubrid_connect($host_ip, $host_port, $db_name);
    if (!$cubrid_con) {
        echo "DB 접속 오류";
        exit;
    }
?>
<?
    $sql = "insert into olympic (host_year,host_nation,host_city,opening_date,closing_date)
values (2008, 'China', 'Beijing', to_date('08-08-2008','mm-dd-yyyy'),to_date('08-24-
2008','mm-dd-yyyy'))";
    $result = cubrid_execute($cubrid_con, $sql);
    if ($result) {
        // 정상적으로 처리 되었으므로 commit 처리한다.
        cubrid_commit($cubrid_con);
        echo("성공적으로 입력되었습니다");
    } else {
        // 에러가 발생했으므로, 에러 메시지를 출력한 후 rollback 처리한다.
        echo(cubrid_error_msg());
        cubrid_commit($cubrid_con);
    }
    cubrid_disconnect($cubrid_con);
?>
</body></html>

```

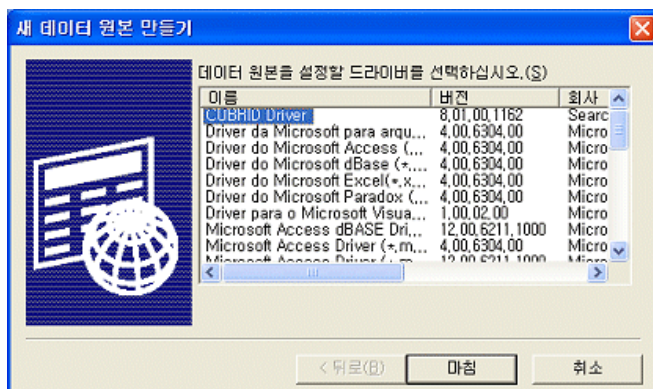
ODBC 와 ASP 를 이용한 프로그램 작성

ODBC 와 ASP 환경 설정


ODBC 요구사항 처리를 위한 클라이언트 환경을 설정하려면 CUBRID ODBC Driver가 설치되어 있어야 한다. CUBRID ODBC Driver는 CUBRID 설치 시 자동으로 설치된다. 서버나 클라이언트 어느 것을 설치해도 된다. 설치된 CUBRID ODBC Driver를 확인하려면 [제어판]의 [관리 도구]에서 [데이터 원본(ODBC)]을 실행한다.



CUBRID ODBC Driver가 확인되었다면 응용 프로그램에서 접속하려는 데이터베이스로 DSN을 설정한다. DSN 설정을 위해 ODBC 데이터 원본 관리자에서 [추가] 버튼을 클릭하면 다음과 같은 대화 상자가 나타난다. 'CUBRID Driver'를 선택하고 [마침] 버튼을 클릭한다.



다음과 같이 [Config CUBRID Data Source] 팝업 창이 나타나면 [DB Name]에는 접속하려는 데이터베이스 이름을, [Server Port]에는 CUBRID 브로커의 포트 번호를 입력하고 [OK] 버튼을 클릭한다.



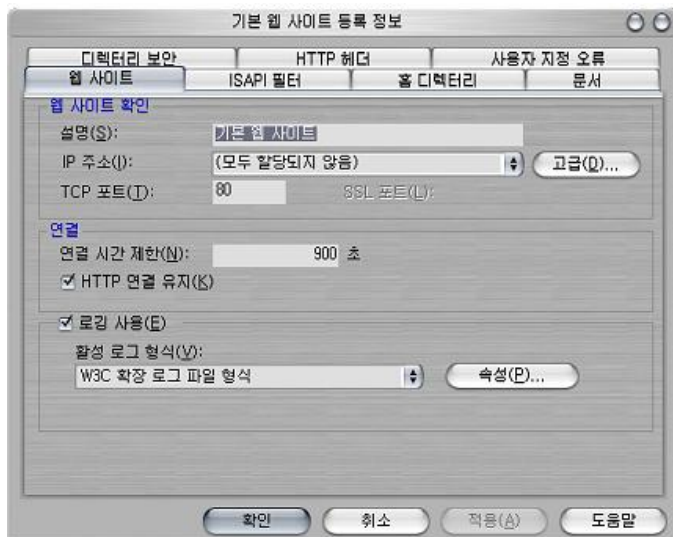
The image shows a 'Config CUBRID Data Sources' dialog box with the following fields and values:

Field	Value
DSN	CUBRID
Description	cubrid_test
DB Name	demodb
DB User	public
Password	
Server Address	localhost
Server Port	33000
FETCH_SIZE	100

Buttons: OK, Cancel

ASP 예제 프로그램

ASP 예제를 실행할 가상 디렉터리의 '기본 웹 사이트' 항목에서 마우스 오른쪽 버튼을 클릭한 뒤 [속성]을 클릭한다.



The image shows the '기본 웹 사이트 등록 정보' (Basic Web Site Registration Information) dialog box. The '웹 사이트' (Web Site) tab is selected. The '웹 사이트 확인' (Web Site Check) section is expanded, showing the following settings:

Section	Field	Value
웹 사이트 확인	설명(S):	기본 웹 사이트
	IP 주소(I):	(모두 할당되지 않음)
	TCP 포트(T):	80
	SSL 포트(L):	
연결	연결 시간 제한(N):	900 초
	<input checked="" type="checkbox"/> HTTP 연결 유지(K)	
로깅 사용(E)	<input checked="" type="checkbox"/> 로깅 사용(E)	
	활성 로그 형식(V):	W3C 확장 로그 파일 형식

Buttons: 확인, 취소, 적용(A), 도움말

위의 그림에서 **웹사이트 확인** 아래 **IP 주소**를 '(모두 할당되지 않음)'으로 선택하면 localhost로 인식한다. 특정한 IP 주소를 통해 예제를 확인하려면 해당 IP에 해당 디렉터리를 가상 디렉터리로 인식시키고 등록 정보에 IP 주소를 등록한다.

다음은 IP 주소를 localhost로 설정한 예제이다.

예제

아래의 예제 코드를 cubrid.asp로 만들고 가상 디렉터리에 저장한다.

```
<HTML>
  <HEAD>
    <meta http-equiv="Content-Type" content="text/html; charset=EUC-KR">
    <title>CUBRID Query Test Page</title>
  </HEAD>

  <BODY topmargin="0" leftmargin="0">

    <table border="0" width="748" cellspacing="0" cellpadding="0">
      <tr>
        <td width="200"></td>
        <td width="287">
          <p align="center"><font size="3" face="Times New Roman"><b><font
color="#FF0000">CUBRID</font>Query Test</b></font></td>
        <td width="200"></td>
      </tr>
    </table>

    <form action="cubrid.asp" method="post" >
    <table border="1" width="700" cellspacing="0" cellpadding="0" height="45">
      <tr>
        <td width="113" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">SERVER IP</font></td>
        <td width="78" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">Broker PORT</font></td>
        <td width="148" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">DB NAME</font></td>
        <td width="113" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">DB USER</font></td>
        <td width="113" valign="bottom" height="16" bgcolor="#DBD7BD"
bordercolorlight="#FFFFCC"><font size="2">DB PASS</font></td>
        <td width="80" height="37" rowspan="4" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED">
          <p><input type="submit" value="실행하기" name="B1" tabindex="7"></p></td>
      </tr>
      <tr>
        <td width="113" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="text" name="server ip" size="20" tabindex="1" maxlength="15"
value="<%=Request("server_ip")%>"></font></td>
        <td width="78" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="text" name="cas port" size="15" tabindex="2" maxlength="6"
value="<%=Request("cas port")%>"></font></td>
        <td width="148" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="text" name="db name" size="20" tabindex="3" maxlength="20"
value="<%=Request("db name")%>"></font></td>
        <td width="113" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="text" name="db user" size="15" tabindex="4"
value="<%=Request("db user")%>"></font></td>
        <td width="113" height="1" bordercolorlight="#FFFFCC" bgcolor="#F5F5ED"><font
size="2"><input type="password" name="db pass" size="15" tabindex="5"
value="<%=Request("db pass")%>"></font></td>
      </tr>
      <tr>
        <td width="573" colspan="5" valign="bottom" height="18" bordercolorlight="#FFFFCC"
bgcolor="#DBD7BD"><font size="2">QUERY</font></td>
      </tr>
      <tr>
        <td width="573" colspan="5" height="25" bordercolorlight="#FFFFCC">
```

```

bgcolor="#F5F5ED"><textarea rows="3" name="query" cols="92"
tabindex="6"><%=Request("query")%></textarea></td>
</tr>
</table>
</form>
<hr>

</BODY>
</HTML>

<%
' DSN과 SQL문을 가져온다.
strIP = Request( "server ip" )
strPort = Request( "cas port" )
strUser = Request( "db_user" )
strPass = Request( "db_pass" )
strName = Request( "db_name" )
strQuery = Request( "query" )

if strIP = "" then
Response.Write "SERVER_IP를 입력하세요"

Response.End ' IP가 없으면 페이지 종료
end if
if strPort = "" then
Response.Write "Port 번호를 입력하세요"

Response.End ' Port가 없으면 페이지 종료
end if
if strUser = "" then
Response.Write "DB_USER를 입력하세요"

Response.End ' DB_User가 없으면 페이지 종료
end if
if strName = "" then
Response.Write "DB_NAME을 입력하세요"

Response.End ' DB_NAME이 없으면 페이지 종료
end if
if strQuery = "" then
Response.Write "확인하고자 하는 Query를 입력하세요"

Response.End ' Query가 없으면 페이지 종료
end if

' 연결 객체 생성
strDsn = "driver={CUBRID Driver};server=" & strIP & ";port=" & strPort & ";uid=" &
strUser & ";pwd=" & strPass & ";db_name=" & strName & ";"
' DB연결
Set DBConn = Server.CreateObject("ADODB.Connection")
DBConn.Open strDsn

' SQL 실행
Set rs = DBConn.Execute( strQuery )
' SQL문에 따라 메시지 보기
if InStr(Ucase(strQuery),"INSERT")>0 then
Response.Write "레코드가 추가되었습니다."
Response.End
end if

if InStr(Ucase(strQuery),"DELETE")>0 then
Response.Write "레코드가 삭제되었습니다."
Response.End
end if

if InStr(Ucase(strQuery),"UPDATE")>0 then
Response.Write "레코드가 수정되었습니다."
Response.End
end if
%>
</table>

```

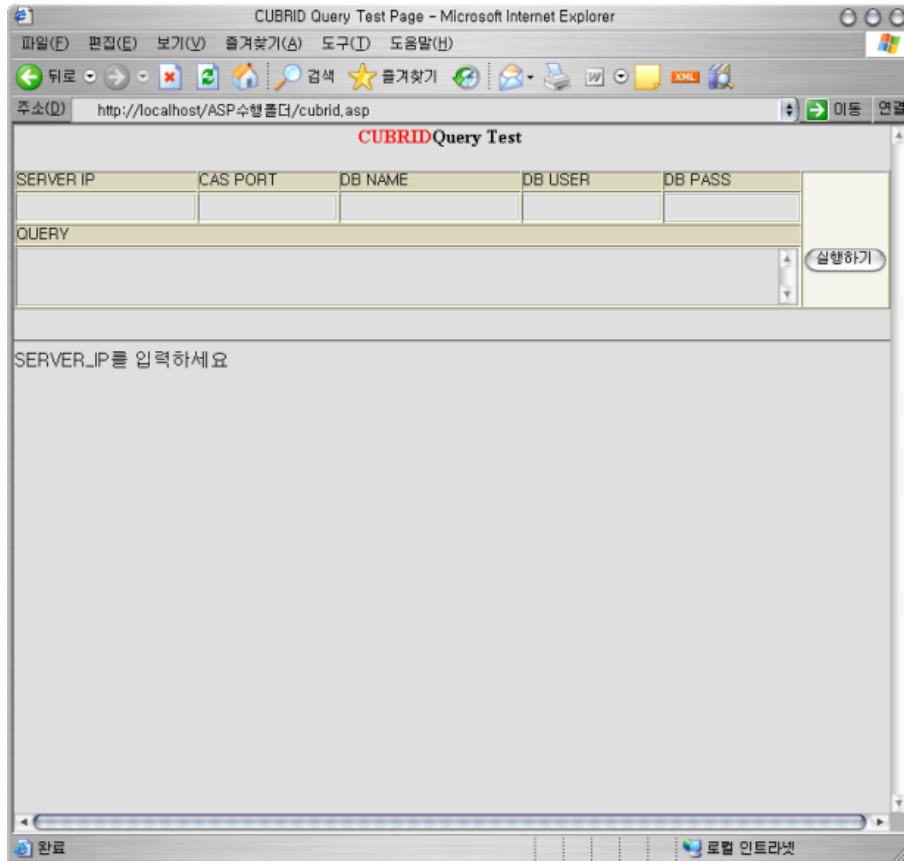


```

<%
' 필드 이름 보여주기
Response.Write "<tr bgColor=#f3f3f3>"
For index =0 to ( rs.fields.count-1 )
    Response.Write "<td><b>" & rs.fields(index).name & "</b></td>"
Next
Response.Write "</tr>"
' 필드 값 보여주기
Do While Not rs.EOF
    Response.Write "<tr bgColor=#f3f3f3>"
    For index =0 to ( rs.fields.count-1 )
        Response.Write "<td>" & rs(index) & "</td>"
    Next
    Response.Write "</tr>"
    rs.MoveNext
Loop
%>
<%
    set  rs = nothing
%>
</table>

```

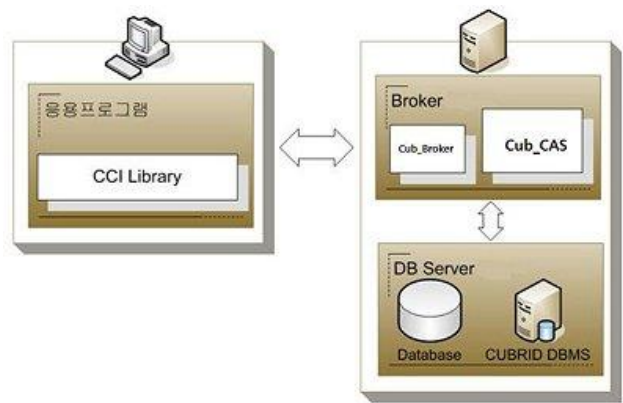
http://localhost/ASP수행폴더/cubrid.asp에 접속하면 수행 결과를 확인할 수 있다. 위의 ASP 예제 코드를 실행하면 다음과 같은 결과를 출력한다. 해당 항목에 알맞은 값을 넣고 Query 항목에 질의문을 입력하고 [실행하기]를 클릭하면 하단에 질의 문의 결과가 출력된다.



CCI 를 이용한 프로그램 작성

CCI 라이브러리

CCI 라이브러리는 CUBRID에서 제공되는 C 언어 인터페이스다. CCI는 브로커를 통해서 접속하므로 다른 인터페이스인 JDBC, PHP, ODBC 등과 동일하게 관리될 수 있다. 실제로 CCI는 PHP, ODBC, Python, Ruby 인터페이스를 구현하는 하부 구조로 사용되었다.



CCI 설치 및 구성

CCI 라이브러리는 CUBRID 설치 패키지에 포함되어 배포되면, 아래 디렉터리에 위치한다.

	Windows	Unix/Linux
C 헤더파일	include/cas_cci.h	include/cas_cci.h
정적 라이브러리	lib/cascci.lib	lib/libcascci.a
동적 라이브러리	lib/cascci.lib bin/cascci.dll	lib/libcascci.so

CCI 사용

CCI 응용 프로그램의 기본 구조

CCI를 이용하는 응용 프로그램은 기본적으로 CAS와 연결하기, 질의 준비, 질의 수행, 응답 처리, 연결 끊기의 과정을 통해 CUBRID를 이용한다. 각 과정에서 CCI는 연결 핸들(connection handle), 질의 핸들(query handle), 응답 핸들(response handle)을 통해 응용 프로그램과 소통한다.

다음은 CCI를 이용하는 응용 프로그램의 작성 단계와 관련 함수를 나타낸다. 자세한 내용은 [CCI API](#)를 참고한다.

- 데이터베이스 연결 핸들 열기(관련 함수: [cci_connect](#))
- prepared statement를 위한 요청 핸들 얻기(관련 함수: [cci_prepare](#))
- prepared statement에 데이터 바인딩하기(관련 함수: [cci_bind_param](#))
- prepared statement 실행하기(관련 함수: [cci_execute](#))
- 실행 결과 처리하기(관련 함수: [cci_cursor](#), [cci_fetch](#), [cci_get_data](#), [cci_get_result_info](#))
- 요청 핸들 닫기(관련 함수: [cci_close_req_handle](#))
- 데이터베이스 연결 핸들 닫기(관련 함수: [cci_disconnect](#))

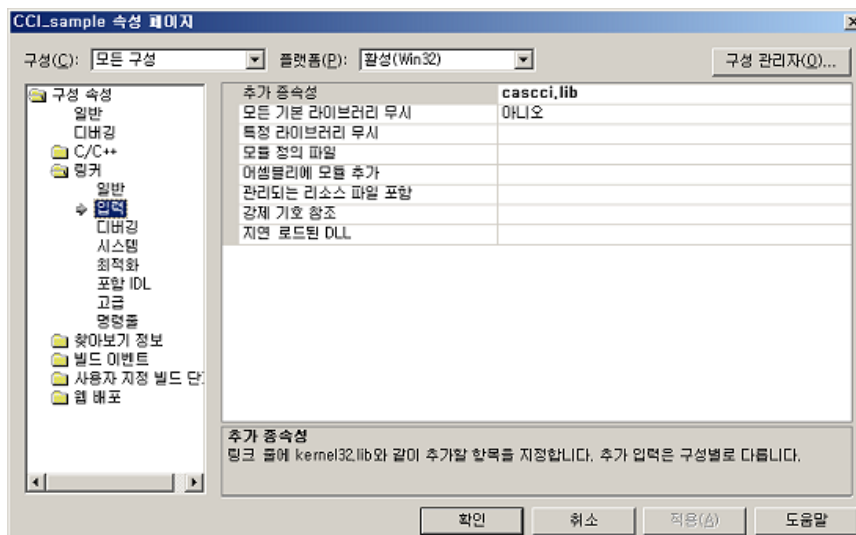
사용 방법

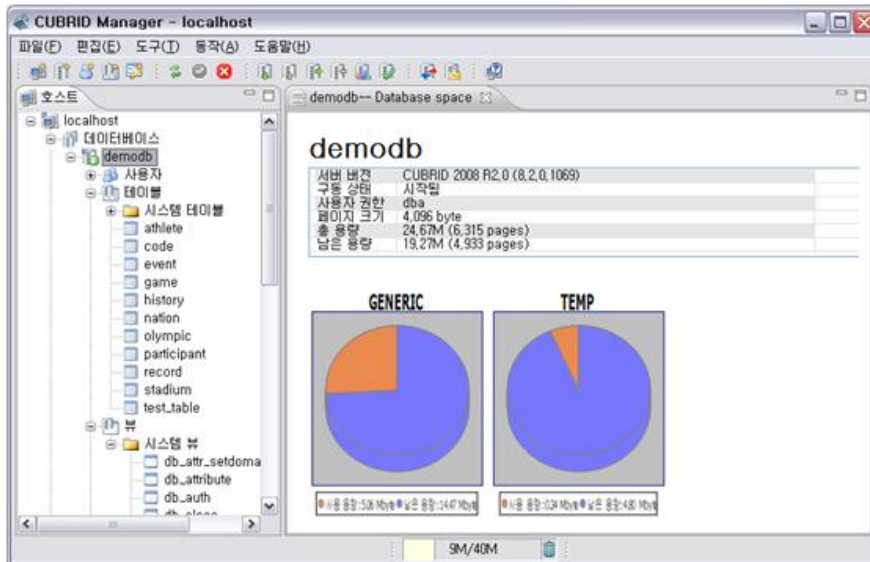
CCI를 이용한 응용 프로그램을 작성했다면 프로그램 특성에 따라 정적 링크 형태로 프로그램을 수행시킬 것인지, 아니면 동적으로 CCI를 호출하여 사용할 것인지를 결정하여 프로그램을 빌드한다. [CCI 설치 및 구성](#)의 표를 참조하여 사용할 라이브러리를 결정한다.

다음은 유닉스/Linux에서 동적인 라이브러리를 사용하여 링크하는 Makefile의 예제이다.

```
CC=gcc
CFLAGS = -g -Wall -I. -I$CUBRID/include
LDFLAGS = -L$CUBRID/lib -lcascci -lnsl
TEST_OBJS = test.o
EXES = test
all: $(EXES)
test: $(TEST_OBJS)
$(CC) -o $@ $(TEST_OBJS) $(LDFLAGS)
```

아래는 Windows에서 정적 라이브러리를 적용하기 위한 설정이다.





다음은 브로커가 가동 중인 상태를 CUBRID 매니저를 통해서 확인한 것이다.

NAME	STA...	PID	PORT	CAS	QU...	CPU	REQ	AUTO
broker1	ON	1488	33000	5	0	0.00	33	ON
quer...	ON	24856	30000	5	0	0.00	0	ON

빌드

프로그램 소스와 Makefile이 준비된 상태에서 make를 수행하면 test라는 실행 파일이 생성된다. 정적 라이브러리를 사용하면 추가로 파일을 배포할 필요가 없고 속도가 빠르다. 하지만, 프로그램의 크기와 메모리 사용량이 커지는 단점이 있다. 동적 라이브러리를 사용하면 성능상의 오버헤드는 있지만, 메모리와 프로그램 크기에 있어 최적화를 이룰 수 있다.

다음은 Linux에서 make를 사용하지 않고 동적인 라이브러리를 사용하여 테스트 프로그램을 빌드하는 명령 행의 예제이다.

```
cc -o test test.c -I$CUBRID/include -L$CUBRID/lib -lnsl -lcascci
```

예제 코드

```
#include <stdio.h>
#include <cas cci.h>
char *cci_client_name = "test";
```

```

int main (int argc, char *argv[])
{
    int con = 0, req = 0, col count = 0, res, ind, i;
    T CCI ERROR error;
    T CCI COL INFO *res col info;
    T CCI SQLX CMD cmd type;
    char *buffer, db_ver[16];
    printf("Program started!\n");
    if ((con=cci_connect("localhost", 30000, "demodb", "PUBLIC", ""))<0) {
        printf( "s(%d): cci_connect fail\n", __FILE__, __LINE__);
        return -1;
    }

    if ((res=cci_get_db_version(con, db_ver, sizeof(db_ver)))<0) {
        printf( "s(%d): cci_get_db_version fail\n", __FILE__, __LINE__);
        goto handle_error;
    }
    printf("DB Version is %s\n",db_ver);
    if ((req=cci_prepare(con, "select * from event", 0,&error))<0) {
        printf( "s(%d): cci_prepare fail(%d)\n", __FILE__, __LINE__,error.err_code);
        goto handle_error;
    }
    printf("Prepare ok! (%d)\n",req);
    res col info = cci_get_result_info(req, &cmd type, &col count);
    if (!res_col_info) {
        printf( "s(%d): cci_get_result_info fail\n", __FILE__, __LINE__);
        goto handle_error;
    }

    printf("Result column information\n"
           "=====\n");
    for (i=1; i<=col_count; i++) {
        printf("name:%s type:%d(precision:%d scale:%d)\n",
               CCI_GET_RESULT_INFO_NAME(res_col_info, i),
               CCI_GET_RESULT_INFO_TYPE(res_col_info, i),
               CCI_GET_RESULT_INFO_PRECISION(res_col_info, i),
               CCI_GET_RESULT_INFO_SCALE(res_col_info, i));
    }
    printf("=====\n");
    if ((res=cci_execute(req, 0, 0, &error))<0) {
        printf( "s(%d): cci_execute fail(%d)\n", __FILE__, __LINE__,error.err_code);
        goto handle_error;
    }
    if ((res=cci_fetch_size(req, 100))<0) {
        printf( "s(%d): cci_fetch_size fail\n", __FILE__, __LINE__);
        goto handle_error;
    }

    while (1) {
        res = cci_cursor(req, 1, CCI_CURSOR_CURRENT, &error);
        if (res == CCI_ER_NO_MORE_DATA) {
            printf("Query END!\n");
            break;
        }
        if (res<0) {
            printf( "s(%d): cci_cursor fail(%d)\n", __FILE__, __LINE__,error.err_code);
            goto handle_error;
        }

        if ((res=cci_fetch(req, &error))<0) {
            printf( "s(%d): cci_fetch fail(%d)\n", __FILE__, __LINE__,error.err_code);
            goto handle_error;
        }

        for (i=1; i<=col_count; i++) {
            if ((res=cci_get_data(req, i, CCI_A_TYPE_STR, &buffer, &ind))<0) {
                printf( "s(%d): cci_get_data fail\n", __FILE__, __LINE__);
                goto handle_error;
            }
            printf("%s\t", buffer);
        }
        printf("\n");
    }
}

```

```

    if ((res=cci_close_req_handle(req))<0) {
        printf( "%s(%d): cci_close_req_handle fail", __FILE__, __LINE__ );
        goto handle_error;
    }
    if ((res=cci_disconnect(con, &error))<0) {
        printf( "%s(%d): cci_disconnect fail(%d)", __FILE__, __LINE__, error.err_code);
        goto handle_error;
    }
    printf("Program ended!\n");
    return 0;
}

handle_error:
    if (req > 0)
        cci_close_req_handle(req);
    if (con > 0)
        cci_disconnect(con, &error);
    printf("Program failed!\n");
    return -1;
}

```


CSQL 인터프리터

CUBRID에서 SQL 문을 실행하는 방법은 GUI (Graphical User Interface) 기반의 CUBRID 매니저를 이용하거나, 콘솔 기반의 CSQL 인터프리터를 이용하는 것이다.

CSQL은 CUBRID에서 명령어 방식으로 SQL 문을 사용할 수 있는 프로그램이다. 여기에서는 CSQL 인터프리터의 간단한 사용법과 관련 명령어를 설명한다.

- CSQL 인터프리터 소개
- CSQL 실행
- 세션 명령어

CSQL 인터프리터 소개

SQL 사용을 위한 도구

CSQL 인터프리터는 CUBRID와 함께 설치되는 대화형(interactive) 방식과 일괄 수행(batch) 방식으로 SQL 질의를 수행하고 수행 결과를 조회할 수 있는 프로그램이다. CSQL 인터프리터는 명령어 라인 입력 방식의 인터페이스를 제공하며, 입력된 SQL 문장과 그 결과는 나중에 사용하기 위해서 파일에 저장할 수도 있다.

CSQL 인터프리터는 CUBRID를 사용하는 가장 기본적인 손쉬운 방법이다. CUBRID를 사용하는데 제공되는 다양한 다양한 API (JDBC, ODBC, PHP, CCI 등)를 활용하여 데이터베이스 응용 프로그램을 작성할 수 있다. 또한, CUBRID에서 제공하는 관리 및 질의 도구인 CUBRID 매니저를 사용할 수도 있다. 사용자는 CSQL 인터프리터가 제공하는 터미널 기반의 환경에서 SQL 질의를 생성하고, 수행 결과를 조회할 수 있다.

CSQL 인터프리터는 CUBRID 데이터베이스에 접속하여 SQL 문을 통해 다양한 작업을 수행한다. CSQL 인터프리터를 이용해 다음과 같은 작업을 수행할 수 있다.

- SQL 문을 이용하여 데이터베이스 조회, 갱신, 삭제 등의 작업
- 외부 셸 명령 실행
- 조회 결과의 저장 혹은 출력
- SQL 스크립트 파일의 작성 및 실행
- 테이블 스키마 조회
- 데이터베이스 서버 시스템 파라미터의 조회 및 변경
- 다양한 데이터베이스 정보(스키마, 트리거, 지연 트리거, workspace, 잠금, 통계) 조회

DBA를 위한 도구

DBA(Database Administrator)는 일상적인 많은 관리 업무를 수행하기 위해서 CUBRID가 설치된 시스템에 접속해서 CUBRID가 제공하는 다양한 관리 유틸리티를 이용해서 작업을 수행한다. 따라서, 터미널 기반의 인터페이스를 제공하는 CSQL 인터프리터는 **DBA**가 데이터베이스 관리 업무를 수행하는데 유용하게 사용된다. 또한, CSQL 인터프리터는 **DBA**에게 필요한 다양한 데이터베이스 정보를 제공한다.

CSQL 인터프리터는 독립 모드(Standalone Mode)로 실행될 수도 있다. 독립 실행 모드는 CSQL 인터프리터가 서버 프로세스의 기능을 포함하여 직접 데이터베이스 파일에 접근하여 수행하는 방식이다. 즉 별도의 데이터베이스 서버 프로세스가 구동되어 있지 않은 상태에서 해당 데이터베이스를 대상으로 SQL 문을 실행할 수 있다. CSQL 인터프리터는 데이터베이스 서버나 브로커 등 어떠한 다른 프로그램의 도움 없이 **csql** 유틸리티 하나로 데이터베이스를 이용할 수 있는 강력한 수단이다.

CSQL 실행

CSQL 실행 모드

대화형 모드(Interactive Mode)

CSQL 인터프리터는 데이터베이스에서 스키마 또는 데이터를 다루기 위한 SQL 문을 입력하고 수행할 수 있다. **csql** 유틸리티를 실행하면 나타나는 프롬프트에 사용자는 구문을 입력한다. 구문을 입력한 후 실행하면 다음 라인에 결과가 표시되는데, 이를 대화형 모드라고 한다.

일괄 수행 모드(Batch Mode)

사용자는 원하는 SQL 문을 임의의 파일에 저장한 후 **csql** 유틸리티가 해당 파일을 읽도록 구문을 실행할 수 있다. 이를 일괄 수행(배치형) 모드라고 한다. 일괄 수행 모드에 대한 자세한 내용은 [CSQL 시작 옵션](#)을 참조한다.

독립 모드 (Standalone Mode)

독립 실행 모드는 CSQL 인터프리터가 서버 프로세스의 기능을 포함하여 직접 데이터베이스 파일에 접근하여 수행하는 방식이다. 즉 별도의 데이터베이스 서버 프로세스가 구동되어 있지 않은 상태에서 해당 데이터베이스를 대상으로 SQL 문을 실행할 수 있다. 독립 모드는 동시에 한 사용자만이 접근이 가능하므로 **DBA(Database Administrator)**가 관리 작업을 위해 수행하는데 적합한 모드이다.

클라이언트/서버 모드 (Client/Server Mode)

클라이언트/서버 모드는 일반적으로 해당 CSQL 인터프리터가 클라이언트 프로세스로 동작하여 데이터베이스 서버 프로세스에 접속하는 방식으로 사용되는 모드이다.

CSQL 사용 방법

로컬 호스트 접속

설명

csql 유틸리티를 사용하여 CSQL 인터프리터를 실행한다. 이 때, 필요에 따라 옵션을 설정할 수 있으며, 옵션을 설정하려면 접속하려는 데이터베이스 이름을 인수로 지정한다. 다음은 로컬 서버에 위치한 데이터베이스에 접속하는 **csql** 유틸리티 구문이다.

구문

```
csql [ options ] database_name
```

원격 호스트 접속

설명

다음은 원격 호스트에 위치한 데이터베이스에 접속하는 **csql** 유틸리티 구문이다.

구문

```
csql [ options ] database_name@remote_host_name
```

단, 원격 호스트에서 CSQL 인터프리터를 실행하려면 다음 조건을 만족해야 한다.

- 원격 호스트와 로컬 호스트에 설치된 CUBRID는 동일한 버전이어야 한다.
- 원격 호스트와 로컬 호스트의 마스터 프로세스가 사용하는 포트 번호가 동일해야 한다.
- **-C** 옵션을 사용하여 클라이언트/서버 모드로 원격 호스트에 접속해야 한다.

예제

다음은 192.168.1.3 위치의 원격 호스트에 존재하는 **demodb**에 접속하여 **csql** 유틸리티를 호출하는 예제이다.

```
csql -C demodb@192.168.1.3
```

CSQL 시작 옵션

프롬프트 상에서 옵션 목록을 보려면, 다음과 같이 옵션을 적용할 데이터베이스를 지정하지 않고 **csql** 유틸리티를 실행한다.

```
% csql
interactive SQL utility, version R3.0
usage: csql [OPTION] database-name valid options:
  -S, --SA-mode           standalone mode execution
  -C, --CS-mode           client-server mode execution
  -u, --user=ARG          alternate user name
  -p, --password=ARG      password string, give "" for none
  -e, --error-continue    don't exit on statement error
  -i, --input-file=ARG    input-file-name
  -o, --output-file=ARG   output-file-name
  -s, --single-line       single line oriented execution
  -c, --command=ARG       CSQL-commands
  -l, --line-output       display each value in a line
  -r, --read-only         read-only mode
  --no-auto-commit        disable auto commit mode execution
  --no-pager              do not use pager
```

옵션

다음은 **csql** 유틸리티와 함께 사용할 수 있는 옵션을 정리한 표이다.

옵션	설명
-S	독립 모드(standalone mode)로 실행하기 위한 옵션이다.
-C	클라이언트/서버 모드(client/server mode)로 실행하기 위한 옵션이다.

-u <i>user_name</i>	데이터베이스에 접속하려는 사용자를 명시할 때 사용하는 옵션으로, 디폴트 값은 PUBLIC 이다.
-p <i>password</i>	데이터베이스에 접속하려는 사용자의 암호가 존재하는 경우 암호를 입력하기 위한 옵션이다.
-e	에러가 발생하더라도 세션을 종료하지 않고 계속 실행하라는 옵션이다.
-i <i>input_file</i>	인수로 지정된 <i>input_file</i> 은 SQL 문이 저장된 파일이며, 배치 모드 실행을 위한 옵션이다.
-o <i>output_file</i>	구문 실행 결과를 화면에 표시하지 않고 지정된 <i>output_file</i> 에 저장한다.
-s	복수 개의 SQL 문이 연속적으로 저장된 파일에서 SQL 문을 하나씩 실행하려고 할 때 사용하는 옵션이다. 각 SQL 문은 세미콜론(;)으로 구분한다.
-c "CSQL commands"	프롬프트 상에서 직접 SQL 문을 수행하기 위한 옵션으로 큰 따옴표 안에 수행하려는 SQL 문을 입력한다.
-l	실행된 SQL 문에 대한 결과를 컬럼이 아닌 라인 형태로 출력하는 옵션이다. 디폴트 값은 컬럼 형태로 출력된다.
-r	데이터베이스에 읽기 전용으로 접속하는 옵션이다.
--no-auto-commit	CSQL 인터프리터의 자동 커밋(autocommit) 모드를 OFF 로 설정하는 옵션이다.
--no-pager	CSQL 인터프리터에서 수행한 질의 결과를 페이지 단위로 출력하지 않고 일괄적으로 출력하는 옵션이다.

독립 모드에서 실행(-S)

-S 옵션을 이용하여 독립 모드로 **demodb**에 접속하여 **csql**을 실행하는 구문이다. **demodb**를 독점적으로 사용하고자 할 때 **-S** 옵션을 이용한다.

```
csql -S demodb
```

클라이언트/서버 모드에서 실행(-C)

-C 옵션을 이용하여 클라이언트/서버 모드로 **demodb**에 접속하여 **csql** 유틸리티를 실행하는 구문이다. **demodb**에 여러 클라이언트가 동시 접속하는 환경에서 **-C** 옵션을 이용한다. 만약 클라이언트/서버 모드로 원격 호스트의 데이터베이스에 접속한 경우라도 **csql** 유틸리티를 실행하는 도중에 발생한 에러 로그는 로컬 호스트의 **cub_client.err** 파일에 기록된다.

```
csql -C demodb
```

배치 모드에서 사용되는 입력 파일을 지정(-i)

-i 옵션을 이용하여 배치 모드에서 사용할 입력 파일의 이름을 지정하는 구문이다. **infile 파일에는** 하나 이상의 SQL 문이 저장되어 있으며, -i 옵션이 지정되지 않으면 CSQL 인터프리터는 대화형 모드로 실행된다.

```
csql -i infile demodb
```

수행 결과를 저장할 출력 파일을 지정(-o)

-o 옵션을 이용하여 질의 수행 결과를 화면에 출력하지 않고 지정된 파일에 저장하도록 하는 구문이다. 이는 CSQL 인터프리터에 의한 질의 수행 결과를 추후 조회하고자 할 때 유용하게 사용될 수 있다.

```
csql -o outfile demodb
```

사용자 이름을 지정(-u)

-u 옵션을 이용하여 지정된 데이터베이스에 접속하려는 사용자 이름을 지정하는 구문이다. 만약 -u 옵션이 지정되지 않으면 가장 낮은 사용자 권한을 가지는 **PUBLIC**이 사용자로 지정된다. 또한 사용자 이름이 유효하지 않은 경우에는 오류가 출력되고 **csql** 유틸리티는 종료된다. 암호가 설정된 사용자 이름이 지정된 경우에는 암호를 입력받기 위한 프롬프트가 출력된다.

```
csql -u DBA demodb
```

사용자 암호를 지정(-p)

-p 옵션을 이용하여 지정된 사용자의 암호를 입력하는 구문이다. 특히, 배치 모드에서는 지정한 사용자에 대한 암호 입력을 요청하는 프롬프트가 출력되지 않으므로 -p 옵션을 이용하여 암호를 입력해야 한다. 잘못된 암호를 입력하면, 오류가 출력되고 **csql** 유틸리티는 종료된다.

```
csql -u DBA -p *** demodb
```

SQL 문을 하나씩 수행하도록 지정(-s)

-s 옵션을 이용하여 입력된 SQL 문을 하나씩 수행하는 구문이다. 이 옵션은 질의 수행을 위해 메모리를 적게 할당하고 싶을 때 유용하게 이용할 수 있다. 이 때, 각 SQL 문은 세미콜론(;)으로 구분한다.

```
csql -s -i infile demodb
```

셸에서 직접 SQL 문을 처리(-c)

-c 옵션을 이용하여 셸 상에서 하나 이상의 SQL 문을 직접 수행하는 구문이다. 이 때, 각 문장은 세미콜론(;)으로 구분한다.

```
csql -c "select * from olympic;select * from stadium" demodb
```

라인 단위로 출력(-l)

-l 옵션을 이용하여 SQL 문을 실행한 결과를 라인 단위로 출력하는 구문이다. -l 옵션이 지정되지 않으면 열 단위로 출력된다.

```
csql -l demodb
```

에러를 무시하고 계속 실행(-e)

-e 옵션을 이용하여 의미상 오류 또는 런타임 에러가 발생하여도 이를 무시하고 계속 SQL 문을 실행하라는 구문이다. 이 때, SQL 문에 오류가 있는 경우에는 **-e** 옵션이 지정되어 있어도 데이터베이스는 종료된다.

```
csql -e demodb
```

읽기 전용으로 접속 (-r)

-r 옵션을 이용하여 읽기 전용으로 데이터베이스에 접속하는 구문이다. 데이터베이스에 읽기 전용으로 접속하면 테이블을 만들거나 데이터를 입력할 수 없고 데이터를 조회만 할 수 있다.

```
csql -r demodb
```

자동 커밋 모드 중지(--no-auto-commit)

--no-auto-commit 옵션을 이용하여 자동 커밋 모드를 중지하는 구문이다. **--no-auto-commit** 옵션을 지정하지 않으면 기본적으로 CSQL 인터프리터는 자동 커밋 모드로 작동되고, 입력된 SQL 문이 실행될 때마다 자동으로 커밋된다. 또한, CSQL 인터프리터를 시작한 후 ;**Autocommit** 세션 명령을 수행해도 동일한 결과를 얻을 수 있다.

```
csql --no-auto-commit demodb
```

질의 수행 결과를 일괄적으로 보여주기(--no-pager)

--no-pager 옵션을 이용하여 CSQL 인터프리터에서 수행한 질의 결과를 페이지 단위로 출력하지 않고, 일괄적으로 출력하라는 구문이다. **--no-pager** 옵션을 지정하지 않으면 페이지 단위로 질의 수행 결과가 출력된다.

```
csql --no-pager demodb
```

세션 명령어

CSQL 인터프리터에는 SQL 문 이외에 CSQL 인터프리터를 제어하는 특별한 명령어가 있으며 이를 세션 명령어라고 한다. 모든 세션 명령어는 반드시 세미콜론(;)으로 시작해야 한다.

세션 명령어

;help를 입력하여 CSQL 인터프리터에서 지원되는 세션 명령어를 확인할 수 있다. 세션 명령어를 전부 입력하지 않고 대문자로 표시된 글자까지만 입력해도 CSQL 인터프리터는 세션 명령어를 인식한다. 세션 명령어는 대소문자를 구분하지 않는다.

```
CUBRID SQL Interpreter
Type `;help' for help messages.
csql> ;help
=== <Help: Session Command Summary> ===
All session commands should be prefixed by `;' and only blanks/tabs
can precede the prefix. Capitalized characters represent the minimum
abbreviation that you need to enter to execute the specified command.
;REAd    [<file-name>]    - read a file into command buffer.
;WRite   [<file-name>]    - (over)write command buffer into a file.
;APPend  [<file-name>]    - append command buffer into a file.
;PRINT   - print command buffer.
;SHELL   - invoke shell.
;CD      - change current working directory.
;EXit    - exit program.
;CLear   - clear command buffer.
;EDIT    - invoke system editor with command buffer.
;List    - display the content of command buffer.
;RUN     - execute sql in command buffer.
;Xrun    - execute sql in command buffer, and clear the command
buffer.
;COMmit  - commit the current transaction.
;ROLLback - roll back the current transaction.
;AUTocommit [ON|OFF]    - enable/disable auto commit mode.
;CHECKpoint - issue checkpoint.
;Killtran - kill transaction.
;REStart   - restart database.
;SHELL_Cmd [shell-cmd]  - set default shell, editor, print and pager
;EDITOR_Cmd [editor-cmd] - command to new one, or display the current
;PRINT_Cmd [print-cmd]  - one, respectively.
;PAger cmd [pager-cmd]
;DATE      - display the local time, date.
;DATAbase  - display the name of database being accessed.
;SCHEMA class-name - display schema information of a class.
;SYntax [sql-cmd-name] - display syntax of a command.
;TRigger ['*'|trigger-name] - display trigger definition.
;Get system_parameter - get the value of a system parameter.
;SEt system_parameter=value - set the value of a system parameter.
;PLan [simple/detail/off] - show query execution plan.
;Info <command> - display internal information.
;Time [ON|OFF] - enable/disable to display the query execution time.
;HISTORYList - display list of the executed queries.
;HISTORYRead <history num> - read entry on the history number into command buffer.
;HELP       - display this help message.
csql>
```

옵션

파일에서 SQL 구문 읽기(;REAd)

;READ 명령어는 파일의 내용을 명령어 버퍼로 읽는 세션 명령어로 지정된 입력 파일에 저장된 SQL

명령어들을 실행하는데 사용할 수 있다. 버퍼에 올려진 파일 내용을 보기 위해서는 **;List** 명령어를 사용한다.

```
csql> ;rea nation.sql
The file has been read into the command buffer.
csql> ;list
insert into "sport_event" ("event_code", "event_name", "gender_type", "num_player") values
(20001, 'Archery Individual', 'M', 1);
insert into "sport_event" ("event_code", "event name", "gender type", "num player") values
20002, 'Archery Individual', 'W', 1);
....
```

파일에 SQL 구문 저장(Write)

;Write는 명령어 버퍼의 내용을 파일에 저장하는 세션 명령어로 사용자가 CSQL 인터프리터에서 입력 혹은 수정한 SQL 명령어를 파일에 저장할 때 사용된다.

```
csql> ;w outfile
Command buffer has been saved.
```

파일에 덧붙이기(Append)

현재 명령어 버퍼의 내용을 출력 파일인 **outfile**에 추가한다.

```
csql> ;ap outfile
Command buffer has been saved.
```

셸 명령어를 실행(SHELL)

;SHELL 세션 명령어로 외부 셸을 호출할 수 있다. CSQL 인터프리터가 실행된 환경에서 새로운 셸이 시작되고, 셸을 마치면 다시 CSQL 인터프리터로 돌아온다. 만약에 **;SHELL_Cmd** 명령어로 수행할 셸 명령어가 지정되어 있다면 셸을 구동하여 지정된 명령어를 실행하고 CSQL 인터프리터로 복귀하게 된다.

```
csql> ;shell
% ls -al
total 2088
drwxr-xr-x 16 DBA cubrid 4096 Jul 29 16:51 .
drwxr-xr-x 6 DBA cubrid 4096 Jul 29 16:17 ..
drwxr-xr-x 2 DBA cubrid 4096 Jul 29 02:49 audit
drwxr-xr-x 2 DBA cubrid 4096 Jul 29 16:17 bin
drwxr-xr-x 2 DBA cubrid 4096 Jul 29 16:17 conf
drwxr-xr-x 4 DBA cubrid 4096 Jul 29 16:14 cubridmanager
% exit
csql>
```

셸 명령어 등록(SHELL_Cmd)

;SHELL_Cmd를 사용하여 **;SHELL** 세션 명령어로 실행할 셸 명령어를 등록한다. 등록된 명령어를 실행하기 위해서는 예제와 같이 **;shell** 명령어를 입력한다.

```
csql> ;shell c ls -la
csql> ;shell
total 2088
drwxr-xr-x 16 DBA cubrid 4096 Jul 29 16:51 .
drwxr-xr-x 6 DBA cubrid 4096 Jul 29 16:17 ..
drwxr-xr-x 2 DBA cubrid 4096 Jul 29 02:49 audit
drwxr-xr-x 2 DBA cubrid 4096 Jul 29 16:17 bin
drwxr-xr-x 2 DBA cubrid 4096 Jul 29 16:17 conf
drwxr-xr-x 4 DBA cubrid 4096 Jul 29 16:14 cubridmanager
csql>
```

현재 작업 디렉터리 변경(;CD)

CSQL 인터프리터를 실행한 현재 작업 디렉터리를 지정된 디렉터리로 변경한다. 경로를 지정하지 않으면 홈 디렉터리로 변경된다.

```
csql> ;cd /home1/DBA/CUBRID
Current directory changed to /home1/DBA/CUBRID.
```

CSQL 인터프리터 종료(;EXIT)

CSQL 인터프리터를 종료한다.

```
csql> ;ex
```

명령어 버퍼 초기화(;CLEAR)

;CLEAR 세션 명령어는 명령어 버퍼의 내용을 초기화한다.

```
csql> ;cl
csql> ;list
```

명령어 버퍼의 내용 보여주기(;LIST)

현재까지 입력 수정된 명령어 버퍼의 내용을 화면에 출력하기 위해서는 **;LIST** 세션 명령어를 사용한다. 명령어 버퍼는 사용자의 SQL 입력, **;READ** 명령어, **;EDIT** 명령어 등으로 수정될 수 있다.

```
csql> ;l
```

SQL 문 실행(;RUN)

명령어 버퍼에 있는 SQL 문을 실행하는 명령어이다. 다음에서 설명하는 **;Xrun** 세션 명령어와 달리 질의 실행 후에도 버퍼는 초기화되지 않는다.

```
csql> ;ru
```

SQL 문 실행 후 명령어 버퍼 초기화(;Xrun)

명령어 버퍼에 있는 SQL 문을 실행하는 명령어이다. 질의 실행 후 명령어의 버퍼는 초기화된다.

```
csql> ;x
```

트랜잭션 커밋(;COMMIT)

현재 수행되고 있는 트랜잭션을 커밋(commit)하는 세션 명령어이다. 자동 커밋(auto-commit) 모드가 아닌 경우, 명시적으로 커밋 명령어를 입력해야 CSQL 인터프리터에서 수행 중이던 트랜잭션이 커밋된다. 자동 커밋(auto-commit) 모드인 경우는 **;RUN**과 **;Xrun** 세션 명령어를 실행할 때마다 트랜잭션이 자동으로 커밋된다.

```
csql> ;co
Current transaction has been committed.
```

트랜잭션 롤백(;ROLLBACK)

현재 수행되고 있는 트랜잭션을 롤백(rollback)하는 세션 명령어이다. **;COMMIT**과 마찬가지로 자동 커밋(auto-commit) 모드가 아닐 경우(OFF)에만 의미가 있다.

```
csql> ;ro
Current transaction has been rolled back.
```

자동 커밋 모드 설정(Autocommit)

자동 커밋(auto-commit) 모드를 **ON** 또는 **OFF**로 설정하는 명령어이다. 만약, **ON** 또는 **OFF**를 지정하지 않으면 현재 설정된 값을 보여준다. 참고로 CSQL 인터프리터는 기본 값이 **ON**이다.

```
csql> ;au off
AUTOCOMMIT IS OFF
```

체크포인트 수행(Checkpoint)

CSQL 세션 내에서 체크포인트 수행을 지시하는 명령어이다. CSQL 인터프리터 접속 시 사용자 지정 옵션(-u *user_name*)에 **DBA** 그룹 멤버가 지정되고 시스템 관리자 모드(--sysadm)로 접속한 경우에만 수행할 수 있다.

체크포인트는 현재 데이터 버퍼에 존재하는 로그 파일(더티 페이지)을 디스크로 내려쓰기(flush)하는 작업이며, CSQL 세션 내에서 파라미터 값을 설정하는 명령어(**set parameter_name value**)를 통해서도 체크포인트 주기를 변경할 수 있다. 체크포인트 수행 주기와 관련된 파라미터는

checkpoint_interval_in_mins와 **checkpoint_every_npages**가 있다. 이에 대한 자세한 내용은 [로그 관련 파라미터](#)를 참고한다.

```
csql> ;ch
Checkpoint has been issued.
```

트랜잭션 모니터링 또는 종료(Killtran)

CSQL 세션 내에서 트랜잭션 상태 정보를 확인하거나 특정 트랜잭션을 종료시키는 명령어이다. CSQL 인터프리터 접속 시 사용자 지정 옵션(-u *user_name*)에 **DBA** 그룹 멤버가 지정되고 시스템 관리자 모드(--sysadm)로 접속한 경우에만 수행할 수 있다.

인자가 생략되면 모든 트랜잭션 상태 정보를 화면 출력하고, 인자로 특정 트랜잭션 ID가 지정되면 해당 트랜잭션을 종료시킨다.

```
csql> ;k
Tran index      User name      Host name      Process id      Program name
-----
1 (+)           dba            myhost         664             cub_cas
2 (+)           dba            myhost         6700            csq1
3 (+)           dba            myhost         2188            cub_cas
4 (+)           dba            myhost         696             csq1
5 (+)           public         myhost         6944            csq1

csql> ;k 3
The specified transaction has been killed.
```

데이터베이스 재접속(REStart)

CSQL 세션 내에서 대상 데이터베이스에 재접속을 시도하는 명령어이다. CSQL 인터프리터를 클라이언트/서버 모드(CS 모드)로 수행하는 경우에는 서버와의 접속이 해제되므로 유의한다. 이 명령어는 HA 환경에서 장애로 인해 다른 서버로 절체가 이루어짐에 따라 도중에 서버와의 연결이 해제되는 경우, 세션을 유지하면서 절체된 서버로 재접속할 때 유용하게 사용할 수 있다.

```
csql> ;res
The database has been restarted.
```

현재 날짜 출력(:DATE)

;DATE는 CSQL 인터프리터에서 현재 날짜 및 시간 정보를 출력한다.

```
csql> ;date
Tue July 29 18:58:12 KST 2008
```

대상 데이터베이스 정보 출력(:DATAbase)

CSQL 인터프리터에서 작업 중인 데이터베이스 이름 및 호스트 이름을 출력한다. 만약, 대상 데이터베이스가 HA모드로 동작 중이라면 현재 HA모드(active, standby, 또는 maintenance)도 함께 출력될 것이다.

```
csql> ;data
demodb@localhost (active)
```

지정한 테이블의 스키마 정보 출력(:SCHEMA)

;Schema 세션 명령어로 지정한 테이블의 스키마 정보를 확인할 수 있다. 해당 테이블의 이름, 컬럼명, 제약 사항 등의 정보가 출력된다.

```
csql> ;sc event
=== <Help: Schema of a Class> ===
<Class Name>
    event
<Attributes>
    code          INTEGER NOT NULL
    sports        CHARACTER VARYING(50)
    name          CHARACTER VARYING(50)
    gender        CHARACTER(1)
    players       INTEGER
<Constraints>
    PRIMARY KEY pk_event_event_code ON event (code)
```

구문 규칙 출력(:SYntax)

지정한 SQL 구문의 규칙을 출력하는 명령어이다. 특정한 구문을 지정하지 않으면 정의된 모든 구문과 해당 구문의 규칙을 보여준다.

```
csql> ;sy alter
=== <Help: Command Syntax> ===
<Name>
    ALTER
<Description>
    Change the definition of a class or virtual class.
<Syntax>
<alter> ::= ALTER [ <class type> ] <class name> <alter clause> ;
<class_type> ::= CLASS | TABLE | VCLASS | VIEW
<alter_clause> ::= ADD <alter add> [ INHERIT <resolution comma list> ] |
                  DROP <alter drop> [ INHERIT <resolution comma list> ] |
                  RENAME <alter rename> [ INHERIT <resolution comma list> ] |
                  CHANGE <alter change> |
                  INHERIT <resolution comma list>
<alter_add> ::= [ ATTRIBUTE | COLUMN ] <class element comma list> |
               CLASS ATTRIBUTE <attribute definition comma list> |
               FILE <file_name_comma_list> |
               METHOD <method_definition_comma_list> |
               QUERY <select statement> |
               SUPERCLASS <class_name_comma_list>
.....
```

트리거 출력(:TRiger)

지정한 트리거 이름을 검색하여 출력하는 명령어이다. 트리거 이름을 지정하지 않으면 정의된 모든 트리거를 보여준다.

```
csql> ;tr
=== <Help: All Triggers> ===
glo_delete_contents
```

파라미터 값 확인(:Get)

;Get 세션 명령어를 이용해 현재 CSQL 인터프리터에 설정된 파라미터 값을 확인할 수 있다. 지정된 파라미터 명이 정확하지 않으면 오류가 발생한다.

```
csql> ;g isolation_level
=== Get Param Input ===
isolation_level=4
```

파라미터 값 설정(:Set)

특정 파라미터의 값을 설정하기 위해서는 **;Set** 세션 명령어를 사용한다. 클라이언트 파라미터에 대해서만 값을 변경할 수 있으며, 서버 파라미터는 변경할 수 없다는 점에 주의한다.

```
csql> ;se block ddl statement=1
=== Set Param Input ===
block_ddl_statement=1
```

질의 실행 계획 보기 수준 설정(:Plan)

;Plan 세션 명령어는 질의 실행 계획 보기의 수준을 설정한다. 수준은 **simple**, **detail**, **off**로 지정한다. 각 설정값의 의미는 다음과 같다.

- **off** : 질의 실행 계획을 출력하지 않음
- **simple** : 질의 실행 계획을 단순하게 출력함. (OPT LEVEL=257)
- **detail** : 질의 실행 계획을 자세하게 출력함. (OPT LEVEL=513)

정보 출력(:Info)

;Info 세션 명령어는 스키마, 트리거, 작업 환경, 잠금, 통계 등의 정보를 확인할 수 있는 명령어이다.

```
csql> ;i lock
*** Lock Table Dump ***
Lock Escalation at = 100000, Run Deadlock interval = 1
Transaction (index 0, unknown, unknown@unknown|-1)
Isolation REPEATABLE CLASSES AND READ UNCOMMITTED INSTANCES
State TRAN ACTIVE
Timeout_period -1
.....
```

서버 실행 통계 정보 출력(:Hist)

;Hist 세션 명령어는 데이터베이스 서버 실행 통계 정보를 확인하기 위한 세션 명령어로서, 이 명령어가 입력된 이후부터 서버 실행 통계 정보를 추출한다. 따라서, 서버 실행 통계 정보를 화면에 출력하기 위해서는 **;dump_hist** 또는 **;x** 와 같은 실행 명령어를 입력해야 한다.

이는 **cubrid.conf** 파일에서 관련 파라미터(**communication_histogram**)를 **yes**로 설정한 경우에만 동작되며, **cubrid statdump** 유틸리티를 이용해서도 서버 실행 통계 정보를 확인할 수 있다. **;.hist** 세션 명령어의 옵션으로 **on**, **off**, **all**을 제공하며, 각 옵션의 의미는 다음과 같다.

- **on** : 해당 트랜잭션에 관한 실행 통계 정보를 출력.
- **all** : 서버 전체에 대한 실행 통계 정보를 출력.
- **off** : 서버 실행 통계 정보를 수집하지 않음.

다음 예제는 서버 전체에 대한 실행 통계 정보를 확인하는 예제이며, 출력되는 통계 정보 항목에 관한 설명은 [데이터베이스 서버 실행 통계 정보 출력](#)을 참고한다.

```
csql> ;.hist all
...
--Executing queries
...
csql> ;.x

*** SERVER EXECUTION GLOBAL STATISTICS ***
Num data page fetches   = 938215
Num data page dirties   = 851792
Num data page ioreads   = 15050
Num data page iowrites  = 0
Num iosynches           = 15187
Num log page ioreads    = 0
Num log page iowrites   = 15187
Num log append records  = 625838
Num log archives        = 0
Num log checkpoints     = 0
Num tran commits        = 40080
Num tran rollbacks      = 40
Hit ratio of Page Buffer = 98.40%

csql> ;.h off
```

질의 수행 시간을 출력(Time)

;Time 세션 명령어로 질의를 수행한 시간을 출력하도록 설정할 수 있다. **ON** 혹은 **OFF**로 지정하며 인자가 없을 경우에는 현재 설정값을 보여준다.

```
csql> ;ti ON
csql> ;ti
TIME IS ON
```

질의 수행 이력 확인(HISTORYList)

이전에 수행된 명령어(입력 내용)을 수행 번호를 포함한 리스트로 보여준다.

```
csql> ;history1
----< 1 >----
select * from nation;
----< 2 >----
select * from athlete;
```

지정된 수행 번호에 해당하는 입력 내용을 버퍼로 불러오기(HISTORYRead)

;HISTORYRead 세션 명령어를 사용해 지정된 **;HISTORYList**에서 확인한 수행 번호에 해당하는 내용을 명령어 버퍼로 불러올 수 있다. 해당 SQL 문을 직접 입력한 것과 같은 상태이므로 바로 **;ru** 나 **;x**를 입력할 수 있다.

```
csql> ;historyr 1
```

기본 편집기를 호출(EDIT)

지정된 편집기를 호출하는 세션 명령어이다. 기본 편집기는 Linux에서는 vi이고, Windows에서는 메모장이다. 다른 편집기로 지정하려면 ;**EDITOR_Cmd** 명령어를 이용한다.

```
csql> ;edit
```

편집기 설정(EDITOR_Cmd)

;**EDIT** 세션 명령어에서 사용될 편집기를 지정한다. 예제와 같이 기본 편집기인 vi 대신에 해당 시스템에 설치된 다른 편집기(예: emacs)를 설정할 수 있다.

```
csql> ;editor c emacs  
csql> ;edit
```


CUBRID SQL 설명서

이 장에서는 CUBRID에서 사용되는 데이터 타입, 함수와 연산자, 데이터 조회나 테이블 조작 등의 SQL 구문에 대해 설명한다. 인덱스나 트리거, 분할, 시리얼 및 사용자 정보 변경 등의 작업을 위한 SQL 구문도 찾아볼 수 있다.

주요 내용은 다음과 같다.

- 데이터 타입
- 연산자와 함수
- 데이터 조회
- 질의 최적화
- 테이블 조작
- 데이터 조작
- 뷰(VIEW)
- 인덱스(INDEX)
- 트랜잭션 관리
- 트리거(TRIGGER)
- 메소드
- 분할
- 시리얼
- 접근 권한 부여 구문
- Java 저장 함수/프로시저
- CUBRID SQL 구문

용어 정리

CUBRID는 상속의 개념이 있는 객체 관계형 데이터베이스 시스템이다. 본 매뉴얼에서는 내용의 이해를 돕고자 관계형 데이터베이스에서 쓰는 용어와 혼용하여 사용하고 있는데, 특히 상속의 개념을 포함하여 설명하는 경우는 클래스, 인스턴스, 속성 등과 같이 객체 기반 용어를 주로 사용하였고 일반 SQL 설명에서는 주로 관계형 데이터베이스 용어를 사용하였다.

이에 대한 정리를 하면 다음과 같다.

일반 관계형 데이터베이스	CUBRID
테이블	클래스, 테이블
컬럼	속성(attribute), 컬럼
레코드	인스턴스(instance), 레코드
데이터 타입	도메인, 데이터 타입

주석

CSQL 인터프리터에서 입력 가능한 주석은 '--'로 시작하여 해당 라인 전부가 적용되는 SQL 방식이다.

추가로 '/'로 시작하는 C++ 언어 방식과 '/*'로 시작하여 '*/'로 끝나는 C 언어 방식도 지원한다.

다음은 CSQL 인터프리터에서 입력 가능한 주석의 예이다.

예제

- -- 사용

```
-- 이것은 SQL 방식의 주석이다.
```

- // 사용

```
// 이것은 C++ 방식의 주석이다.
```

- /* */ 사용

```
/* 이것은 C 방식의 주석이다. */
```

```
/* 이것은 C 방식을 이용하여
```

```
두개의 라인이 주석으로 적용되었다 */
```

식별자

식별자 작성 원칙

CSQL 인터프리터에서 입력 가능한 식별자는 다음의 원칙에 따라 작성한다.

- 반드시 문자로 시작되어야 한다. 즉, 숫자나 기호로 시작할 수 없다.
- 대소문자를 구별하지 않는다.
- 예약어는 허용되지 않는다.

<identifier>

```
:: = <identifier_letter> [ { <other_identifier> }; ]
```

<identifier_letter>

```
:: = <upper_case_letter>  
| <lower_case_letter>
```

<other_identifier>

```
:: = <identifier_letter>  
| <digit>  
| _  
| #
```

<digit>

```
:: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<upper_case_letter>

```
:: = A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V  
| W | X | Y | Z
```

<lower_case_letter>

```
:: = a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v  
| w | x | y | z
```

허용되는 식별자

문자로 시작되는 식별자

식별자의 첫 글자에는 반드시 문자를 사용해야 하며, 연산자로 사용되는 특수 문자를 제외한 나머지 특수 문자를 포함할 수 있다. 다음은 허용되는 식별자의 예제이다.

```
a  
a b  
ssn#  
this_is_an_example_#
```

큰따옴표, 대괄호, 백틱 부호로 둘러싸인 식별자

허용되지 않는 식별자 또는 예약어에 해당하더라도, 큰따옴표(" ")나 대괄호([]) 또는 백틱 부호(`)로 식별자를 둘러싸면 예외적으로 허용된다. 큰따옴표는 쿼리 관련 파라미터인 **ansi_quote**가 yes일 때에만 식별자를 감싸는 부호로 사용할 수 있고, 이 값이 no이면 문자열을 감싸는 부호로 사용한다. 다음은 허용되는 식별자의 예제이다.

```
"select"  
"@lowcost"  
"low cost"  
"abc"def"  
[position]
```

허용되지 않는 식별자

특수 문자나 숫자로 시작되는 식별자

다음과 같이 언더바(_), #을 제외한 특수 문자, 숫자로 시작되는 식별자는 허용되지 않는다.

```
a  
#ack  
%nums  
2fer  
88abs
```

공백을 포함하는 식별자

다음과 같이 중간에 공백을 포함하는 식별자는 허용되지 않는다.

```
coll t1
```

연산자로 사용하는 특수 문자를 포함하는 식별자

다음과 같이 연산자로 사용되는 특수 문자(+, -, *, /, %, ||, !, <, >, =, |, ^, &, ~)를 포함하는 식별자는 허용되지 않는다.

```
col+  
col~  
col&&
```

예약어

아래는 CUBRID 키워드(keywords) 중 명령어, 함수명, 타입명 등으로 예약되어 있는 예약어(reserved words)를 정리한 표이다. 사용자는 테이블 이름, 컬럼 이름, 변수 이름과 같은 식별자(identifier)로 아래에 정리된 예약어를 사용할 수 없다. 단, 큰따옴표(" ")나 대괄호([]) 또는 백틱 부호(`)로 둘러싸는 방법으로 예약어를 식별자로 사용할 수 있다.

ABSOLUTE	ACTION	ADD
ADD_MONTHS	AFTER	ALIAS
ALL	ALLOCATE	ALTER
AND	ANY	ARE
AS	ASC	ASSERTION
ASYNC	AT	ATTACH
ATTRIBUTE	AVG	BEFORE
BETWEEN	BIGINT	BIT
BIT_AND	BIT_LENGTH	BOOLEAN
BOTH	BREADTH	BY
CALL	CASCADE	CASCADEED
CASE	CAST	CATALOG
CHANGE	CHAR	CHARACTER
CHECK	CLASS	CLASSES
CLOSE	CLUSTER	COALESCE
COLLATE	COLLATION	COLUMN
COMMIT	COMPLETION	CONNECT
CONNECT_BY_ISCYCLE	CONNECT_BY_ISLEAF	CONNECT_BY_ROOT
CONNECTION	CONSTRAINT	CONSTRAINTS
CONTINUE	CONVERT	CORRESPONDING
COUNT	CREATE	CROSS
CURRENT	CURRENT_DATE	CURRENT_DATETIME
CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER
CURSOR	CYCLE	DATA

DATA_TYPE	DATABASE	DATE
DATETIME	DAY	DAY_HOUR
DAY_MILLISECOND	DAY_MINUTE	DAY_SECOND
DEALLOCATE	DEC	DECIMAL
DECLARE	DEFAULT	DEFERRABLE
DEFERRED	DELETE	DEPTH
DESC	DESCRIBE	DESCRIPTOR
DIAGNOSTICS	DICTIONARY	DIFFERENCE
DISCONNECT	DISTINCT	DISTINCTROW
DIV	DOMAIN	DOUBLE
DROP	EACH	ELSE
ELSEIF	END	EQUALS
ESCAPE	EVALUATE	EXCEPT
EXCEPTION	EXCLUDE	EXEC
EXECUTE	EXISTS	EXTERNAL
EXTRACT	FALSE	FETCH
FILE	FIRST	FLOAT
FOR	FOREIGN	FOUND
FROM	FULL	FUNCTION
GENERAL	GET	GLOBAL
GO	GOTO	GRANT
GROUP	HAVING	HOUR
HOUR_MILLISECOND	HOUR_MINUTE	HOUR_SECOND
IDENTITY	IF	IGNORE
IMMEDIATE	IN	INDEX
INDICATOR	INHERIT	INITIALLY
INNER	INOUT	INPUT
INSERT	INT	INTEGER
INTERSECT	INTERSECTION	INTERVAL
INTO	IS	ISOLATION

JOIN	KEY	LANGUAGE
LAST	LDB	LEADING
LEAVE	LEFT	LESS
LEVEL	LIKE	LIMIT
LIST	LOCAL	LOCAL_TRANSACTION_ID
LOCALTIME	LOCALTIMESTAMP	LOOP
LOWER	MATCH	MAX
METHOD	MILLISECOND	MIN
MINUTE	MINUTE_MILLISECOND	MINUTE_SECOND
MOD	MODIFY	MODULE
MONETARY	MONTH	MULTISET
MULTISET_OF	NA	NAMES
NATIONAL	NATURAL	NCHAR
NEXT	NO	NONE
NOT	NULL	NULLIF
NUMERIC	OBJECT	OCTET_LENGTH
OF	OFF	OID
ON	ONLY	OPEN
OPERATION	OPERATORS	OPTIMIZATION
OPTION	OR	ORDER
OTHERS	OUT	OUTER
OUTPUT	OVERLAPS	PARAMETERS
PARTIAL	PENDANT	POSITION
PRECISION	PREORDER	PREPARE
PRESERVE	PRIMARY	PRIOR
PRIVATE	PRIVILEGES	PROCEDURE
PROTECTED	PROXY	QUERY
READ	REAL	RECURSIVE
REF	REFERENCES	REFERENCING
REGISTER	RELATIVE	RENAME

REPLACE	RESIGNAL	RESTRICT
RETURN	RETURNS	REVOKE
RIGHT	ROLE	ROLLBACK
ROLLUP	ROUTINE	ROW
ROWNUM	ROWS	SAVEPOINT
SCHEMA	SCOPE	SCROLL
SEARCH	SECOND	SECOND_MILLISECOND
SECTION	SELECT	SENSITIVE
SEQUENCE	SEQUENCE_OF	SERIALIZABLE
SESSION	SESSION_USER	SET
SET_OF	SETEQ	SHARED
SIBLINGS	SIGNAL	SIMILAR
SIZE	SMALLINT	SOME
SQL	SQLCODE	SQLERROR
SQLEXCEPTION	SQLSTATE	SQLWARNING
STATISTICS	STRING	STRUCTURE
SUBCLASS	SUBSET	SUBSETEQ
SUBSTRING	SUM	SUPERCLASS
SUPERSET	SUPERSETEQ	SYS_CONNECT_BY_PATH
SYS_DATE	SYS_DATETIME	SYS_TIME
SYS_TIMESTAMP	SYS_USER	SYSDATE
SYSDATETIME	SYSTEM_USER	SYSTIME
TABLE	TEMPORARY	TEST
THEN	THERE	TIME
TIMESTAMP	TIMEZONE_HOUR	TIMEZONE_MINUTE
TO	TRAILING	TRANSACTION
TRANSLATE	TRANSLATION	TRIGGER
TRIM	TRUE	TYPE
UNDER	UNION	UNIQUE
UNKNOWN	UPDATE	UPPER

USAGE	USE	USER
USING	UTIME	VALUE
VALUES	VARCHAR	VARIABLE
VARYING	VCLASS	VIEW
VIRTUAL	VISIBLE	WAIT
WHEN	WHENEVER	WHERE
WHILE	WITH	WITHOUT
WORK	WRITE	XOR
YEAR	YEAR_MONTH	ZONE

데이터 타입

문자 스트링 데이터 타입

정의와 특성

정의

CUBRID는 네 종류의 문자 스트링(character string) 타입을 지원한다.

- 고정길이 문자 스트링 : **CHAR**(*n*)
- 가변길이 문자 스트링 : **VARCHAR**(*n*)
- 고정길이 국가 문자 스트링 : **NCHAR**(*n*)
- 가변길이 국가 문자 스트링 : **NCHAR VARYING**(*n*)

다음은 문자 스트링 타입을 사용할 때 적용되는 규칙이다.

- 문자 스트링은 작은 따옴표로 감싸서 표현한다. SQL 구문 관련 파라미터인 **ansi_quotes**의 값에 따라 문자 스트링을 감싸는 부호로 큰 따옴표도 사용할 수 있다. **ansi_quotes** 값을 no로 설정하면 큰 따옴표로 감싼 문자열을 식별자로 처리하지 않고 문자 스트링으로 처리한다. 디폴트 값은 **yes**이다. 자세한 설명은 [구문/타입 관련 파라미터](#)를 참고한다.
- ANSI 표준에 따라 두 개의 문자 스트링 사이에 공간으로 취급할 수 있는 문자(예: 공백, 탭, 줄바꿈 등)가 있다면, 두 개의 문자 스트링은 연속된 하나의 문자 스트링으로 취급된다. 예를 들면, 다음과 같이 두 개의 문자 스트링 사이에 줄바꿈이 있는 경우가 있다.

```
'abc'
'def'
```

위 문자 스트링은 아래에 있는 하나의 문자 스트링과 동일하다.

```
'abcdef'
```

- 작은 따옴표 자체를 문자 스트링에 포함시키려면, 두 개의 작은 따옴표를 연속으로 입력하면 된다. 예를 들어, 아래의 왼쪽 문자 스트링은 실제로 오른쪽과 같이 저장된다.

```
' 'abcde' 'fghij'      'abcde'fghij'
```

- 모든 문자 스트링에 대한 토큰의 최대 크기는 16KB이다.
- 국가 문자 스트링은 다국어 환경에서 영어 외의 문자 스트링을 저장할 때 사용할 수 있다. 단, 아래 예제와 같이 문자 스트링을 감싸는 시작 따옴표 앞에 반드시 대문자 N을 붙여야 한다.

```
N'Härder'
```

특성

길이(Length)

CHAR나 **VARVAHR** 타입에서는 문자 스트링의 크기(바이트)를 지정하며, **NCHAR**나 **NCHAR VARYING** 타입에서는 문자의 개수를 지정한다.

입력된 문자 스트링이 지정된 길이를 초과하는 경우, 초과 부분이 공백 문자(ASCII로 32)인 경우에는 이를 절삭하고, 초과 부분이 공백 문자가 아닌 일반 문자인 경우에는 에러를 발생시킨다. 이때, 지정된 길이에 맞도록 데이터를 자르지(truncate) 않음에 유의한다.

또한, **CHAR**와 **NCHAR**와 같은 고정 길이 문자 스트링 타입에서는 선언한 길이에 고정되므로, 문자를 저장할 때 오른쪽에 공백 문자(trailing space)를 채운다. 한편, **VARCHAR** 또는 **NCHAR VARYING**과 같은 가변 길이 문자 스트링 타입에서는 공백 문자를 채우지 않고 실제 입력된 문자 스트링만큼 저장한다.

CHAR 또는 **VARCHAR** 타입에서 지정할 수 있는 최대 길이는 1,073,741,823이며, **NCHAR** 또는 **NCHAR VARYING** 타입에서 지정할 수 있는 최대 길이는 536,870,911이다. 또한, **CSQL** 문장으로 한 번에 입력 또는 출력할 수 있는 최대 크기는 8192KB이다.

문자 세트(Character Set, charset)

문자 세트(문자 집합)는 특정 문자(symbol)를 컴퓨터에 저장할 때, 어떠한 코드로 인코딩할 것인지에 대한 규칙이 정의된 집합을 의미한다. CUBRID가 지원하는 문자 세트는 다음과 같으며, **CUBRID_LANG** 환경 변수로 설정할 수 있다. 그 외의 문자 세트(예: UTF-8)인 경우 데이터 저장은 가능하나, 문자 함수 및 **LIKE** 검색을 지원하지 않는다.

문자 세트	CUBRID_LANG
8 비트 ISO 8859-1 Latin	en_US
KSC 5601-1992 (EUC-KR)	ko_KR.euckr

위 문자 세트의 모든 문자가 문자 스트링에 포함될 수 있다(**NULL** 문자는 'W0'로 나타낸다).

문자 세트의 정렬(Collating Character Set)

콜레이션(collation)은 어느 문자 세트가 설정된 상태에서 데이터베이스에 저장된 값들을 검색하거나 정렬하는 작업을 위해 문자들을 서로 비교할 때 사용하는 규칙들의 집합이다. 따라서 이러한 규칙은 **CHAR()** 또는 **VARCHAR()**와 같은 문자 스트링 데이터 타입에만 적용된다. 단, **NCHAR()**, **NCHAR VARYING()**과 같은 국가 문자 스트링 데이터 타입에서는 해당 문자 세트의 인코딩 알고리즘에 따라 정렬 규칙이 결정된다.

문자 스트링 변환(Character String Coercion)

고정길이와 가변길이 문자 스트링 사이에는 두 문자의 길이가 비교 가능할 수 있도록 자동 변환이 된다. 자동 변환은 동일한 문자 세트에 속하는 문자 스트링에만 적용된다.

예를 들어, 데이터 타입이 **CHAR(5)**인 컬럼을 추출하여 데이터 타입이 **CHAR(10)**인 컬럼에 삽입하는 경우 자동으로 데이터 타입이 **CHAR(10)**으로 변환되어 삽입된다. 문자 스트링을 명시적으로 변환할 수도 있는데, 이 때에는 **CAST** 연산자를 사용 한다([CAST 연산자](#) 참조).

CHAR(n)

설명

고정길이 문자 스트링은 **CHAR(n)**로 나타낸다. 여기서 *n*은 ASCII 문자 스트링의 바이트 수를 나타내는 것으로, 영문의 경우 한 문자는 1바이트를 차지한다. 한글의 경우 데이터 입력 환경의 문자 세트(character set)에 따라 한 문자가 차지하는 바이트 수가 다르므로 주의하여 지정한다(예: EUC-KR: 2바이트, UTF-8: 3바이트). *n*이 생략되면 길이는 디폴트 값인 1로 지정된다.

문자 스트링의 크기가 *n*을 초과하는 경우, 초과 부분이 공백 문자이면 절삭하여 처리하고, 초과 부분이 일반 문자이면 오류를 반환한다. *n*보다 작은 문자 스트링이 저장될 때에는 나머지 부분이 공백 문자로 채워진다.

CHAR(n)와 **CHARACTER(n)**는 같은 의미로 사용된다.

참고 사항

- **CHAR**는 항상 ISO 8859-1 Latin 문자 세트를 사용한다.
- *n*은 1부터 1,073,741,823(1G) 사이의 정수이다.
- 공백 값은 빈 따옴표("")로 처리하며, 이 경우 **LENGTH** 함수의 리턴 값은 0이 아니라 **CHAR(n)**에서 정의한 고정길이이다. 즉, CHAR(10)인 컬럼에 공백 값을 넣더라도 리턴 값은 10이며, *n*이 생략되면 디폴트 값이 **1**이므로 **CHAR(1)**로 간주된다.
- 채우는(padding) 문자로 사용되는 공백은 특수 문자를 비롯한 어느 문자보다도 작은 것으로 간주된다.

예제 1

CHAR(12)에 'pacesetter'를 저장하면 'pacesetter '가 된다(10자리 문자 스트링과 공백 문자 2개로 구성됨).
CHAR(10)에 'pacesetter '를 저장하면 'pacesetter'가 된다(10을 넘어서는 부분이 공백 문자이므로 이를 절삭하고 10자리 문자 스트링으로 구성됨).
CHAR(4)에 'pacesetter'를 저장하면 오류가 발생한다(문자 스트링의 크기가 4보다 크기 때문에 오류 발생).
CHAR에 'p '를 저장하면 'p'가 된다(*n*이 생략되면 길이는 기본값인 1로 지정됨).

예제 2

EUC-KR 환경에서 CHAR(10)에 '큐브리드'를 저장하면 정상 처리된다.
EUC-KR 환경에서 CHAR(10)에 '큐브리드'를 저장하면 LENGTH() 함수의 리턴 값은 10이다.
UTF-8 환경에서 CHAR(10)에 '큐브리드'를 저장하면 오류가 발생한다(한글 한 문자가 3바이트로 처리되는 경우).
UTF-8 환경에서 CHAR(12)에 '큐브리드'를 저장하면 정상 처리된다.

VARCHAR(n) 또는 CHAR VARYING(n)

설명

가변길이 문자 스트링은 **VARCHAR(n)**로 나타낸다. 여기서 *n*은 ASCII 문자 스트링의 바이트 수를 나타내는 것으로, 영문의 경우 한 문자는 1바이트를 차지한다. 한글의 경우 데이터 입력 환경의 문자 세트(character

set)에 따라 한 문자가 차지하는 바이트 수가 다르므로 주의하여 지정한다(예: EUC-KR: 2바이트, UTF-8: 3바이트). 만약, n 이 생략되면 길이는 최대 길이인 1,073,741,823로 지정된다.

문자 스트링의 크기가 n 을 초과하는 경우, 초과 부분이 공백 문자이면 절삭하여 처리하고, 초과 부분이 일반 문자이면 오류를 반환한다. n 보다 작은 문자 스트링이 저장되는 경우, **CHAR(n)**는 나머지 부분을 공백 문자로 채우지만 **VARCHAR(n)**에는 해당 문자 스트링 크기만큼만 저장한다.

VARCHAR(n)와 **CHARACTER VARYING(n)**, **CHAR VARYING(n)**은 같은 의미로 사용된다.

참고 사항

- **STRING**은 **VARCHAR**(최대 길이)와 같다.
- n 은 1부터 1,073,741,823(1G) 사이의 정수이다.
- 공백 값은 빈 따옴표("")로 처리하며, 이 경우 **LENGTH** 함수의 리턴 값은 0이다.

예제 1

VARCHAR(4)에 'pacesetter'를 저장하면 오류가 발생한다(문자 스트링의 크기가 4보다 크기 때문에 오류 발생. 지정된 크기를 초과하는 문자 스트링을 절삭하지 않음).
VARCHAR(12)에 'pacesetter'를 저장하면 'pacesetter'가 된다(10자리 문자 스트링으로 구성됨).
VARCHAR(12)에 'pacesetter '를 저장하면 'pacesetter '가 된다(10자리 문자 스트링과 공백 문자 2개로 구성됨).
VARCHAR(10)에 'pacesetter '를 저장하면 'pacesetter'가 된다(10을 넘어서는 부분이 공백 문자이므로 이를 절삭하고 10자리 문자 스트링으로 구성됨).
VARCHAR에 'p'를 저장하면 'p'가 된다(n 이 생략되면 최대 길이는 디폴트 값인 1,073,741,823로 지정되고, 저장 시 나머지 부분은 공백 문자로 채워지지 않음).

예제 2

EUC-KR 환경에서 **VARCHAR(10)**에 '큐브리드'를 저장하면 정상 처리된다.
EUC-KR 환경에서 **CHAR(10)**에 '큐브리드'를 저장한 후 **CHAR_LENGTH()** 함수를 사용하면 8이 출력된다.
UTF-8 환경에서 **VARCHAR(10)**에 '큐브리드'를 저장하면 오류가 발생한다(UTF-8 문자 세트에서 한글 한 문자는 3바이트로 처리됨).
UTF-8 환경에서 **VARCHAR(12)**에 '큐브리드'를 저장하면 정상 처리된다.

STRING

설명

STRING은 가변길이 문자 스트링 데이터 타입이다. **STRING**은 [VARCHAR](#)를 최대 길이로 지정한 것과 같다. 즉 **STRING**은 **VARCHAR(1,073,741,823)**과 동일하다.

NCHAR(n)

설명

NCHAR(*n*)는 영어권 외 국가에서 문자 스트링을 저장할 때 사용할 수 있는 타입으로서, 앞에서 설명한 CUBRID 지원 문자 세트의 경우에만 사용할 수 있다. *n*은 문자의 개수를 나타내며, *n*이 생략되면 길이는 디폴트 값 1로 지정된다. 문자의 개수가 *n*을 초과하는 경우, 초과 부분이 공백 문자인 경우에는 이를 절삭하여 처리하고, 초과 부분이 일반 문자인 경우에는 오류를 반환한다. 또한, 문자의 개수가 *n*보다 작은 문자 스트링이 저장될 때에는 나머지 부분이 공백 문자로 채워진다.

국가 문자 스트링 타입에 한글을 저장하기 위해서는 테이블 생성 이전에, 운영 체제의 로캘(locale)을 korean으로 설정하거나 **CUBRID_LANG** 환경 변수의 값을 **ko_KR.euckr**로 설정해야 한다.

NCHAR(*n*)와 **NATIONAL CHAR(*n*)**, 그리고 **NATIONAL CHARACTER(*n*)**는 같은 의미로 사용된다.

참고 사항

- n*은 1부터 5,368,709,111사이의 정수이다.
- 하나의 데이터베이스에서 사용할 수 있는 국가 문자 세트는 하나로 지정되어 있다. 예를 들어, 같은 데이터베이스에서 8비트 ISO 8889-1 Latin 코드 세트를 사용하면서 동시에 EUC 인코딩 코드 세트를 사용할 수는 없다.
- 국가 문자 스트링으로 선언된 컬럼에 고정이든 가변이든 일반 스트링을 지정하면 오류가 발생한다.
- 서로 다른 문자 세트를 혼용하여 사용할 경우에도 오류가 발생한다.

예제

EUC-KR 환경에서 NCHAR(5)에 '큐브리드'를 저장하면 정상적으로 저장된다.
EUC-KR 환경에서 NCHAR(5)에 '큐브리드'를 저장한 후 CHAR_LENGTH() 함수를 사용하면 5가 출력된다.
UTF-8 환경에서 NCHAR(5)에 '큐브리드'를 저장하면 오류가 발생한다(UTF-8 문자 세트를 지원하지 않음).

NCHAR VARYING(n)

설명

NCHAR VARYING(*n*)은 가변길이의 국가 문자 스트링 타입이며, 이에 대한 설명은 [NCHAR\(*n*\)](#)의 설명 및 참고 사항을 참고한다. **NCHAR(*n*)**와의 차이점은 문자의 개수가 *n*보다 작아도 오른쪽 부분에 공백 문자(trailing space)를 채우지 않는다는 것이다.

NCHAR VARYING(*n*)와 **NATIONAL CHAR VARYING(*n*)**, 그리고 **NATIONAL CHARACTER VARYING(*n*)**는 같은 의미로 사용된다.

예제

EUC-KR 환경에서 NCHAR VARYING(5)에 '큐브리드'를 저장하면 정상적으로 저장된다.
EUC-KR 환경에서 NCHAR VARYING(5)에 '큐브리드'를 저장한 후 CHAR_LENGTH() 함수를 사용하면 4가 출력된다.

UTF-8 환경에서 NCHAR VARYING(5)에 '큐브리드'를 저장하면 오류가 발생한다(UTF-8 문자 세트를 지원하지 않음).

비트 스트링 데이터 타입

정의와 특성

정의

비트 스트링은 0과 1로 이루어진 이진 값의 순열(sequence)이다. CUBRID는 두 가지 비트 스트링을 지원한다.

- 고정길이 비트 스트링(**BIT**)
- 가변길이 비트 스트링(**BIT VARYING**)

메소드의 인자나 속성의 도메인으로 비트 스트링을 사용할 수 있으며, 비트 스트링 리터럴은 2진수 형식이나 16진수 형식을 사용한다.

2진수 형식으로 사용할 때에는 다음과 같이 문자 **B** 뒤에 0과 1로 이루어진 스트링을 붙이거나, **0b** 뒤에 값을 붙여 표현한다.

```
B'1010'  
0b1010
```

16진수 형식은 대문자 **X** 뒤에 0-9 그리고 A-F 문자로 이루어진 스트링을 붙이거나 **0x** 뒤에 값을 붙여 표현한다. 예를 들어, 아래는 앞에서 2진수로 표현한 것과 같은 값을 16진수로 나타낸 것이다.

```
X'a'  
0xA
```

16진수에서 사용되는 문자는 대소문자를 구분하지 않는다. 즉, X'4f'와 X'4F'는 같은 값으로 간주한다.

특성

길이(Length)

비트 스트링이 테이블 속성이나 메소드 선언에 사용될 때에는 최대 길이를 표시해야 한다. 비트 스트링이 가질 수 있는 최대 길이는 1,073,741,823비트이다.

비트 스트링의 변환(Bit String Coercion)

고정길이와 가변길이 비트 스트링 간에는 서로 비교를 위하여 자동 변환이 이루어진다. 명시적인 변환은 **CAST** 연산자를 이용해야 한다.

BIT(n)

설명

고정길이 2진수 혹은 16진수 비트 스트링은 **BIT(n)**로 나타내는데, 여기서 *n*은 최대 비트의 개수를 나타낸다. 만약, *n*이 생략되면 길이는 1로 지정된다.

참고 사항

- n 은 0보다 큰 숫자여야 한다.
- 스트링의 크기가 n 을 넘어설 경우에는 오류로 처리된다.
- n 보다 작은 비트 스트링이 저장될 때에는 나머지 오른쪽 부분이 0으로 채워진다.

예제

```
CREATE TABLE bit_tbl(a1 BIT, a2 BIT(1), a3 BIT(8), a4 BIT VARYING);
INSERT INTO bit_tbl VALUES (B'1', B'1', B'1', B'1');
INSERT INTO bit_tbl VALUES (0b1, 0b1, 0b1, 0b1);
INSERT INTO bit_tbl(a3,a4) VALUES (B'1010', B'1010');
INSERT INTO bit_tbl(a3,a4) VALUES (0xaa, 0xaa);
SELECT * FROM bit_tbl;
;xr

=== <Result of SELECT Command in Line 6> ===

a1                a2                a3                a4
=====
X'8'              X'8'              X'80'             X'8'
X'8'              X'8'              X'80'             X'8'
NULL              NULL              X'a0'             X'a'
NULL              NULL              X'aa'             X'aa'
```

BIT VARYING(n)

설명

가변길이 비트 스트링은 **BIT VARYING(n)**으로 나타낸다. 여기서 n 은 최대 비트의 개수를 나타낸다. 만약, n 이 생략되면 최대 길이인 1,073,741,823로 지정된다.

참고 사항

- 스트링의 크기가 n 을 넘어설 경우에는 오류로 처리된다.
- n 보다 작은 비트 스트링이 저장될 때에도 나머지 부분이 0으로 채워지지 않는다.
- n 은 0보다 큰 숫자여야 한다.

예제

```
CREATE TABLE bitvar_tbl(a1 BIT VARYING, a2 BIT VARYING(8));
INSERT INTO bitvar_tbl VALUES (B'1', B'1');
INSERT INTO bitvar_tbl VALUES (0b1010, 0b1010);
INSERT INTO bitvar_tbl VALUES (0xaa, 0xaa);
INSERT INTO bitvar_tbl(a1) VALUES (0xaaa);
SELECT * FROM bitvar_tbl;
;xr

=== <Result of SELECT Command in Line 6> ===

a1                a2
=====
X'8'              X'8'
X'a'              X'a'
X'aa'             X'aa'
X'aaa'            NULL

INSERT INTO bitvar_tbl(a2) VALUES (0xaaa);
;xr
```

ERROR: Data overflow coercing X'aaa' to type bit varying.

수치형 데이터 타입

정의와 특성

정의

CUBRID는 정수 또는 실수를 저장하기 위해 다음의 수치형 데이터 타입을 지원한다.

CUBRID가 지원하는 수치형 데이터 타입

타입	Bytes	최소값	최대값	정확/근사치
SHORT SMALLINT	2	-32,768	32,767	정확한 수치
INT INTEGER	4	-2,147,483,648	+2,147,483,647	정확한 수치
BIGINT	8	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807	정확한 수치
NUMERIC DECIMAL	16	정밀도 p : 1 스케일 s : 0	정밀도 p : 38 스케일 s : 38	정확한 수치
FLOAT REAL	4	-3.402823466E+38 (ANSI/IEEE 754-1985 표준)	+3.402823466E+38 (ANSI/IEEE 754-1985 표준)	근사치 부동소수점: 7 자리
DOUBLE DOUBLE PRECISION	8	- 1.7976931348623157E+308 (ANSI/IEEE 754-1985 표준)	+1.7976931348623157E+308 (ANSI/IEEE 754-1985 표준)	근사치 부동소수점: 15 자리
MONETARY	12	-3.402823466E+38	+3.402823466E+38	근사치

수치형 데이터 타입은 정확한(exact) 형태와 근사치(approximate) 형태로 구분된다. 정확한 수치형 데이터 타입(**SMALLINT**, **INT**, **BIGINT**, **NUMERIC**)은 정확하고 일관된 값을 가져야 하는 경우에 사용된다. 근사치 수치형 데이터 타입(**FLOAT**, **DOUBLE**, **MONETARY**)은 리터럴 값이 같아도 시스템에 따라 다르게 해석될 수 있으므로 주의한다.

CUBRID는 수치형 데이터 타입에 대해 UNSIGNED 타입을 지원하지 않는다.

특성

정밀도와 스케일(Precision and Scaling)

숫자 데이터 타입의 정밀도(precision)는 그 값이 유지할 수 있는 유효한 자릿수로 정의된다. 이는 정확한 수치형이든 근사치 수치형이든 마찬가지이다.

스케일(scale)은 소수점 이하의 자릿수를 나타내는데, 정확한 수치형에서만 의미가 있다. 정확한 수치형으로 선언된 속성은 항상 고정된 정밀도와 스케일을 갖게 된다. **NUMERIC**(또는 **DECIMAL**) 데이터 타입은 항상 최소한 한 자리의 정밀도를 갖는다. 그리고, 스케일의 범위는 0과 선언된 정밀도 사이여야 한다. 스케일이 정밀도보다 클 수는 없다. **INTEGER**나 **SMALLINT**, **BIGINT** 데이터 타입에서는 스케일은 0이고(즉, 소수점 이하가 없음), 정밀도는 시스템에 의해 고정된다.

수치형 리터럴(Numeric Literals)

수치형 값을 입력하기 위해서는 특별한 기호가 사용될 수 있는데, 플러스(+)는 양수를, 마이너스(-)는 음수를 나타내는 데 사용한다. 과학용 표기법이 사용될 수도 있다. 화폐 값을 표현하기 위하여 시스템에 지정된 통화 기호를 사용할 수도 있다. 수치형 리터럴로 표현 가능한 최대 정밀도는 255이다.

수치형 변환(Numeric Coercions)

모든 수치형 데이터 타입은 상호 비교 가능하고, 이를 위해 서로 공통된 수치형 데이터 타입으로 자동 변환이 이루어진다. 명시적인 변환은 **CAST** 연산자를 이용해야 한다. 서로 다른 수치형 데이터가 서로 정렬되거나, 수식에서 계산될 때에는 시스템에 의하여 자동으로 변환된다. 예를 들어, **FLOAT** 타입의 속성 값과 **INTEGER** 타입의 속성 값을 더하게 되면, 시스템이 자동적으로 **INTEGER** 속성 값을 가장 근사한 **FLOAT** 값으로 변환한 후 덧셈을 수행한다.

주의 CUBRID 2008 R2.0 미만 버전에서는 입력된 상수가 **INTEGER** 범위를 넘어서면 **NUMERIC**으로 처리되었으나, CUBRID 2008 R2.0 이상 버전에서는 **BIGINT**로 처리된다.

INT 또는 INTEGER

설명

INTEGER 데이터 타입은 정수 표현을 위해 사용하며, 표현할 수 있는 값의 범위는 -2,147,483,648에서 +2,147,483,647이다. 작은 정수를 표현하기 위해 **SMALLINT**를 사용하거나, 큰 정수를 표현하기 위해 **BIGINT**를 사용할 수 있다.

INTEGER

참고 사항

- 만약, **INT** 타입에 실수가 입력되면, 소수점 아래 숫자가 반올림되어 정수값이 저장된다.
- INTEGER**와 **INT**는 같은 의미로 사용된다.

예제

INTEGER에 8934를 지정하면 8934가 저장됨.
INTEGER에 7823467를 지정하면 7823467이 저장됨.
INTEGER에 89.8를 지정하면 90이 저장됨(소수점 뒤의 수치는 반올림됨).
INTEGER에 3458901122를 지정하면 오류가 발생함(표현 가능 범위를 초과하면 오류 발생).

SHORT 또는 SMALLINT

설명

SMALLINT 데이터 타입은 작은 정수 표현을 위해 사용되며, 표현할 수 있는 값의 범위는 -32,768에서 +32,767이다.

SMALLINT

참고 사항

SMALLINT 타입에 실수가 입력되면, 소수점 아래 숫자가 반올림되어 정수값이 저장된다.

예제

SMALLINT에 8934를 지정하면 8934가 저장됨.
SMALLINT에 34.5를 지정하면 35가 저장됨 (소수점 이하의 숫자는 반올림됨).
SMALLINT에 23467를 지정하면 23467이 저장됨.
SMALLINT에 89354를 지정하면 오류가 발생함 (표현 가능 범위를 초과하면 오류 발생).

BIGINT

설명

BIGINT 데이터 타입은 큰 정수 표현을 위해 사용되며, 표현할 수 있는 값의 범위는 -9,223,372,036,854,775,808에서 9,223,372,036,854,775,807이다.

BIGINT

참고 사항

- **BIGINT** 타입에 실수가 입력되면, 소수점 아래 숫자가 반올림되어 정수값이 저장된다.
- 정밀도와 표현할 수 있는 범위를 기준으로는 다음과 같이 정렬할 수 있다.

SMALLINT \subset **INTEGER** \subset **BIGINT** \subset **NUMERIC**

예제

BIGINT에 8934를 지정하면 8934가 저장됨.
BIGINT에 89.1을 지정하면 89가 저장됨.
BIGINT에 89.8을 지정하면 90이 저장됨 (소수점 뒤의 수치는 반올림됨).
BIGINT에 3458901122를 지정하면 3458901122가 저장됨.

NUMERIC 또는 DECIMAL

설명

NUMERIC 또는 **DECIMAL** 데이터 타입은 고정 소수점 숫자를 표현하기 위해 사용되며, 다음과 같이 전체 자리 수(정밀도)와 소수점 아래 자릿수(스케일)을 옵션으로 지정하여 정의할 수 있다. 정밀도 *p*의 최소값은 1이고 최대값은 38이며, 정밀도 *p*가 생략되면 디폴트 값은 15이므로, 정수부가 15자리를 초과하는 데이터를

입력할 수 없다. 또한, 스케일 s 가 생략되면 스케일의 디폴트 값은 0이므로 소수점 아래 첫째 자리에서 반올림한 정수를 반환한다.

```
NUMERIC [(p[, s])]
```

참고 사항

- 정밀도는 반드시 스케일 이상이어야 한다.
- 정밀도는 (데이터의 정수부 자리 수 + 스케일) 이상이 되도록 지정한다.
- NUMERIC**과 **DECIMAL**, 그리고 **DEC**는 같은 의미로 사용된다.

예제

```
NUMERIC에 12345.6789를 지정하면 123460이 저장됨 (스케일 디폴트 값은 0이므로 소수점 아래 첫째 자리에서 반올림함).
NUMERIC(4)에 12345.6789를 지정하면 오류가 발생함 (정밀도는 데이터의 정수부 자릿수 이상이어야 함).
NUMERIC(3,4)를 선언하면 오류가 발생함 (정밀도는 스케일 이상이어야 함).
NUMERIC(4,4)에 0.123456789를 지정하면 .1235가 저장됨 (소수점 아래 다섯째 자리에서 반올림함).
NUMERIC(4,4)에 -0.123456789를 지정하면 -.1235가 저장됨 (소수점 아래 다섯째 자리에서 반올림한 후, - 부호를 붙임).
```

FLOAT 또는 REAL

설명

FLOAT(또는 **REAL**) 데이터 타입은 부동 소수점 숫자를 표현하기 위해 사용되며, 표현할 수 있는 값의 범위는 -3.402823466E+38 에서 -1.175494351E-38, 0, 그리고 +1.175494351E-38 에서 +3.402823466E+38이며, 이는 ANSI/IEEE 754-1985 표준을 준수한다.

정밀도 p 의 최소값은 1이고 최대값은 38이며, 정밀도 p 가 생략되거나 7 이하로 지정되면 단일 정밀도(single-precision, 7자리의 유효 숫자)로 표현된다. 만약 정밀도 p 가 7보다 크고 38 이하이면, 이중 정밀도(double-precision, 15자리의 유효 숫자)로 표현되며, **DOUBLE** 데이터 타입으로 변환된다.

```
FLOAT[(p)]
```

참고 사항

- FLOAT** 타입의 유효 자리 수는 7이다.
- FLOAT** 타입으로 표현할 수 있는 범위는 CUBRID가 수행되는 시스템마다 다르다.
- FLOAT** 타입은 근사치 데이터를 저장하므로 데이터 비교 시 주의해야 한다.
- FLOAT**와 **REAL**은 같은 의미로 사용된다.

예제

```
FLOAT에 -1234.56789를 지정하면 -1.234568e+003이 저장된다 (정밀도가 생략되면, 7개의 유효 숫자로 표현하므로 8번째 숫자를 반올림함).
FLOAT(5)에 1234.56789를 지정하면 1.234568e+003이 저장된다 (정밀도가 7 이하이면, 7개의 유효 숫자로 표현하므로 8번째 숫자를 반올림함).
```

FLOAT(5)에 12345678.9를 지정하면 1.234568e+007이 저장된다(정밀도가 7 이하이면, 7개의 유효 숫자로 표현하므로 8번째 숫자를 반올림함).
FLOAT(10)에 12345678.9를 지정하면 1.2345678900000000e+007이 저장된다(정밀도가 7보다 크고 38 이하이면, 15개의 유효 숫자로 표현하므로 0을 채움).

DOUBLE 또는 DOUBLE PRECISION

설명

DOUBLE 데이터 타입은 부동 소수점 숫자를 표현하기 위해 사용되며, 표현할 수 있는 값의 범위는 -1.7976931348623157E+308에서 -2.2250738585072014E-308, 0, 그리고 2.2250738585072014E-308에서 1.7976931348623157E+308이며, 이는 ANSI/IEEE 754-1985 표준을 준수한다.

정밀도 p 를 지정할 수 없으며, 이 타입이 지정된 데이터는 이중 정밀도(double-precision, 15자리의 유효 숫자)로 표현된다.

DOUBLE

참고 사항

- **DOUBLE**의 유효 자리 수는 15자리이다.
- **DOUBLE** 타입으로 표현할 수 있는 범위는 CUBRID가 수행되는 시스템마다 다르다.
- **DOUBLE** 타입은 근사치 데이터를 저장하므로 데이터 비교 시 주의해야 한다.
- **DOUBLE**과 **DOUBLE PRECISION**은 같은 의미로 사용된다.

예제

DOUBLE에 1234.56789를 지정하면 1.2345678900000000e+003이 저장된다.

MONETARY

설명

MONETARY 데이터 타입은 근사치 숫자 타입이다. 시스템에 따라 다를 수 있지만, 표현할 수 있는 값의 범위는 **FLOAT**과 같으며, 소수점 이하 둘째 자리까지 표시된다. 또한, 1,000 단위마다 쉼표가 붙는다.

MONETARY

참고 사항

달러 기호나 소수점을 사용할 수도 있으나, 쉼표(.)는 사용할 수 없다.

예제

MONETARY에 12345.6789를 지정하면 \$12,345.68이 저장된다(소수점 아래 세 번째 자리에서 반올림함).
MONETARY에 123456789를 지정하면 \$123,456.789.00이 저장된다.

날짜/시간 데이터 타입

정의와 특성

정의

날짜/시간 데이터 타입은 날짜, 시간 혹은 이 두 가지를 모두 표현할 때 사용하는 데이터 타입으로, CUBRID는 다음과 같은 데이터 타입을 지원한다.

CUBRID가 지원하는 날짜/시간 데이터 타입

타입	최소값	최대값	비고
DATE	0001 년 1 월 1 일	9999 년 12 월 31 일	0 년, 0 월, 0 일은 입력할 수 없다.
TIME	00 시 00 분 00 초	23 시 59 분 59 초	0 시, 0 분, 0 초는 입력할 수 있다.
TIMESTAMP	1970 년 1 월 1 일 0 시 0 분 0 초 (GMT)	2038 년 1 월 19 일 3 시 14 분 7 초 (GMT)	TIMESTAMP 컬럼의 디폴트 값을
	1970 년 1 월 1 일 9 시 0 분 0 초 (KST)	2038 년 1 월 19 일 12 시 14 분 7 초 (KST)	SYS_TIMESTAMP 로 지정하더라도, 데이터 삽입/갱신 시점의 TIMESTAMP 값이 저장되지 않으므로 주의한다.
DATETIME	0001 년 1 월 1 일 0 시 0 분 0.000 초	9999 년 12 월 31 일 23 시 59 분 59.999 초	

특성

범위와 해상도(Range and Resolution)

- 시간 값의 표현은 기본적으로 24시간 시스템에 의하여 그 범위가 결정된다. 날짜는 그레고리력(Gregorian calendar)을 따른다. 이 두 제약점을 벗어나는 값이 날짜나 시간으로 입력되면 오류가 발생한다.
- DATE** 중 연도의 범위는 0001 ~ 9999 AD이다.
- CUBRID 2008 R3.0 버전부터는 연도를 두 자리만 표기하면, 00~69는 2000~2069로 변환되고, 70~99는 1970~1999로 변환된다. R3.0 미만 버전에서는 01~99까지의 두 자리 연도를 표기하면, 각각 0001~0099로 변환된다.
- TIMESTAMP**의 범위는 GMT로 1970년 1월 1일 0시 0분 0초부터 2038년 1월 19일 03시 14분 07초까지이다. KST (GMT+9)로는 1970년 1월 1일 9시 0분 0초부터 2038년 1월 19일 12시 14분 07초까지 저장할 수 있다.

- 날짜, 시간, 타임스탬프와 관련된 연산은 시스템의 반올림 시스템에 따라 결과가 달라질 수 있다. 이러한 경우, 시간과 타임스탬프는 가장 근접한 초를 최소 해상도로, 날짜는 가장 근접한 날짜를 최소 해상도로 하여 결정된다.

변환(Coercion)

날짜/시간 데이터 타입의 값은 서로 똑같은 항목을 가지고 있는 경우에만 **CAST** 연산자를 이용한 명시적인 변환이 가능하며, 묵시적 변환은 수행되지 않는다. 아래의 표는 명시적 변환이 가능한 타입을 설명한다. 날짜/시간 데이터 타입 간 산술 연산에 대한 내용은 [날짜/시간 데이터 타입의 산술 연산과 타입 변환](#)을 참고한다.

날짜/시간 데이터 타입의 명시적 변환

TO					
FROM		DATE	TIME	DATETIME	TIMESTAMP
	DATE	-	X	O	O
	TIME	X	-	X	X
	DATETIME	O	O	-	O
	TIMESTAMP	O	O	O	-

DATE

설명

DATE 데이터 타입은 연도(*yyyy*), 월(*mm*), 일(*dd*)을 표현하며, 지원 범위는 '01/01/0001'에서 '12/31/9999'까지이다. 연도는 생략 가능하며, 생략될 경우 현재 시스템의 연도 값이 자동으로 지정된다. 출력 포맷 및 입력 포맷은 다음과 같다.

```
'mm/dd{/yyyy}'
'[yyyy-]mm-dd'
```

참고 사항

- 모든 항목은 정수 형태로 입력되어야 한다.
- CSQL은 'MM/DD/YYYY' 포맷으로 날짜 값을 출력하고, JDBC 응용 프로그램 및 CUBRID 매니저는 'YYYY-MM-DD' 포맷으로 날짜 값을 출력한다.
- 문자 스트링 타입의 데이터를 **DATE** 타입으로 변환하는 함수는 **TO_DATE()**이다.

예제

```
DATE '2008-10-31'은 '10/31/2008'로 저장된다.
DATE '10/31'은 '10/31/2010'으로 저장된다(연도가 생략되면 현재 연도가 자동으로 지정됨).
DATE '00-10-31'은 '10/31/2000'로 저장된다.
DATE '0000-10-31'은 예외가 출력된다(연도의 최소값은 1).
DATE '70-10-31'은 '10/31/1970'로 저장된다.
```


DATE '0070-10-31'은 '10/31/0070'로 저장된다.

TIME

설명

TIME 데이터 타입은 시각(*hh*), 분(*mi*), 초(*ss*)를 표현하며, 지원 범위는 '00:00:00'에서 '23:59:59'까지이다.

초는 생략 가능하며, 생략될 경우 0초로 지정된다. 입력 포맷은 12시간 표기법(AM/PM 표기법) 또는 24시간 표기법이 모두 허용되며, 다음과 같이 작성한다.

```
'hh:mi [:ss] [am | pm]'
```

참고 사항

- 모든 항목은 정수로 입력되어야 한다.
- CSQL은 항상 AM/PM 표기법으로 시간 값을 출력하고, JDBC 응용 프로그램 및 CUBRID 매니저는 24시간 표기법으로 시간 값을 출력한다.
- 24시간 표기법으로 시간 값을 입력할 때에도 AM/PM을 지정할 수 있으며, 이때 시간 값과 지정된 AM 또는 PM이 일치하지 않으면 오류가 발생한다.
- 모든 시간 값은 데이터베이스에는 24시간 표기법으로 저장되고, C API 함수인 **db_time_decode**를 이용하면 24시간 표기법으로 반환된다.
- 문자 스트링 타입의 데이터를 **TIME** 타입으로 변환하는 함수는 **TO_TIME()**이다.

예제

```
TIME '00:00:00'은 '12:00:00 AM'으로 출력된다.
TIME '1:15'는 '01:15:00 AM'으로 간주된다.
TIME '13:15:45'는 '01:15:45 PM'으로 간주된다.
TIME '13:15:45 pm'은 정상적으로 저장된다.
TIME '13:15:45 am'은 오류가 발생한다(주어진 시간 값과 AM/PM이 불일치).
```

TIMESTAMP

설명

TIMESTAMP 데이터 타입은 날짜(년, 월, 일)와 시간(시, 분, 초)을 결합한 데이터 값을 표현하며, GMT로 1970-01-01 00:00:00부터 2038-01-19 03:14:07까지 표현 가능하다. 이 범위를 초과하거나 밀리초 단위의 시간 데이터를 저장하는 경우라면, **DATETIME** 데이터 타입을 이용할 수 있다. **TIMESTAMP** 데이터 타입의 입력 포맷은 다음과 같다.

```
'hh:mi [:ss] [am|pm] mm/dd [/yyyy] '
'hh:mi [:ss] [am|pm] [yyyy-]mm-dd'

'mm/dd [/yyyy] hh:mi [:ss] [am|pm] '
'[yyyy-]mm-dd hh:mi [:ss] [am|pm] '
```

참고 사항

- 모든 항목은 정수로 입력되어야 한다.

- 연도를 생략하면 디폴트로 현재 연도가 지정되고, 시간 값(시/분/초)을 생략하면 12:00:00 AM으로 지정된다.
- 시스템의 현재 타임스탬프 값은 **SYS_TIMESTAMP**(또는 **SYSTIMESTAMP**, **CURRENT_TIMESTAMP**) 함수를 이용하여 **TIMESTAMP** 데이터 타입에 저장할 수 있다. 단, 테이블 생성시 **TIMESTAMP** 타입 컬럼에 디폴트 값으로 **SYS_TIMESTAMP**를 지정하면, 데이터 **INSERT** 시점이 아닌 테이블 생성 시점의 타임스탬프 값이 디폴트 값으로 지정됨을 주의한다.
- **TIMESTAMP** 함수 또는 **TO_TIMESTAMP** 함수를 사용하면, 문자 스트링 데이터 타입의 데이터를 **TIMESTAMP** 데이터 타입으로 변환할 수 있다.

예제

```
TIMESTAMP '10/31'은 '12:00:00 AM 10/31/2010'으로 출력된다(연도/시간이 생략될 경우, 디폴트 값으로 출력).
TIMESTAMP '10/31/2008'은 '12:00:00 AM 10/31/2008'로 출력된다(시간이 생략될 경우, 디폴트 값으로 출력).
TIMESTAMP '13:15:45 10/31/2008'은 '01:15:45 PM 10/31/2008'로 출력된다.
TIMESTAMP '01:15:45 PM 2008-10-31'은 '01:15:45 PM 10/31/2008'로 출력된다.
TIMESTAMP '13:15:45 2008-10-31'은 '01:15:45 PM 10/31/2008'로 출력된다.
TIMESTAMP '10/31/2008 01:15:45 PM'은 '01:15:45 PM 10/31/2008'로 출력된다.
TIMESTAMP '10/31/2008 13:15:45'는 '01:15:45 PM 10/31/2008'로 출력된다.
TIMESTAMP '2008-10-31 01:15:45 PM'은 '01:15:45 PM 10/31/2008'로 출력된다.
TIMESTAMP '2008-10-31 13:15:45'는 '01:15:45 PM 10/31/2008'로 출력된다.
TIMESTAMP '2099-10-31 01:15:45 PM'은 오류가 발생한다(TIMESTAMP 표현 가능 범위 초과).
```

DATETIME

설명

DATETIME 타입은 날짜(년, 월, 일)와 시간(시, 분, 초, 밀리초)을 결합한 데이터 값을 표현하며, GMT로 0001-01-01 00:00:00.000부터 9999-12-31 23:59:59.999까지 표현할 수 있다. **DATETIME** 타입 데이터의 입력 포맷은 다음과 같다.

```
'hh:mi[:ss[.ff]] [am|pm] mm/dd[/yyyy]'
'hh:mi[:ss[.ff]] [am|pm] [yyyy-]mm-dd'
'mm/dd[/yyyy] hh:mi[:ss[.ff]] [am|pm]'
'[yyyy-]mm-dd hh:mi[:ss[.ff]] [am|pm]'
```

참고 사항

- 모든 항목은 정수로 입력되어야 한다.
- 연도를 생략하면 디폴트로 현재 연도가 지정되고, 시간 값(시/분/초/밀리초)을 생략하면 12:00:00.000 AM으로 지정된다.
- 시스템의 현재 타임스탬프 값은 **SYS_DATETIME**(또는 **SYSDATETIME**, **CURRENT_DATETIME**, **CURRENT_DATETIME()**, **NOW()**)를 이용하여 **DATETIME** 타입에 저장할 수 있다. 단, 테이블 생성시 **DATETIME** 타입 컬럼에 디폴트 값으로 **SYS_DATETIME**을 지정하면, 데이터 **INSERT** 시점이 아닌 테이블 생성 시점의 시간 값이 디폴트 값으로 지정됨을 주의한다.
- 문자 스트링 타입의 데이터를 **DATETIME** 타입으로 변환하는 함수는 **TO_DATETIME()**이다.

예제

```
DATETIME '10/31'은 '12:00:00.000 AM 10/31/2010'으로 출력된다(연도/시간이 생략될 경우, 디폴트 값으로 출력).
DATETIME '10/31/2008'은 '12:00:00.000 AM 10/31/2008'로 출력된다.
DATETIME '13:15:45 10/31/2008'은 '01:15:45.000 PM 10/31/2008'로 출력된다.
DATETIME '01:15:45 PM 2008-10-31'은 '01:15:45.000 PM 10/31/2008'로 출력된다.
DATETIME '13:15:45 2008-10-31'은 '01:15:45.000 PM 10/31/2008'로 출력된다.
DATETIME '10/31/2008 01:15:45 PM'은 '01:15:45.000 PM 10/31/2008'로 출력된다.
DATETIME '10/31/2008 13:15:45'는 '01:15:45.000 PM 10/31/2008'로 출력된다.
DATETIME '2008-10-31 01:15:45 PM'은 '01:15:45.000 PM 10/31/2008'로 출력된다.
DATETIME '2008-10-31 13:15:45'는 '01:15:45.000 PM 10/31/2008'로 출력된다.
DATETIME '2099-10-31 01:15:45 PM'은 '01:15:45.000 PM 10/31/2099'로 출력된다.
```

집합형 데이터 타입

정의와 특성

정의

여러 개의 데이터 값을 하나의 속성에 저장할 수 있도록 하는 것은 관계형 데이터베이스의 확장 기능이다. 컬렉션의 각 원소는 서로 다른 데이터 타입이 될 수도 있고, 심지어 서로 다른 클래스(단, 가상 클래스 제외) 도메인으로 가질 수도 있다. 예를 들어, SET (INTEGER, tbl_1)은 정수, 또는 사용자가 정의한 클래스 tbl_1의 행 값의 집합을 도메인으로 지정할 수 있다. 또한, SET ()과 같이 도메인 리스트를 지정하지 않으면 사용자 정의 클래스를 포함한 모든 데이터 타입을 원소로 허용한다는 의미이다.

도메인 리스트가 두 개 이상인 컬렉션 타입 컬럼의 데이터를 조회할 수 있는 환경은 **csql** 유틸리티와 CCI API이며, CUBRID 매니저 및 그 밖의 CUBRID API(JDBC, ODBC, OLEDB, PHP)에서는 조회할 수 없음을 유의한다.

CUBRID가 지원하는 집합형 데이터 타입

타입	설명	타입 정의	입력 데이터	저장 데이터
SET	중복을 허용하	col_name SET	{'c','c','c','b','b', 'a'}	{'a','b','c'}
	지 않는	col_name SET	{3,3,3,2,2,1,0,'c','c','c','b','b', 'a'}	{0,1,2,3,'a','b','c'}
	합집합	(int,	'a'}	
		VARCHAR(20))		
MULTISET	중복을 허용하	col_name MULTISET	{'c','c','c','b','b', 'a'}	{'a','b','b','c','c','c'}
	는 합집	col_name VARCHAR(20)	{3,3,3,2,2,1,0,'c','c','c','b','b', 'a'}	{0,1,2,2,3,3,3,'a','b','b', 'c','c','c'}
	합	col_name MULTISET (int,		
		VARCHAR(20))		

LIST	중복을	col_name LIST	{'c','c','c','b','b', 'a'}	{'c','c','c','b','b','a'}
SEQUENCE	허용하	VARCHAR(20)	{3,3,3,2,2,1,0,'c','c','c','b','b',	{3,3,3,2,2,1,0,'c','c','c','b','b','a'}
	고, 데	col_name LIST	'a'}	
	이터 입	(int,		
	력 순서	VARCHAR(20))		
	대로 저			
	장하는			
	합집합			

위의 표에 나타난 바와 같이, 컬렉션 타입으로 지정되는 값은 중괄호('{, }') 안에 각 값들을 쉼표(,)로 구분하여 나열할 수 있다.

특성

변환(Coercions)

컬렉션 타입은 지정된 도메인이 같다면 **CAST** 연산자를 이용하여 명시적으로 타입 변환이 가능하다. 아래 표는 명시적 변환이 가능한 컬렉션 타입에 관한 것이다.

명시적 변환이 가능한 컬렉션 타입

	TO			
FROM	SET	MULTISET	LIST	
SET	-	O	O	
MULTISET	O	-	X	
LIST	O	O	-	

SET

설명

SET은 각 원소가 서로 다른 값을 갖는 집합이다. **SET**의 원소는 여러 종류의 데이터 타입을 가질 수 있고, 다른 클래스의 인스턴스를 가질 수도 있다.

예제

```
csql> CREATE TABLE set_tbl ( col 1 set(int, CHAR(1)));
csql> INSERT INTO set_tbl VALUES ({3,3,3,2,2,1,0,'c','c','c','b','b','a'});
csql> INSERT INTO set_tbl VALUES ({NULL});
csql> INSERT INTO set_tbl VALUES ({''});
csql> SELECT * FROM set_tbl;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===
   col 1
=====
{0, 1, 2, 3, 'a', 'b', 'c'}
{NULL}
{' ' }
```

```

csql> SELECT CAST(col_1 AS MULTISSET), CAST(col_1 AS LIST) FROM set_tbl;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

      cast(col_1 as multiset)      cast(col_1 as sequence)
=====
{0, 1, 2, 3, 'a', 'b', 'c'} {0, 1, 2, 3, 'a', 'b', 'c'}
{NULL} {NULL}
{' '} {' '}

csql> INSERT INTO set_tbl VALUES ('');
ERROR: Cannot coerce T' to type set.
ERROR: Incompatible data type on attribute col_1.

```

MULTISET

설명

MULTISET는 중복이 허용되는 집합이다. **MULTISET**의 원소는 여러 종류의 데이터 타입을 가질 수 있고, 다른 클래스의 인스턴스를 가질 수도 있다.

예제

```

csql> CREATE TABLE multiset_tbl ( col_1 multiset(int, CHAR(1)));
csql> INSERT INTO multiset_tbl VALUES ({3,3,3,2,2,1,0,'c','c','c','b','b', 'a'});
csql> SELECT * FROM multiset_tbl;
csql> ;xr

=== <Result of SELECT Command> ===
      col_1
=====
{0, 1, 2, 2, 3, 3, 3, 'a', 'b', 'b', 'c', 'c', 'c'}

csql> SELECT CAST(col_1 AS SET), CAST(col_1 AS LIST) FROM multiset_tbl;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

      cast(col_1 as set)      cast(col_1 as sequence)
=====
{0, 1, 2, 3, 'a', 'b', 'c'} {3, 3, 3, 2, 2, 1, 0, 'c', 'c', 'c', 'b', 'b', 'a'}

```

LIST 또는 SEQUENCE

설명

LIST(=SEQUENCE)는 원소가 입력된 순서가 유지되며, 중복이 허용되는 집합이다. **LIST**의 원소는 여러 종류의 데이터 타입을 가질 수 있고, 다른 클래스의 인스턴스를 가질 수도 있다.

예제

```

csql> CREATE TABLE list_tbl ( col_1 list(int, CHAR(1)));
csql> INSERT INTO list_tbl VALUES ({3,3,3,2,2,1,0,'c','c','c','b','b', 'a'});
csql> SELECT * FROM list_tbl;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===
      col_1
=====
{3, 3, 3, 2, 2, 1, 0, 'c', 'c', 'c', 'b', 'b', 'a'}

```


테이블 정의

CREATE TABLE

테이블 정의

설명

CREATE TABLE 문을 사용하여 새로운 테이블을 생성한다.

구문

```
CREATE {TABLE | CLASS} <table_name>
    [ <subclass definition> ]
    [ ( <column definition> [,<table constraint>]... ) ]
    [ CLASS ATTRIBUTE ( <column_definition_comma_list> ) ]
    [ METHOD <method_definition_comma_list> ]
    [ FILE <method_file_comma_list> ]
    [ INHERIT <resolution_comma_list> ]
    [ REUSE_OID ]

<column_definition> ::=
column_name column_type [ <default or shared> ] | [ <column constraint> ]...

<default or shared> ::=
{SHARED <value_specification> | DEFAULT <value_specification> } |
AUTO_INCREMENT [(seed, increment)]

<column constraint> ::=
NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY <referential definition>

<table constraint> ::=
[ CONSTRAINT <constraint_name> ] UNIQUE [ KEY | INDEX ]( column_name_comma_list ) |
[ { KEY | INDEX } [ <constraint_name> ]( column_name_comma_list ) |
[ PRIMARY KEY ( column_name_comma_list ) ] |
[ <referential_constraint> ]

<referential_constraint> ::=
FOREIGN KEY ( column_name_comma_list ) <referential definition>

<referential definition> ::=
REFERENCES [ referenced_table_name ] ( column_name_comma_list )
[ <referential_triggered_action> ... ]

<referential_triggered action> ::=
{ ON UPDATE <referential action> } |
{ ON DELETE <referential action> } |
{ ON CACHE OBJECT cache_object_column_name }

<referential action> ::=
CASCADE | RESTRICT | NO ACTION | SET NULL

<subclass_definition> ::=
{ UNDER | AS SUBCLASS OF } table_name_comma_list

<method_definition> ::=
[ CLASS ] method_name
[ ( [ argument_type_comma_list ] ) ]
[ result_type ]
[ FUNCTION function_name ]

<resolution> ::=
[ CLASS ] { column_name | method_name } OF superclass_name
[ AS alias ]
```

- *table_name* : 생성할 테이블 이름을 지정한다(최대 255바이트).
- *column_definition* :
- *column_name* : 생성할 컬럼의 이름을 지정한다.
- *column_type* : 컬럼의 데이터 타입을 지정한다.
 - [**SHARED** *value* | **DEFAULT** *value*] : 컬럼의 초기값을 지정한다. (자세한 내용은 [컬럼 정의](#) 참조)
- *column_constraints* : 컬럼의 제약 조건을 지정하며 제약 조건의 종류에는 **NOT NULL**, **UNIQUE**, **PRIMARY KEY**, **FOREIGN KEY**가 있다(자세한 내용은 [제약 조건 정의](#) 참조).

예제

```
CREATE TABLE olympic (
  host_year      INT      NOT NULL PRIMARY KEY,
  host_nation    VARCHAR(40) NOT NULL,
  host_city      VARCHAR(20) NOT NULL,
  opening_date   DATE      NOT NULL,
  closing_date   DATE      NOT NULL,
  mascot        VARCHAR(20) ,
  slogan         VARCHAR(40) ,
  introduction    VARCHAR(1500)
);
```

컬럼 정의

컬럼은 테이블에서 각 열에 해당하는 항목이며, 컬럼은 컬럼 이름과 데이터 타입을 명시하여 정의한다.

```
<column definition> ::=
column name column type [ [ <default or shared> ] | [ <column constraint> ] ]...

<default or shared> ::=
{ SHARED <value_specification> | DEFAULT <value_specification> } |
AUTO_INCREMENT [ (seed, increment) ]

<column constraint> ::=
NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY <referential definition>
```

컬럼 이름

설명

컬럼 이름 작성 원칙은 [식별자](#)를 참고한다. 생성한 컬럼의 이름은 **ALTER TABLE** 문의 **RENAME COLUMN** 절을 사용하여 변경할 수 있다. 자세한 내용은 [RENAME COLUMN 절](#)을 참고한다.

예제

다음은 full_name과 age, 2개의 컬럼을 가지는 manager2 테이블을 생성하는 예제이다.

```
CREATE TABLE manager2 (full_name VARCHAR(40), age INT );
```

주의 사항

- 컬럼 이름의 첫 글자는 반드시 알파벳이어야 하고 최대 길이는 255자이다.
- 컬럼 이름은 테이블 내에서 고유해야 한다.

컬럼의 초기 값 설정(SHARED, DEFAULT)

설명

SHARED, DEFAULT는 컬럼 초기 값과 관련된 속성이다. **SHARED, DEFAULT** 값은 **ALTER TABLE** 문에서 변경할 수 있다.

- **SHARED** : 컬럼 값은 모든 행에서 동일하다. 따라서 **SHARED** 속성은 **UNIQUE** 제약 조건과 동시에 정의할 수 없다. 초기에 설정한 값과 다른 새로운 값을 **INSERT**하면, 해당 컬럼 값은 모든 행에서 새로운 값으로 갱신된다.
- **DEFAULT** : 새로운 행을 삽입할 때 컬럼 값을 지정하지 않으면 **DEFAULT** 속성으로 설정한 값이 저장된다. 테이블 생성 시 컬럼에 **DEFAULT** 값으로 **SYS_TIMESTAMP**를 지정하면, **CREATE TABLE** 시점의 타임스탬프 값이 디폴트 값으로 지정된다. 데이터가 **INSERT**되는 시점의 타임스탬프 값을 입력하려면 데이터 입력 시 **INSERT** 구문의 **VALUES**에 반드시 **SYS_TIMESTAMP**를 입력해야 한다.

예제

```
CREATE TABLE colval tbl
( id INT, name VARCHAR SHARED 'AAA', phone VARCHAR DEFAULT '000-0000');
INSERT INTO colval_tbl(id) VALUES (1),(2);
SELECT * FROM colval_tbl;
;xr

=== <Result of SELECT Command in Line 3> ===

      id  name                phone
=====
      1  'AAA'                '000-0000'
      2  'AAA'                '000-0000'

--updating column values on every row
INSERT INTO colval_tbl(id, name) VALUES (3,'BBB');
INSERT INTO colval_tbl(id) VALUES (4),(5);
SELECT * FROM colval_tbl;
;xr

=== <Result of SELECT Command in Line 3> ===

      id  name                phone
=====
      1  'BBB'                '000-0000'
      2  'BBB'                '000-0000'
      3  'BBB'                '000-0000'
      4  'BBB'                '000-0000'
      5  'BBB'                '000-0000'

5 rows selected.

--changing DEFAULT value in the ALTER TABLE statement
ALTER TABLE colval_tbl CHANGE phone DEFAULT '111-1111'
INSERT INTO colval_tbl(id) VALUES (6);
SELECT * FROM colval_tbl;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  name                phone
=====
      1  'BBB'                '000-0000'
      2  'BBB'                '000-0000'
      3  'BBB'                '000-0000'
      4  'BBB'                '000-0000'
      5  'BBB'                '000-0000'
```

```

6  'BBB'          '111-1111'

6 rows selected.

```

자동 증가 특성(AUTO INCREMENT)

설명

컬럼 값에 자동으로 일련 번호를 부여하기 위해 컬럼에 **AUTO_INCREMENT** 속성을 정의할 수 있다.

SMALLINT, **INTEGER**, **BIGINT**, **NUMERIC(*p*,0)** 도메인에 한정하여 정의할 수 있다.

동일한 컬럼에 **AUTO_INCREMENT** 속성과 **SHARED** 또는 **DEFAULT** 속성을 동시에 정의할 수 없으며, 사용자가 직접 입력한 값과 자동 증가 특성에 의해 입력된 값이 서로 충돌되지 않도록 주의해야 한다.

구문

```
AUTO_INCREMENT [(seed, increment)]
```

- *seed*: 번호가 시작하는 초기값이다. 양의 정수만 허용되며 디폴트 값은 **1**이다.
- *increment*: 행마다 증가되는 증가값이다. 양의 정수만 허용되며 디폴트 값은 **1**이다.

예제

```

CREATE TABLE auto tbl(id INT AUTO INCREMENT, name VARCHAR);
INSERT INTO auto tbl VALUES(NULL, 'AAA'),(NULL, 'BBB'),(NULL, 'CCC');
INSERT INTO auto tbl(name) VALUES ('DDD'),('EEE');
SELECT * FROM auto tbl;
;xr

=== <Result of SELECT Command in Line 4> ===

      id  name
=====
      1  'AAA'
      2  'BBB'
      3  'CCC'
      4  'DDD'
      5  'EEE'

5 rows selected.

```

주의 사항

- 자동 증가 특성만으로는 **UNIQUE** 제약 조건을 가지지 않는다.
- 자동 증가 특성이 정의된 컬럼에 **NULL**을 입력하면 자동 증가된 값이 저장된다.
- 초기값 및 자동 증가 특성에 의해 증가된 최종 값은 해당 도메인에서 허용되는 최소/최대값을 넘을 수 없다.
- 자동 증가 특성은 순환되지 않으므로 타입의 최대값을 넘어갈 경우 오류가 발생하며, 이에 대한 롤백이 일어나지 않는다. 따라서 이와 같은 경우 해당 칼럼을 삭제 후 다시 생성해야 한다.

예를 들어, 아래와 같이 테이블을 생성했다면, A의 최대값은 32767이다. 32767이 넘어가는 경우 에러가 발생하므로, 초기 테이블 생성시에 컬럼 A의 최대값이 해당 타입의 최대값을 넘지 않는다는 것을 감안해야 한다.

```
create table tbl(A smallint auto_increment, B char(5));
```

제약 조건 정의

제약 조건으로 **NOT NULL**, **UNIQUE**, **PRIMARY KEY**, **FOREIGN KEY**를 정의할 수 있다. 또한 제약 조건은 아니지만 **INDEX** 또는 **KEY**를 사용하여 인덱스를 생성할 수도 있다.

```
<column_constraint> ::=
NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY <referential_definition>

<table_constraint> ::=
[ CONSTRAINT <constraint_name> ] UNIQUE [ KEY | INDEX ] ( column_name_comma_list ) |
[ { KEY | INDEX } [ <constraint_name> ] ( column_name_comma_list ) |
[ PRIMARY KEY ( column_name_comma_list ) ] |
[ <referential_constraint> ]

<referential_constraint> ::=
FOREIGN KEY ( column_name_comma_list ) <referential_definition>

<referential_definition> ::=
REFERENCES [ referenced_table_name ] ( column_name_comma_list )
[ <referential_triggered_action> ... ]

<referential_triggered_action> ::=
{ ON UPDATE <referential_action> } |
{ ON DELETE <referential_action> } |
{ ON CACHE OBJECT cache_object_column_name }

<referential_action> ::=
CASCADE | RESTRICT | NO ACTION | SET NULL
```

NOT NULL 제약

설명

NOT NULL 제약 조건이 정의된 컬럼은 반드시 **NULL**이 아닌 값을 가져야 한다. 모든 컬럼에 대해 **NOT NULL** 제약 조건을 정의할 수 있다. **INSERT**, **UPDATE** 구문을 통해 **NOT NULL** 속성 컬럼에 **NULL** 값을 입력하거나 갱신하면 에러가 발생한다.

예제

```
CREATE TABLE const_tbl1(id INT NOT NULL, INDEX i index(id ASC), phone VARCHAR);

CREATE TABLE const_tbl2(id INT NOT NULL PRIMARY KEY, phone VARCHAR);
INSERT INTO const_tbl2 (NULL,'000-0000');
;xr

In line 2, column 25,
ERROR: syntax error, unexpected Null
```

UNIQUE 제약

설명

UNIQUE 제약 조건은 정의된 컬럼이 고유한 값을 갖도록 하는 제약 조건이다. 기존 레코드와 동일한 컬럼 값을 갖는 레코드가 추가되면 에러가 발생한다.

UNIQUE 제약 조건은 단일 컬럼뿐만 아니라 하나 이상의 다중 컬럼에 대해서도 정의가 가능하다. **UNIQUE** 제약 조건이 다중 컬럼에 대해 정의되면 각 컬럼 값에 대해 고유성이 보장되는 것이 아니라, 다중 컬럼 값의 조합에 대해 고유성이 보장된다.

예제

UNIQUE 제약 조건이 다중 컬럼에 대해 정의되면 컬럼 전체 값의 조합에 대해 고유성이 보장된다. 아래의 예와 같이 두 번째 입력 구문은 a 컬럼의 값은 같지만 b 컬럼의 값이 고유하므로 성공한다. 세 번째 입력 구문은 a, b 전체에 대해 첫 번째 값과 동일하므로 오류가 발생한다.

```
--UNIQUE constraint is defined on a single column only
CREATE TABLE const_tbl5(id INT UNIQUE, phone VARCHAR);
INSERT INTO const_tbl5(id) VALUES (NULL), (NULL);
INSERT INTO const_tbl5 VALUES (1, '000-0000');
SELECT * FROM const_tbl5;
;xr

=== <Result of SELECT Command in Line 4> ===

      id  phone
=====
      NULL NULL
      NULL NULL
       1 '000-0000'

3 rows selected.

INSERT INTO const_tbl5 VALUES (1, '111-1111');

ERROR: Operation would have caused one or more unique constraint violations.

--UNIQUE constraint is defined on several columns
CREATE TABLE const_tbl6(id INT, phone VARCHAR, CONSTRAINT UNIQUE(id,phone));
INSERT INTO const_tbl6 VALUES (1,NULL), (2,NULL), (1,'000-0000'), (1,'111-1111');
SELECT * FROM const_tbl6;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  phone
=====
       1 NULL
       2 NULL
       1 '000-0000'
       1 '111-1111'

4 rows selected.
```

PRIMARY KEY 제약

설명

테이블에서 키(key)란 각 행을 고유하게 식별할 수 있는 하나 이상의 컬럼들의 집합을 말한다.

후보키(candidate key)는 테이블 내의 각 행을 고유하게 식별하는 컬럼들의 집합을 의미하며, 사용자는 이러한 후보 키 중 하나를 기본키(primary key)로 정의할 수 있다. 즉, 기본키로 정의된 컬럼 값은 각 행에서 고유하게 식별된다.

예제

```
CREATE TABLE const tbl7(
id INT NOT NULL,
phone VARCHAR,
CONSTRAINT pk_id PRIMARY KEY(id));

CONSTRAINT keyword
CREATE TABLE const tbl8(
id INT NOT NULL PRIMARY KEY,
phone VARCHAR);

--primary key is defined on multiple columns
CREATE TABLE const tbl8 (
host year      INT NOT NULL,
event code     INT NOT NULL,
athlete code   INT NOT NULL,
medal          CHAR(1)  NOT NULL,
score          VARCHAR(20),
unit           VARCHAR(5),
PRIMARY KEY(host_year, event_code, athlete_code, medal)
);
```

FOREIGN KEY 제약

설명

외래키(foreign key)란 참조 관계에 있는 다른 테이블의 기본키를 참조하는 컬럼 또는 컬럼들의 집합을 말한다. 외래키와 참조되는 기본키는 동일한 데이터 타입을 가져야 한다. 외래키가 기본키를 참조함에 따라 연관되는 두 테이블 사이에는 일관성이 유지되는데, 이를 참조 무결성 (referential integrity)이라 한다.

구문

```
[ CONSTRAINT < constraint_name > ]
FOREIGN KEY ( column_name_comma_list )
REFERENCES [ referenced_table_name ] ( column_name_comma_list )
[ <referential_triggered_action> ]

<referential_triggered_action> :
ON UPDATE <referential_action>
[ ON DELETE <referential_action> [ ON CACHE OBJECT cache_object_column_name ] ]

<referential_action> :
CASCADE | RESTRICT | NO ACTION | SET NULL
```

- *constraint_name* : 제약 조건의 이름을 지정한다.
- *column_name* : **FOREIGN KEY** 키워드 뒤에 외래키로 정의하고자 하는 컬럼 이름을 명시한다. 정의되는 외래키의 개수(식별된 속성의 개수)는 제한이 없으나 기본키의 개수와 동일한 개수만큼 정의되어야 한다.
- *referenced_table_name* : 참조되는 테이블의 이름을 지정한다.
- *column_name* : **REFERENCES** 키워드 뒤에 참조되는 기본키 컬럼 이름을 지정한다.
- *referential_triggered_action* : 참조 무결성이 유지되도록 특정 연산에 따라 대응하는 트리거 동작을 정의하는 것이며, **ON UPDATE**, **ON DELETE**, **ON CACHE OBJECT**가 올 수 있다. 각각의 동작은 중복하여 정의 가능하며, 정의 순서는 무관하다.
 - **ON UPDATE** : 외래키가 참조하는 기본키 값을 갱신하려 할 때 수행할 작업을 정의한다. 사용자는 **NO ACTION**, **RESTRICT**, **SET NULL** 중 하나의 옵션을 지정할 수 있으며, 기본은 **RESTRICT**이다.

- **ON DELETE** : 외래키가 참조하는 기본키 값을 삭제하려 할 때 수행할 작업을 정의한다. 사용자는 **NO ACTION, RESTRICT, CASCADE, SET NULL** 중 하나의 옵션을 지정할 수 있으며, 기본은 **RESTRICT**이다.
- **ON CACHE OBJECT** : 객체 지향 모델링에서는 직접 객체 참조(object reference)를 이용한 객체 탐색이 가능한데, 이것을 참조 무결성 외래키와 연계하여 지원하는 것이 **ON CACHE OBJECT** 옵션이다. **ON CACHE OBJECT** 옵션은 외래키 설정에 OID 참조 관계를 부가하고, 설정된 OID는 기본키 테이블에 대한 외래키의 캐시(CACHE) 포인트 개념으로 사용된다. 이렇게 설정된 OID는 시스템 내부적으로만 관리되고, 사용자에게 의해 변경될 수 없다.

ON CACHE OBJECT를 정의하기 위해서는 기본키를 가진 테이블을 도메인으로하는 컬럼이 이미 정의되어 있어야 하며, *cache_object_column_name*에 명시되어야 한다. **ON CACHE OBJECT**로 정의된 속성은 기존 객체 타입의 OID와 동일하게 OID를 사용할 수 있으며, 복제 시 OID 참조 관계를 유지시켜 줄 수 있다.

- *referential_action* : 기본키 값이 삭제 또는 갱신될 때 이를 참조하는 외래키의 값을 유지할 것인지 또는 변경할 것인지 지정할 수 있다.
 - **CASCADE** : 기본키가 삭제되면 외래키도 삭제한다. **ON DELETE** 연산에 대해서만 지원된다.
 - **RESTRICT** : 기본키 값이 삭제되거나 업데이트되지 않도록 제한한다. 삭제 또는 업데이트를 시도하는 트랜잭션은 롤백된다.
 - **SET NULL** : 기본키가 삭제되거나 업데이트되면, 이를 참조하는 외래키 컬럼 값을 **NULL**로 업데이트한다.
 - **NO ACTION** : **RESTRICT** 옵션과 동일하게 동작한다.

예제

```
--creating two tables where one is referencing the other
CREATE TABLE a tbl(
id INT NOT NULL DEFAULT 0 PRIMARY KEY,
phone VARCHAR(10));

CREATE TABLE b tbl(
ID INT NOT NULL,
name VARCHAR(10) NOT NULL,
CONSTRAINT pk id PRIMARY KEY(id),
CONSTRAINT fk id FOREIGN KEY(id) REFERENCES a tbl(id)
ON DELETE CASCADE ON UPDATE RESTRICT);

INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333');
INSERT INTO b_tbl VALUES(1,'George'), (2,'Laura'), (3,'Max');
SELECT a.id, b.id, a.phone, b.name FROM a_tbl a, b_tbl b WHERE a.id=b.id;
;xr

=== <Result of SELECT Command in Line 1> ===

=====
            id            id            phone            name
=====
            1              1            '111-1111'          'George'
            2              2            '222-2222'          'Laura'
            3              3            '333-3333'          'Max'

--when deleting primay key value, it cascades foreign key value
DELETE FROM a tbl WHERE id=3;
;xr

1 rows affected.

Current transaction has been committed.
```

```

1 command(s) successfully processed.

SELECT a.id, b.id, a.phone, b.name FROM a tbl a, b tbl b WHERE a.id=b.id;
;xr

=== <Result of SELECT Command in Line 1> ===

      id      id      phone      name
=====
      1      1      '111-1111'    'George'
      2      2      '222-2222'    'Laura'

--when attempting to update primay key value, it restricts the operation
UPDATE  a_tbl SET id = 10 WHERE phone = '111-1111';
;xr

In the command from line 1,

ERROR: Update/Delete operations are restricted by the foreign key 'fk id'.

0 command(s) successfully processed.

```

주의 사항

- 참조 제한 조건에는 참조 대상이 되는 기본키 테이블의 이름 및 기본키와 일치하는 컬럼명들이 정의된다. 만약, 컬럼명 목록을 지정하지 않을 경우에는 기본키 테이블의 기본키가 원래 지정된 순서대로 지정된다.
- 참조 제한 조건의 기본키의 개수는 외래키의 개수와 동일해야 한다. 참조 제한 조건의 기본키는 동일한 컬럼명이 중복될 수 없다.

KEY 또는 INDEX

설명

KEY와 **INDEX**는 동일하며, 해당 컬럼을 키로 하는 인덱스를 생성한다. 인덱스 이름을 지정할 수 있으며, 생략하면 자동 부여된다.

예제

```

CREATE TABLE const_tbl3(id INT, phone VARCHAR, INDEX(id DESC, phone ASC));

CREATE TABLE const_tbl4(id INT, phone VARCHAR, KEY i_key(id DESC, phone ASC));

```

컬럼 옵션

설명

특정 컬럼에 **UNIQUE** 또는 **INDEX**를 정의할 때, 해당 컬럼 이름 뒤에 **ASC** 또는 **DESC** 옵션을 명시할 수 있다. 이 키워드는 오름차순 또는 내림차순 인덱스 값 저장을 위해 명시된다.

구문

```
column_name [ASC|DESC]
```

예제

```
CREATE TABLE const_tbl(
id VARCHAR,
name VARCHAR,
CONSTRAINT UNIQUE INDEX(id DESC, name ASC)
);

INSERT INTO const_tbl VALUES('1000', 'john'), ('1000','johnny'), ('1000', 'jone');
INSERT INTO const_tbl VALUES('1001', 'johnny'), ('1001','john'), ('1001', 'jone');

SELECT * FROM const_tbl WHERE id > '100';
=====
      id      name
      --      --
      1001     john
      1001  johnny
      1001     jone
      1000     john
      1000  johnny
      1000     jone
```

테이블 옵션(REUSE_OID)

설명

테이블 생성 시 **REUSE_OID** 옵션을 명시하면, 레코드 삭제(**DELETE**)로 인해 삭제된 OID를 새로운 레코드 삽입(**INSERT**) 시 재사용할 수 있다. **REUSE_OID** 옵션을 명시하여 생성된 테이블을 OID 재사용 테이블 또는 참조 불가능(non-referable)한 테이블이라고 한다.

OID(Object Identifier)는 볼륨 번호, 페이지 번호, 슬롯 번호와 같은 물리적 위치 정보로 표현되는 객체 식별자이다. CUBRID는 OID를 이용하여 객체의 참조 관계를 관리하고, 객체 조회, 저장, 삭제를 수행한다. OID를 이용하면 테이블을 참조하지 않고도 힙 파일 내의 해당 오브젝트에 직접 접근할 수 있어 접근성이 향상되지만, 객체가 삭제되더라도 참조 관계를 유지하기 위해 해당 객체의 OID를 보존하기 때문에 **DELETE/INSERT** 연산이 많은 경우 저장 공간 재사용률이 저하되는 문제가 있다.

테이블 생성 시 **REUSE_OID** 옵션을 명시하면, 해당 테이블 내의 데이터 삭제 시 해당 OID가 함께 삭제되며, **INSERT**된 다른 데이터가 해당 OID를 재사용할 수 있다. 단, OID 재사용 테이블을 다른 테이블이 참조할 수 없고, OID 재사용 테이블 내 객체들의 OID 값을 조회할 수 없다.

예제

```
--creating table with REUSE OID option specified
CREATE TABLE reuse_tbl (a INT PRIMARY KEY) REUSE OID;
INSERT INTO reuse_tbl VALUES (1);
INSERT INTO reuse_tbl VALUES (2);
INSERT INTO reuse_tbl VALUES (3);
;xr

3 rows affected.

--an error occurs when column type is a OID reusable table itself
CREATE TABLE tbl_1 ( a reuse_tbl);
;xr

In line 1, column 34,

ERROR: The class 'reuse_tbl' is marked as REUSE OID and is non-referable. Non-referable
classes can't be the domain of an attribute and their instances' OIDs cannot be returned.
```



```
--an error occurs when a table references a OID reusable table
CREATE TABLE tbl 2
(b int, FOREIGN KEY(b) REFERENCES reuse tbl(a) ON CACHE OBJECT oid value);
INSERT INTO tbl 2(b) VALUES(1);
SELECT oid value.a FROM tbl 2;
;xr

ERROR: The class 'reuse tbl' is marked as REUSE OID and is non-referable. Non-referable
classes can't be the domain of an attribute and their instances' OIDs cannot be returned.

--an error occurs when instance methods are called in a OID reusable table
CREATE TABLE tbl 3 UNDER glo REUSE OID;
CALL new_lo import('/home/cubrid/.bashrc') on class tbl_3;
SELECT tbl 3 into x from tbl 3;
;xr

=== <Result of CALL Command in Line 2> ===

      Result
=====
      tbl 3

In line 3, column 21,

ERROR: The class 'tbl 3' is marked as REUSE OID and is non-referable. Non-referable
classes can't be the domain of an attribute and their instances' OIDs cannot be returned.

2 command(s) successfully processed.
```

주의 사항

- 다른 테이블이 OID 재사용 테이블을 참조할 수 없다.
- OID 재사용 테이블에 대해 갱신 가능한(updatable) 뷰를 생성할 수 없다.
- 다른 테이블의 테이블 속성(class attribute) 도메인으로 OID 재사용 테이블을 지정할 수 없다.
- OID 재사용 테이블 객체들의 OID 값을 읽을 수 없다.
- OID 재사용 테이블에서 인스턴스 메소드를 호출할 수 없다. 메소드가 정의된 클래스를 상속받은 서브클래스가 OID 재사용 테이블로 정의되어도 마찬가지로 인스턴스 메소드를 호출할 수 없다. (예: glo 클래스 상속)
- OID 재사용 테이블은 CUBRID 2008 R2.2 버전 이상에서만 지원되며, 하위 호환성을 보장하지 않는다. 즉, 더 낮은 버전의 데이터베이스 서버에서 OID 재사용 테이블이 존재하는 데이터베이스에 접근할 수 없다.
- OID 재사용 테이블은 분할 테이블로 관리될 수 있으며, 복제될 수 있다.

CREATE TABLE LIKE

설명

CREATE TABLE ... LIKE 문을 사용하면, 이미 존재하는 테이블의 스키마와 동일한 스키마를 갖는 테이블을 생성할 수 있다. 기존 테이블에서 정의된 컬럼 속성, 테이블 제약 조건, 인덱스도 그대로 복제된다. 원본 테이블에서 자동 생성된 인덱스의 이름은 새로 생성된 테이블의 이름에 맞게 새로 생성되지만, 사용자에게 의해 지어진 인덱스 이름은 그대로 복제된다. 그러므로 **USING INDEX** 문으로 특정 인덱스를 사용하도록 작성된 질의문이 있다면 주의해야 한다.

CREATE TABLE ... LIKE 문은 스키마만 복제하므로 컬럼 정의문을 작성할 수 없다.

구문

```
CREATE {TABLE | CLASS} <new_table_name> LIKE <old_table_name>
```

- *new_table_name*: 새로 생성할 테이블 이름이다.
- *old_table_name*: 데이터베이스에 이미 존재하는 원본 테이블 이름이다. **CREATE TABLE ... LIKE** 문에서 아래의 테이블은 원본 테이블로 지정될 수 없다.
- 분할 테이블
- **AUTO_INCREMENT** 컬럼이 포함된 테이블
- 상속 또는 메소드를 사용하는 테이블

예제

```
CREATE TABLE a_tbl(  
id INT NOT NULL DEFAULT 0 PRIMARY KEY,  
phone VARCHAR(10));  
INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333');
```

```
--creating an empty table with the same schema as a tbl  
CREATE TABLE new_tbl LIKE a_tbl;  
SELECT * FROM new_tbl;  
;xr
```

=== <Result of SELECT Command in Line 1> ===

There are no results.

0 rows selected.

;schema a_tbl

=== <Help: Schema of a Class> ===

<Class Name>

a_tbl

<Attributes>

id	INTEGER DEFAULT 0 NOT NULL
phone	CHARACTER VARYING(10)

<Constraints>

PRIMARY KEY pk a_tbl id ON a_tbl (id)

Current transaction has been committed.

;schema new_tbl

=== <Help: Schema of a Class> ===

<Class Name>

new_tbl

<Attributes>

id	INTEGER DEFAULT 0 NOT NULL
phone	CHARACTER VARYING(10)

<Constraints>

PRIMARY KEY pk new_tbl id ON new_tbl (id)

Current transaction has been committed.

CREATE TABLE AS SELECT

설명

CREATE TABLE ... AS SELECT 문을 사용하여 **SELECT** 문의 결과 레코드를 포함하는 새로운 테이블을 생성할 수 있다. 새로운 테이블에 대해 컬럼 및 테이블 제약 조건을 정의할 수 있으며, 다음의 규칙을 적용하여 **SELECT** 결과 레코드를 반영한다.

- 새로운 테이블에 컬럼 *col_1*이 정의되고, *select_statement*에 동일한 컬럼 *col_1*이 명시된 경우, **SELECT** 결과 레코드가 새로운 테이블 *col_1* 값으로 저장된다. 컬럼 이름은 같고 컬럼 타입이 다르면 타입 변환을 시도한다.
- 새로운 테이블에 컬럼 *col_1*, *col_2*가 정의되고, *select_statement*의 컬럼 리스트에 *col_1*, *col_2*, *col_3*이 명시되어 모두 포함 관계가 성립하는 경우, 새로 생성되는 테이블에는 *col_1*, *col_2*, *col_3*이 생성되고, **SELECT** 결과 데이터가 모든 컬럼 값으로 저장된다. 컬럼 이름은 같고 컬럼 타입이 다르면 타입 변환을 시도한다.
- 새로운 테이블에 컬럼 *col_1*, *col_2*가 정의되고, *select_statement*의 컬럼 리스트에 *col_1*, *col_3*이 명시되어 포함 관계가 성립하지 않는 경우, 새로 생성되는 테이블에는 *col_1*, *col_2*, *col_3*이 생성되고, *select_statement*에 명시된 컬럼 *col_1*, *col_3*에 대해서만 **SELECT** 결과 데이터가 저장되고, *col_2*에는 NULL이 저장된다.
- *select_statement*의 컬럼 리스트에는 컬럼 별칭(alias)이 포함될 수 있으며, 이 경우 컬럼 별칭이 새로운 테이블 컬럼 이름으로 사용된다. 함수 호출이나 표현식이 사용된 경우 별칭이 없으면 유효하지 않은 컬럼 이름이 생성되므로, 이 경우에는 별칭을 사용하는 것이 좋다.
- **REPLACE** 옵션은 새로운 테이블의 컬럼(*col_1*)에 **UNIQUE** 제약 조건이 정의된 경우에만 유효하다. *select_statement*의 결과 레코드에 중복된 값이 존재하는 경우, **REPLACE** 옵션이 명시되면 컬럼 *col_1*에는 고유한 값이 저장되고, **REPLACE** 옵션이 생략되면 **UNIQUE** 제약 조건에 위배되므로 에러 메시지가 출력된다.

구문

```
CREATE {TABLE | CLASS} <table_name>
    [( <column_definition> [, <table_constraint>]... )]
    [REPLACE] AS <select_statement>
```

- *table_name*: 새로 생성할 테이블 이름이다.
- *column_definition*, *table_constraint*: 컬럼을 정의한다. 생략하면 **SELECT** 문의 컬럼 스키마가 복제된다. **SELECT** 문의 컬럼 제약 조건이나 **AUTO_INCREMENT** 속성은 복제되지 않는다.
- *table_constraint*: 테이블 제약 조건을 정의한다.
- *select_statement*: 데이터베이스에 이미 존재하는 원본 테이블을 대상으로 하는 **SELECT** 문이다.

예제

```
CREATE TABLE a_tbl(
id INT NOT NULL DEFAULT 0 PRIMARY KEY,
phone VARCHAR(10));
INSERT INTO a_tbl VALUES (1, '111-1111'), (2, '222-2222'), (3, '333-3333');
```

```
--creating a table without column definition
CREATE TABLE new tbl1 AS SELECT * FROM a tbl;
SELECT * FROM new tbl1;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  phone
=====
      1  '111-1111'
      2  '222-2222'
      3  '333-3333'

3 rows selected.

--all of column values are replicated from a tbl
CREATE TABLE new tbl2
(id INT NOT NULL AUTO INCREMENT PRIMARY KEY, phone VARCHAR) AS SELECT * FROM a tbl;
SELECT * FROM new tbl2;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  phone
=====
      1  '111-1111'
      2  '222-2222'
      3  '333-3333'

3 rows selected.

--some of column values are replicated from a_tbl and the rest is NULL
CREATE TABLE new tbl3
(id INT, name VARCHAR) AS SELECT id, phone FROM a tbl;
SELECT * FROM new tbl3;
;xr

=== <Result of SELECT Command in Line 1> ===

      name                id  phone
=====
      NULL                1  '111-1111'
      NULL                2  '222-2222'
      NULL                3  '333-3333'

3 rows selected.

--column alias in the select statement should be used in the column definition
CREATE TABLE new tbl4
(id1 int, id2 int)AS SELECT t1.id id1, t2.id id2 FROM new tbl1 t1, new tbl2 t2;
SELECT * FROM new tbl4;
;xr

=== <Result of SELECT Command in Line 1> ===

      id1    id2
=====
      1      1
      1      2
      1      3
      2      1
      2      2
      2      3
      3      1
      3      2
      3      3

9 rows selected.

--REPLACE is used on the UNIQUE column
```

```
CREATE TABLE new tbl5(id1 int UNIQUE) REPLACE AS SELECT * FROM new tbl4;
SELECT * FROM new tbl5;
;xr
```

=== <Result of SELECT Command in Line 1> ===

id1	id2
1	3
2	3
3	3

3 rows selected.

ALTER TABLE

개요

설명

ALTER 구문을 이용하여 테이블의 구조를 변경할 수 있다. 대상 테이블에 컬럼 추가/삭제, 인덱스 생성/삭제, 기존 컬럼의 타입 변경, 테이블 이름 변경, 컬럼 이름 변경 등을 수행하거나 테이블 제약 조건을 변경한다.

TABLE은 **CLASS**와 동의어이고, **VIEW**는 **VCLASS**와 동의어이다. **COLUMN**은 **ATTRIBUTE**와 동의어이다.

구문

```
ALTER [ <class type> ] <table name> <alter clause> ;

<class_type> ::= TABLE | CLASS | VCLASS | VIEW

<alter_clause> ::= ADD <alter_add> [ INHERIT <resolution_comma_list> ] |
                  ADD { KEY | INDEX } [index_name] (<index_col_name>) |
                  ALTER [ COLUMN ] column_name SET DEFAULT <value_specification> |
                  DROP <alter_drop> [ INHERIT <resolution_comma_list> ] |
                  DROP { KEY | INDEX } index_name |
                  DROP FOREIGN KEY constraint_name |
                  DROP PRIMARY KEY |
                  RENAME <alter_rename> [ INHERIT <resolution_comma_list> ] |
                  CHANGE <alter_change> |
                  INHERIT <resolution_comma_list>

<alter_add> ::= [ ATTRIBUTE | COLUMN ] [(|<class_element_comma_list>|)] [ FIRST | AFTER
old_column_name ] |
              CLASS ATTRIBUTE <column_definition_comma_list> |
              CONSTRAINT <constraint_name> <column_constraint> ( column_name ) |
              FILE <file_name_comma_list> |
              METHOD <method_definition_comma_list> |
              QUERY <select_statement> |
              SUPERCLASS <class_name_comma_list>

<alter_change> ::= FILE <file_path_name> AS <file_path_name> |
                  METHOD <method_definition_comma_list> |
                  QUERY [ <unsigned_integer_literal> ] <select_statement> |
                  <column_name> DEFAULT <value_specification>

<alter_drop> ::= [ ATTRIBUTE | COLUMN | METHOD ]
               <column_name_comma_list> |
               FILE <file_name_comma_list> |
               QUERY [ <unsigned_integer_literal> ] |
               SUPERCLASS <class_name_comma_list> |
               CONSTRAINT <constraint_name>

<alter_rename> ::= [ ATTRIBUTE | COLUMN | METHOD ]
                <old_column_name> AS <new_column_name> |
                FUNCTION OF <column_name> AS <function_name>
```

```

FILE <file path name> AS <file path name>

<resolution> ::= { column name | method name } OF <superclass name>
[ AS alias ]

<class element> ::= <column definition> | <table constraint>

<column_constraint> ::= UNIQUE [ KEY ] | PRIMARY KEY | FOREIGN KEY

<index_col_name> ::=
column_name [(length)] [ ASC | DESC ]

```

주의 사항

테이블의 소유자, **DBA**, **DBA**의 멤버만이 테이블 스키마를 변경할 수 있으며, 그 밖의 사용자는 소유자나 **DBA**로부터 이름을 변경할 수 있는 권한을 받아야 한다(권한 관련 사항은 [권한 부여](#) 참조).

ADD COLUMN 절

설명

ADD COLUMN 절을 사용하여 새로운 컬럼을 추가할 수 있다. **FIRST** 또는 **AFTER** 키워드를 사용하여 새로 추가할 컬럼의 위치를 지정할 수 있다.

구문

```

ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name
ADD [ COLUMN | ATTRIBUTE ] [(|<column definition>|)] [ FIRST | AFTER old_column_name ]

column_definition ::=
column_name column_type
{ [ NOT NULL | NULL ] |
  [ { SHARED <value specification> | DEFAULT <value specification> }
    | AUTO_INCREMENT [(seed, increment)] ] |
  [ UNIQUE [ KEY ] |
    [ PRIMARY KEY | FOREIGN KEY REFERENCES
      [ referenced_table_name ]( column_name_comma_list )
      [ <referential triggered action> ... ]
    ]
  ] } ...

<referential_triggered_action> ::=
{ ON UPDATE <referential_action> } |
{ ON DELETE <referential_action> } |
{ ON CACHE OBJECT cache_object_column_name }

<referential_action> ::=
CASCADE | RESTRICT | NO ACTION | SET NULL

```

- *table_name*: 컬럼을 추가할 테이블의 이름을 지정한다.
- *column_definition*: 새로 추가할 컬럼의 이름, 데이터 타입, 제약 조건을 정의한다.
- **AFTER** *old_column_name*: 새로 추가할 컬럼 앞에 위치하는 기존 컬럼 이름을 명시한다.

예제

```

CREATE TABLE a_tbl;
ALTER TABLE a_tbl ADD COLUMN age INT DEFAULT 0 NOT NULL;
INSERT INTO a_tbl(age) VALUES(20),(30),(40);
ALTER TABLE a_tbl ADD COLUMN name VARCHAR FIRST;
ALTER TABLE a_tbl ADD COLUMN id INT NOT NULL AUTO_INCREMENT UNIQUE;
ALTER TABLE a_tbl ADD COLUMN phone VARCHAR(13) DEFAULT '000-0000-0000' AFTER name;

```

```

SELECT * FROM a_tbl;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

  name                phone                age                id
=====
  NULL                '000-0000-0000'                20                1
  NULL                '000-0000-0000'                30                2
  NULL                '000-0000-0000'                40                3

--adding multiple columns
ALTER TABLE a_tbl ADD COLUMN (age1 int, age2 int, age3 int);
;xr

1 command(s) successfully processed.

```

ADD CONSTRAINT 절

설명

ADD CONSTRAINT 절을 사용하여 새로운 제약 조건을 추가할 수 있다.

구문

```

ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name
ADD CONSTRAINT < constraint name > column constraint ( index column name )

column constraint ::=
UNIQUE [ KEY ] |
PRIMARY KEY |
FOREIGN KEY REFERENCES [referenced table name]( column name comma list )
[ <referential triggered action> ... ]

<referential triggered action> ::=
{ ON UPDATE <referential_action> } |
{ ON DELETE <referential_action> } |
{ ON CACHE OBJECT cache_object_column_name }

<referential_action> ::=
CASCADE | RESTRICT | NO ACTION | SET NULL

```

- *table_name*: 제약 조건을 추가할 테이블의 이름을 지정한다.
- *constraint_name*: 새로 추가할 제약 조건의 이름을 지정할 수 있으며, 생략할 수 있다. 생략하면 자동으로 부여된다.
- *column_constraint*: 지정된 컬럼에 대해 제약 조건을 정의한다. 제약 조건에 대한 자세한 설명은 [제약 조건 정의](#)를 참고한다.

예제

```

ALTER TABLE a_tbl ADD CONSTRAINT PRIMARY KEY(id);
ALTER TABLE a_tbl ADD CONSTRAINT UNIQUE u_key1(id);

```

ADD INDEX 절

설명

ADD INDEX 절은 특정 컬럼에 대해 인덱스 속성을 추가로 정의할 수 있다.

구문

```
ALTER [ TABLE | CLASS ] table_name ADD { KEY | INDEX } [ index_name ] (<index_col_name>)  
  
<index_col_name> ::=  
column_name [(length)] [ ASC | DESC ]
```

- *table_name*: 변경하고자 하는 테이블의 이름을 지정한다.
- *index_name*: 인덱스의 이름을 지정한다. 생략하면 자동으로 부여된다.
- *index_col_name*: 인덱스를 정의할 대상 컬럼을 지정하며, 이때 컬럼 옵션으로 인덱스 키의 *prefix_length*와 **ASC** 또는 **DESC**을 함께 지정할 수 있다.

예제

```
ALTER TABLE a_tbl ADD INDEX (age ASC), ADD INDEX(phone DESC);  
;schema a_tbl  
  
=== <Help: Schema of a Class> ===  
  
<Class Name>  
  
a_tbl  
  
<Attributes>  
  
name CHARACTER VARYING(1073741823) DEFAULT ''  
phone CHARACTER VARYING(13) DEFAULT '111-1111'  
age INTEGER  
id INTEGER AUTO_INCREMENT NOT NULL  
  
<Constraints>  
  
UNIQUE u a_tbl id ON a_tbl (id)  
INDEX i a_tbl age ON a_tbl (age)  
INDEX i a_tbl phone d ON a_tbl (phone DESC)  
  
Current transaction has been committed.
```

ALTER COLUMN ... SET DEFAULT 절

설명

ALTER COLUMN ... SET DEFAULT 절은 디폴트 값이 없는 컬럼에 디폴트 값을 지정하거나 기존의 디폴트 값을 변경할 수 있다. [CHANGE 절](#)을 이용하면, 단일 구문으로 여러 개의 컬럼의 디폴트 값을 변경할 수 있다.

구문

```
ALTER [ TABLE | CLASS ] table_name ALTER [ COLUMN ] column_name SET DEFAULT value
```

- *table_name*: 디폴트 값을 변경할 컬럼이 속한 테이블의 이름을 지정한다.
- *column_name*: 새로운 디폴트 값을 적용할 컬럼의 이름을 지정한다.
- *value*: 새로운 디폴트 값을 지정한다.

예제

```
;schema a_tbl  
  
=== <Help: Schema of a Class> ===
```



```

<Class Name>

a_tbl

<Attributes>

name          CHARACTER VARYING(1073741823)
phone         CHARACTER VARYING(13) DEFAULT '000-0000-0000'
age           INTEGER
id            INTEGER AUTO INCREMENT NOT NULL

<Constraints>

UNIQUE u_a_tbl_id ON a_tbl (id)

Current transaction has been committed.

ALTER TABLE a_tbl ALTER COLUMN name SET DEFAULT '';
ALTER TABLE a_tbl ALTER COLUMN phone SET DEFAULT '111-1111';

;schema a_tbl

=== <Help: Schema of a Class> ===

<Class Name>

a_tbl

<Attributes>

name          CHARACTER VARYING(1073741823) DEFAULT ''
phone         CHARACTER VARYING(13) DEFAULT '111-1111'
age           INTEGER
id            INTEGER AUTO INCREMENT NOT NULL

<Constraints>

UNIQUE u_a_tbl_id ON a_tbl (id)

```

CHANGE 절

설명

CHANGE 절에서 **DEFAULT** 키워드를 사용하여 디폴트 값이 없는 컬럼에 새로운 디폴트 값을 지정하거나 기존의 디폴트 값을 변경할 수 있다.

구문

```
ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name CHANGE [ CLASS ] column_name DEFAULT
value [, column_name DEFAULT value, ...]
```

- *table_name*: 디폴트 값을 변경할 컬럼이 속한 테이블의 이름을 지정한다.
- *column_name*: 새로운 디폴트 값을 적용할 컬럼의 이름을 지정한다.
- *value*: 새로운 디폴트 값을 지정한다.

예제

```

ALTER TABLE a_tbl CHANGE name DEFAULT 'abc', phone DEFAULT '000-0000';

;schema a_tbl

=== <Help: Schema of a Class> ===

```

```

<Class Name>

a tbl

<Attributes>

name          CHARACTER VARYING(1073741823) DEFAULT 'abc'
phone         CHARACTER VARYING(13)  DEFAULT '000-0000'
age           INTEGER
id            INTEGER AUTO_INCREMENT NOT NULL

<Constraints>

UNIQUE u a tbl id ON a tbl (id)
INDEX i_a_tbl_age ON a_tbl (age)
INDEX i_a_tbl_phone_d ON a_tbl (phone DESC)

```

RENAME COLUMN 절

설명

RENAME COLUMN 절을 사용하여 이미 존재하는 컬럼의 이름을 변경할 수 있다.

구문

```

ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name
RENAME [ COLUMN | ATTRIBUTE ] old_column_name { AS | TO } new_column_name

```

- *table_name*: 변경할 컬럼이 존재하는 테이블 이름을 지정한다.
- *old_column_name*: 테이블에 이미 존재하는 컬럼 이름을 지정한다.
- *new_column_name*: 새로운 컬럼 이름을 **AS** 키워드 뒤에 명시한다.

예제

```
ALTER TABLE a_tbl RENAME COLUMN name AS name1;
```

DROP COLUMN 절

설명

DROP COLUMN 절을 사용하여 테이블에 존재하는 컬럼을 삭제할 수 있다. 삭제하고자 하는 컬럼들을 쉼표(,)로 구분하여 여러 개의 컬럼을 한 번에 삭제할 수 있다.

구문

```

ALTER [ TABLE | CLASS | VCLASS | VIEW ] table_name
DROP [ COLUMN | ATTRIBUTE ] column_name, ...

```

- *table_name*: 삭제할 컬럼이 존재하는 테이블의 이름을 명시한다.
- *column_name*: 삭제할 컬럼의 이름을 명시한다. 쉼표로 구분하여 여러 개의 컬럼을 지정할 수 있다.

예제

```
ALTER TABLE a_tbl DROP COLUMN age1, age2, age3;
```

DROP CONSTRAINT 절

설명

DROP CONSTRAINT 절을 사용하여, 테이블에 이미 정의된 **UNIQUE**, **PRIMARY KEY**, **FOREIGN KEY** 제약 조건을 삭제할 수 있다. 삭제할 제약 조건 이름을 지정해야 하며, 이는 CSQL 명령어(**schema table_name**)를 사용하여 확인할 수 있다.

구문

```
ALTER [ TABLE | CLASS ] table_name  
DROP CONSTRAINT constraint_name
```

- *table_name* : 제약 조건을 삭제할 테이블의 이름을 지정한다.
- *constraint_name* : 삭제할 제약 조건의 이름을 지정한다.

예제

```
ALTER TABLE a_tbl DROP CONSTRAINT pk_a_tbl_id;  
ALTER TABLE a_tbl DROP CONSTRAINT fk_a_tbl_id;  
ALTER TABLE a_tbl DROP CONSTRAINT u_a_tbl_id;
```

DROP INDEX 절

설명

DROP INDEX 절을 사용하여 인덱스를 삭제할 수 있다.

구문

```
ALTER [ TABLE | CLASS ] table_name DROP INDEX index_name
```

- *table_name* : 제약 조건을 삭제할 테이블의 이름을 지정한다.
- *index_name* : 삭제할 인덱스의 이름을 지정한다.

예제

```
ALTER TABLE a_tbl DROP INDEX i_a_tbl_age;
```

DROP PRIMARY KEY 절

설명

DROP PRIMARY KEY 절을 사용하여 테이블에 정의된 기본키 제약 조건을 삭제할 수 있다. 하나의 테이블에는 하나의 기본키만 정의될 수 있으므로 기본키 제약 조건 이름을 지정하지 않아도 된다.

구문

```
ALTER [ TABLE | CLASS ] table_name DROP INDEX PRIMARY KEY
```

- *table_name* : 기본키 제약 조건을 삭제할 테이블의 이름을 지정한다.

예제

```
ALTER TABLE a_tbl DROP PRIMARY KEY;
```

DROP FOREIGN KEY 절

설명

DROP FOREIGN KEY 절을 사용하여 테이블에 정의된 외래키 제약 조건을 모두 삭제할 수 있다.

구문

```
ALTER [ TABLE | CLASS ] table_name DROP FOREIGN KEY constraint_name
```

- *table_name*: 제약 조건을 삭제할 테이블의 이름을 지정한다.
- *constraint_name*: 삭제할 외래키 제약 조건의 이름을 지정한다.

예제

```
ALTER TABLE a_tbl DROP FOREIGN KEY fk_a_tbl_id;
```

DROP TABLE

설명

DROP 구문을 이용하여 기존의 테이블을 삭제할 수 있다. 하나의 **DROP** 구문으로 여러 개의 테이블을 삭제할 수 있으며 테이블이 삭제되면 포함된 행도 모두 삭제된다.

구문

```
DROP [ TABLE | CLASS | VIEW | VCLASS ] <table_specification_comma_list>
```

```
<table_specification> ::=  
<single_table_spec> | ( <single_table_spec_comma_list> )
```

```
<single_table_spec> ::=  
|[ ONLY ] table_name  
| ALL table_name [ ( EXCEPT table_name, ... ) ]
```

- *table_name*: 삭제할 테이블의 이름을 지정한다. 쉼표로 구분하여 여러 개의 테이블을 한 번에 삭제할 수 있다.
- **ONLY** 키워드 뒤에 슈퍼클래스 이름이 명시되면, 해당 슈퍼클래스만 삭제하고 이를 상속받는 서브클래스는 삭제하지 않는다.
- **ALL** 키워드 뒤에 슈퍼클래스 이름이 지정되면, 해당 슈퍼클래스 및 이를 상속받는 서브클래스를 모두 삭제한다. **EXCEPT** 키워드 뒤에 삭제하지 않을 서브클래스 리스트를 명시할 수 있다.

예제

다음은 history 테이블을 삭제하는 예제이다.

```
DROP TABLE history ;
```

RENAME TABLE

설명

RENAME TABLE 구문을 사용하여 테이블 이름을 변경할 수 있으며, 여러 개의 테이블 이름을 변경하는 경우 테이블 이름 리스트를 명시할 수 있다.

구문

```
RENAME [ TABLE | CLASS | VIEW | VCLASS ] old_table_name { AS | TO } new_table_name [,  
old_table_name { AS | TO } new_table_name, ... ]
```

- *old_table_name*: 변경할 테이블의 이름을 지정한다.
- *new_table_name*: 테이블의 새로운 이름을 지정한다.

예제

```
RENAME TABLE a_tbl AS aa_tbl;  
RENAME TABLE a_tbl TO aa_tbl, b_tbl TO bb_tbl;
```

주의 사항

테이블의 소유자, **DBA**, **DBA**의 멤버만이 테이블의 이름을 변경할 수 있으며, 그 밖의 사용자는 소유자나 **DBA**로부터 이름을 변경할 수 있는 권한을 받아야 한다(권한 관련 사항은 [권한 부여](#) 참조).

인덱스 정의

CREATE INDEX

설명

CREATE INDEX 구문을 이용하여 지정한 테이블에 인덱스를 생성한다.

구문

```
CREATE [ REVERSE ] [ UNIQUE ] INDEX [ index_name ]  
ON table_name ( column_name[(prefix length)] [ASC | DESC] [ {, column_name[(prefix length)]  
[ASC | DESC]} ... ] ) [ ; ]
```

- **REVERSE** : 역순으로 인덱스를 생성한다. 역순 인덱스는 내림차순 정렬 연산 속도를 향상시킨다.
- **UNIQUE** : 유일한 값을 갖는 고유 인덱스를 생성한다.
- *index_name* : 생성하려는 인덱스의 이름을 명시한다. 인덱스 이름은 테이블 안에서 고유한 값이어야 한다. 생략하면 자동으로 부여된다.
- *prefix_length* : 문자 또는 비트 스트링 타입의 컬럼에 인덱스를 설정하는 경우, 컬럼 값의 앞 부분 일부를 prefix로 지정하여 인덱스를 생성하기 위하여 컬럼 이름 뒤 괄호 안에 바이트 단위로 prefix 길이를 지정할 수 있다. 단, *prefix_length*는 다중 컬럼 인덱스 및 **UNIQUE** 인덱스에는 지정할 수 없다.
- *table_name* : 인덱스를 생성할 테이블의 이름을 명시한다.
- *column_name* : 인덱스를 적용할 컬럼의 이름을 명시한다. 다중 컬럼 인덱스를 생성할 경우 둘 이상의 컬럼 이름을 명시한다.
- **ASC | DESC** : 컬럼의 정렬 방향을 설정한다. **REVERSE** 인덱스인 경우 **ASC**는 무시되고 **DESC**로 처리된다.

예제 1

다음은 단일 컬럼 인덱스를 생성하는 예제이다. 스트링 타입으로 정의한 `nation_code` 컬럼에 대해서는 1바이트 길이만큼 prefix를 지정하여 인덱스를 생성한다.

```
CREATE INDEX ON game(nation_code(1));  
CREATE INDEX game_date_idx ON game(game_date);
```

예제 2

다음은 역순 인덱스를 생성하는 예제이다.

```
CREATE REVERSE INDEX gold_index ON participant(gold);
```

예제 3

다음은 다중 컬럼 인덱스를 생성하는 예제이다.

```
CREATE INDEX name_nation_idx ON athlete(name, nation_code);
```

ALTER INDEX

설명

ALTER INDEX 문을 사용하여 인덱스를 재생성 한다. (즉, 인덱스를 삭제하고 다시 생성한다.) 재생성 할 인덱스를 지정하는 방법에는 두 가지가 있다.

- 인덱스 이름으로 지정하는 방법
- 인덱스가 지정된 테이블 이름과 컬럼 이름으로 지정하는 방법

구문

```
ALTER [ REVERSE ] [ UNIQUE ] INDEX index_name
[ON { ONLY } table_name ( column_name [ {, column_name } ...] ) ] REBUILD [ ; ]

ALTER [ REVERSE ] [ UNIQUE ] INDEX
ON { ONLY } table_name ( column_name [ {, column_name } ...] ) REBUILD [ ; ]
```

- **REVERSE** : 역순으로 인덱스를 생성한다. 역순 인덱스는 내림차순 정렬 연산 속도를 향상시킨다.
- **UNIQUE** : 유일한 값을 갖는 고유 인덱스를 생성한다.
- *index_name* : 변경하려는 인덱스의 이름을 명시한다. 인덱스 이름은 테이블 안에서 고유한 값이어야 한다.
- *table_name* : 인덱스를 생성할 테이블의 이름을 명시한다.
- *column_name* : 인덱스를 적용할 컬럼의 이름을 명시한다. 다중 컬럼 인덱스를 생성할 경우 둘 이상의 컬럼 이름을 명시한다.

예제

다음은 인덱스를 재생성하는 여러 가지 방법을 보여주는 예제이다.

```
ALTER INDEX i_game medal ON game(medal) REBUILD;
ALTER INDEX game_date_idx REBUILD;
```

DROP INDEX

설명

DROP INDEX 문을 사용하여 인덱스를 삭제할 수 있다. 삭제할 인덱스를 지정하는 방법에는 다음과 같은 두 가지 방법이 있다.

- 인덱스 이름으로 지정하는 방법
- 인덱스가 지정된 테이블 이름과 컬럼 이름으로 지정하는 방법

구문

```
DROP [ REVERSE ] [ UNIQUE ] INDEX index_name
[ON table_name ( column name [ {, column name } ...] ) ] [ ; ]

DROP [ REVERSE ] [ UNIQUE ] INDEX
ON table_name ( column_name [ {, column_name } ...] ) [ ; ]
```

- **REVERSE** : 삭제하려는 인덱스가 역순 인덱스임을 지정한다.

- **UNIQUE** : 삭제하여는 인덱스가 고유 인덱스임을 지정한다.
- *index_name* : 삭제할 인덱스의 이름을 지정한다.
- *table_name* : 삭제할 인덱스가 지정된 테이블 이름을 지정한다.
- *column_name* : 삭제할 인덱스가 지정된 컬럼 이름을 지정한다.

예제

다음은 인덱스를 삭제하는 여러 가지 방법을 보여주는 예제이다.

```
DROP INDEX ON game(medal);  
DROP INDEX game date idx;  
DROP REVERSE INDEX gold index ON participant(gold);  
DROP INDEX name_nation_idx ON athlete(name, nation_code);
```


뷰 정의

CREATE VIEW

개요

설명

뷰(가상 테이블)는 물리적으로 존재하지 않는 가상의 테이블이며, 기존의 테이블이나 뷰에 대한 질의문을 이용하여 뷰를 생성할 수 있다. **VIEW**와 **VCLASS**는 동의어로 사용된다.

CREATE VIEW 문을 이용하여 뷰를 생성한다.

구문

```
CREATE [OR REPLACE] {VIEW | VCLASS} <view_name>
[ <subclass_definition> ]
[ ( <view_column_def_comma_list> ) ]
[ CLASS ATTRIBUTE
  ( <column_definition_comma_list> ) ]
[ METHOD <method_definition_comma_list> ]
[ FILE <method_file_comma_list> ]
[ INHERIT <resolution_comma_list> ]
[ AS <select_statement> ]
[ WITH CHECK OPTION ]

<view_column_definition> ::= <column_definition> | <column_name>

<column_definition> :
column_name column_type [ <default_or_shared> ] [ <column_constraint_list> ]

<default or shared> :
{SHARED [ <value_specification> ] | DEFAULT <value_specification> } |
AUTO_INCREMENT [ (seed, increment) ]

<column_constraint> :
NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY REFERENCES...

<subclass_definition> :
{ UNDER | AS SUBCLASS OF } table_name comma_list

<method_definition> :
[ CLASS ] method_name
[ ( [ argument_type_comma_list ] ) ]
[ result_type ]
[ FUNCTION function_name ]

<resolution> :
[ CLASS ] { column_name | method_name } OF superclass_name
[ AS alias ]
```

- **OR REPLACE** : **CREATE** 뒤에 **OR REPLACE** 키워드가 명시되면, *view_name*이 기존의 뷰와 이름이 중복되더라도 에러를 출력하지 않고 기존의 뷰를 새로운 뷰로 대체한다.
- *view_name* : 생성하려는 뷰의 이름을 지정한다. 뷰의 이름은 데이터베이스 내에서 고유해야 한다.
- *view_column_definition*
 - *column_name* : 뷰의 컬럼을 정의한다.
- *column_type* : 컬럼의 데이터 타입을 정의한다.

AS select_statement : 유효한 **SELECT** 문이 명시되어야 한다. 이를 기반으로 뷰가 생성된다.

WITH CHECK OPTION : 이 옵션이 명시되면 *select_statement* 내 **WHERE** 절에 명시된 조건식을 만족하는 경우에만 업데이트 또는 삽입이 가능하다. 조건식을 위반하는 가상 테이블에 대한 갱신을 허용하지 않기 위해서 사용한다.

예제

```
CREATE TABLE a_tbl(
id INT NOT NULL,
phone VARCHAR(10));
INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333'), (4, NULL), (5,
NULL);

--creating a new view based on AS select statement from a tbl
CREATE VIEW b view AS SELECT * FROM a_tbl WHERE phone IS NOT NULL WITH CHECK OPTION;
SELECT * FROM b view;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  phone
=====
      1  '111-1111'
      2  '222-2222'
      3  '333-3333'

3 rows selected.

--WITH CHECK OPTION doesn't allow to update column value which violates WHERE clause
UPDATE b view SET phone=NULL;
;xr

In line 1, column 72,
ERROR: Check option exception on view b_view.

--creating view which name is as same as existing view name
CREATE OR REPLACE VIEW b_view AS SELECT * FROM a_tbl ORDER BY id DESC;
;xr

Current transaction has been committed.

1 command(s) successfully processed.

--the existing view has been replaced as a new view by OR REPLACE keyword
SELECT * FROM b_view;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  phone
=====
      5  NULL
      4  NULL
      3  '333-3333'
      2  '222-2222'
      1  '111-1111'

5 rows selected.
```

업데이트 가능한 VIEW의 생성 조건

뷰의 데이터를 업데이트하려면 뷰가 업데이트할 수 있도록 정의되어야 한다.

다음의 조건을 만족한다면 해당 뷰는 업데이트할 수 있다.

- **FROM** 절은 반드시 하나의 테이블 또는 업데이트 가능한 뷰만 포함해야 한다. 단, **FROM** (*class_x*, *class_y*)과 같이 괄호에 포함된 두 테이블은 하나의 테이블을 표현하므로 업데이트할 수 있다.
- **DISTINCT**, **UNIQUE** 구문을 포함하지 않는다.
- **GROUP BY... HAVING** 구문을 포함하지 않는다.
- **SUM()**, **AVG()**와 같은 집계 함수를 포함하지 않는다.
- **UNION**이 아닌 **UNION ALL**을 사용하여 업데이트 가능한 질의만으로 질의를 구성한 경우 업데이트할 수 있다. 단, 테이블은 **UNION ALL**을 구성하는 질의 중 어느 한 질의에만 존재해야 한다.
- **UNION ALL** 구문을 사용하여 생성된 뷰에 행을 입력하는 경우, 행이 입력될 테이블은 시스템이 결정한다. 행이 입력될 테이블을 사용자가 제어하는 것은 불가능하므로 사용자가 제어하기 원한다면 테이블에 직접 입력하거나 입력을 위한 별도의 뷰를 생성해야 한다.

뷰가 위의 규칙을 모두 충족하여 업데이트할 수 있어도, 해당 뷰의 각 컬럼이 업데이트 불가능할 수 있다. 다음과 같은 컬럼은 업데이트할 수 없다.

- 경로 표현식
- 산술 연산자가 포함된 숫자 타입의 컬럼

뷰에 정의된 컬럼이 업데이트 가능하더라도 **FROM** 구문에 포함된 테이블에 대해 업데이트를 위한 적절한 권한이 있어야 뷰를 수정할 수 있다. 또한 뷰에 대한 접근 권한이 있어야 한다. 뷰에 접근 권한을 부여하는 방법은 테이블에 접근 권한을 부여하는 방식과 동일하다. 권한 부여에 대한 자세한 내용은 [권한 부여](#)를 참조한다.

ALTER VIEW

ADD QUERY 절

설명

ALTER VIEW 문에 **ADD QUERY** 절을 사용하여 뷰의 질의 명세부에 질의를 추가할 수 있다. 뷰 생성 시 정의된 질의문에는 1이 부여되고, **ADD QUERY** 절에서 추가한 질의문에는 2가 부여된다.

구문

```
ALTER [ VIEW | VCLASS ] view_name
ADD QUERY select_statement
[ INHERIT resolution [ (, resolution ) ] ]

resolution :
{ column_name | method_name } OF superclass_name [ AS alias ]
```

- *view_name* : 질의를 추가할 뷰의 이름 명시한다.
- *select_statement* : 추가할 질의를 명시한다.

예제

```
SELECT * FROM b_view;
```

```

;xr

=== <Result of SELECT Command in Line 1> ===

      id  phone
=====
      1  '111-1111'
      2  '222-2222'
      3  '333-3333'
      4  NULL
      5  NULL

ALTER VIEW b view ADD QUERY SELECT * FROM a tbl WHERE id IN (1,2);
SELECT * FROM b_view;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

      id  phone
=====
      1  '111-1111'
      2  '222-2222'
      3  '333-3333'
      4  NULL
      5  NULL
      1  '111-1111'
      2  '222-2222'

7 rows selected.

```

AS SELECT 절

설명

ALTER VIEW 문에 **AS SELECT** 절을 사용하여 가상 테이블에 정의된 **SELECT** 질의를 변경할 수 있다. 이는 **CREATE OR REPLACE** 문과 유사하게 동작한다. **ALTER VIEW** 문의 **CHANGE QUERY** 절에 질의 번호 1을 명시하여 질의를 변경할 수도 있다.

구문

```
ALTER [ VIEW | VCLASS ] view_name AS select_statement
```

- *view_name*: 변경할 가상 테이블의 이름을 명시한다.
- *select_statement*: 가상 테이블 생성 시 정의된 **SELECT** 문을 대체할 새로운 질의문을 명시한다.

예제

```

ALTER VIEW b view AS SELECT * FROM a tbl WHERE phone IS NOT NULL;
SELECT * FROM b_view;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  phone
=====
      1  '111-1111'
      2  '222-2222'
      3  '333-3333'

3 rows selected.

```

CHANGE QUERY 절

설명

ALTER VIEW 문의 **CHANGE QUERY** 절을 사용하여 뷰 질의 명세부에 정의된 질의를 변경할 수 있다.

구문

```
ALTER [ VIEW | VCLASS ] view_name  
CHANGE QUERY [ integer ] select_statement [ ; ]
```

- *view_name*: 변경할 뷰의 이름을 명시한다.
- *integer*: 변경할 질의의 번호를 명시한다. 디폴트 값은 1이다.
- *select_statement*: 질의 번호가 *integer*인 질의를 대체할 새로운 질의를 명시한다.

예제

```
--adding select_statement which query number is 2 and 3 for each  
ALTER VIEW b view ADD QUERY SELECT * FROM a tbl WHERE id IN (1,2);  
ALTER VIEW b view ADD QUERY SELECT * FROM a_tbl WHERE id = 3;  
SELECT * FROM b view;  
;xr  
  
=== <Result of SELECT Command in Line 1> ===  
  
      id  phone  
=====
```

1	'111-1111'
2	'222-2222'
3	'333-3333'
4	NULL
5	NULL
1	'111-1111'
2	'222-2222'
3	'333-3333'

```
  
8 rows selected.  
  
--altering view changing query number 2  
ALTER VIEW b view CHANGE QUERY 2 SELECT * FROM a tbl WHERE phone IS NULL;  
SELECT * FROM b view;  
;xr  
  
=== <Result of SELECT Command in Line 1> ===  
  
      id  phone  
=====
```

1	'111-1111'
2	'222-2222'
3	'333-3333'
4	NULL
5	NULL
4	NULL
5	NULL
3	'333-3333'

DROP QUERY 절

설명

ALTER VIEW 문의 **DROP QUERY** 예약어를 이용하여 뷰 질의 명세부에 정의된 질의를 삭제할 수 있다.

예제

```
ALTER VIEW b view DROP QUERY 2,3;
SELECT * FROM b_view;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  phone
=====
      1  '111-1111'
      2  '222-2222'
      3  '333-3333'
      4  NULL
      5  NULL

5 rows selected.
```

DROP VIEW

설명

뷰는 **DROP VIEW** 문을 이용하여 삭제할 수 있다. 뷰를 삭제하는 방법은 일반 테이블을 삭제하는 방법과 동일하다.

구문

```
DROP [ VIEW | VCLASS ] view_name [ { ,view_name , ... } ]
```

- *view_name* : 삭제하려는 뷰의 이름을 지정한다.

예제

```
DROP VIEW b_view;
```

RENAME VIEW

설명

뷰의 이름은 **RENAME VIEW** 문을 사용하여 변경할 수 있다.

구문

```
RENAME [ TABLE | CLASS | VIEW | VCLASS ] old_view_name AS new_view_name [ ; ]
```

- *old_view_name* : 변경할 뷰의 이름을 지정한다.
- *new_view_name* : 뷰의 새로운 이름을 지정한다.

예제

다음은 game_2004 뷰의 이름을 info_2004로 변경하는 예제이다.

```
RENAME VIEW game_2004 AS info_2004;
```

시리얼

CREATE SERIAL

시리얼(SERIAL)은 고유한 순번을 생성하는 객체이다. 시리얼은 다음과 같은 특성을 갖는다.

- 시리얼은 다중 사용자 환경에서 고유한 순번을 생성하는데 용이하다.
- 시리얼 번호는 테이블과 독립적으로 생성된다. 따라서 하나 이상의 테이블에 동일한 시리얼을 사용할 수 있다.
- **public**을 포함하여 모든 사용자가 시리얼 객체를 생성할 수 있다. 일단 생성되면 모든 사용자들이 **CURRENT_VALUE**, **NEXT_VALUE**를 통해 시리얼 숫자를 가져갈 수 있다.
- 시리얼 객체의 소유자와 **dba**만 시리얼 객체를 갱신하고 삭제할 수 있다. 소유자가 **public**이면 모든 사용자가 갱신하거나 삭제할 수 있다.

설명

CREATE SERIAL 문을 이용하여 데이터베이스에 시리얼 객체를 생성한다.

구문

```
CREATE SERIAL serial_name
[ START WITH initial ]
[ INCREMENT BY interval ]
[ MINVALUE min | NOMINVALUE ]
[ MAXVALUE max | NOMAXVALUE ]
[ CACHE integer | NOCACHE ]
```

- *serial_identifier*: 생성할 시리얼의 이름을 지정한다.
- **START WITH initial**: 처음 생성되는 시리얼 숫자를 지정한다. 이 값은 38자리 이하의 숫자이다. 오름차순 시리얼의 경우 디폴트 값은 시리얼의 최소값이며 내림차순 시리얼의 경우 디폴트 값은 시리얼의 최대값이다.
- **INCREMENT BY interval**: 시리얼 숫자 간의 간격을 지정한다. *interval* 값으로 0을 제외한 38자리 이하의 어떤 정수도 지정할 수 있다. *interval*의 절대값은 **MAXVALUE**와 **MINVALUE**의 차이보다 작아야 한다. 음수가 설정되면 시리얼은 내림차순이 되고 양수가 설정되면 오름차순이 된다. 디폴트 값은 **1**이다.
- **MINVALUE**: 시리얼의 최소값을 지정한다. 이 값은 38자리 이하의 숫자이다. **MINVALUE**는 초기값보다 작거나 같아야 하고 최대값보다 작아야 한다.
- **NOMINVALUE**: 오름차순 시리얼에 대해서는 1, 내림차순 시리얼에 대해서는 $-(10)^{38}$ 이 최소값으로 자동 지정된다.
- **MAXVALUE**: 시리얼의 최대값을 지정한다. 이 값은 38자리 이하의 숫자이다. **MAXVALUE**는 초기값보다 작거나 같아야 하고 최소값보다 커야 한다.
- **NOMAXVALUE**: 오름차순 시리얼에 대해서는 $(10)^{37}$, 내림차순 시리얼에 대해서는 -1이 최대값으로 자동 지정된다.

- **CYCLE** : 시리얼 값이 최대 또는 최소값에 도달한 후에 연속적으로 값을 생성하도록 지정한다. 오름차순 시리얼은 최대값에 도달한 후에 다음 값으로 최소값이 생성된다. 내림차순 시리얼은 최소값에 도달한 후에 다음 값으로 최대값이 생성된다.
- **NOCYCLE** : 시리얼이 최대 또는 최소값에 도달한 후에 시리얼 값이 더 이상 생성되지 않도록 지정한다. 디폴트 값은 **NOCYCLE**이다.
- **CACHE** : 시리얼 성능을 향상시키기 위하여 *integer*에 지정된 개수만큼의 시리얼을 캐시에 저장하고, 시리얼 값을 요청받으면 캐시된 시리얼 값을 가져온다. 또한, 메모리에 캐시된 시리얼을 전부 사용하게 되면, *integer* 개수 만큼의 시리얼을 디스크로부터 메모리로 다시 가져오며, 데이터베이스 서버가 중간에 종료되면 캐시된 시리얼 값들은 삭제된다. 따라서 데이터베이스 서버가 재시작되기 이전과 이후의 시리얼 값은 비연속적일 수 있다. 캐시된 시리얼은 트랜잭션 롤백되지 않으므로, 롤백을 수행하더라도 다음에 요청하는 시리얼은 이전에 최종 요청한 시리얼 값의 다음 값이 된다. **CACHE** 키워드 뒤에 *integer*는 생략할 수 없으며, 1 이하의 숫자가 지정되면 시리얼 캐시가 적용되지 않는다.
- **NOCACHE** : 시리얼 캐시 기능을 사용하지 않으며, 매번 시리얼 값이 업데이트되고, 요청 시마다 디스크로부터 시리얼 값을 가져온다.

예제 1

```
--creating serial with default values
CREATE SERIAL order no;

--creating serial within a specific range
CREATE SERIAL order no START WITH 10000 INCREMENT BY 2 MAXVALUE 20000;
--creating serial with specifying the number of cached serial values
CREATE SERIAL order no START WITH 10000 INCREMENT BY 2 MAXVALUE 20000 CACHE 3;

--selecting serial information from the db serial class
SELECT * FROM db_serial;
;xr

=== <Result of SELECT Command in Line 1> ===
```

name	current val	increment val	max val	min val	c
yclic	started	cached num	att name		
'order no'	10006	2	20000	10000	
0	1	3	NULL		

예제 2

다음은 선수 번호와 이름을 저장하는 athlete_idx 테이블을 생성하고 order_no 시리얼을 이용하여 인스턴스를 생성하는 예제이다. order_no.CURRENT_VALUE은 시리얼의 현재 값을 반환하고, order_no.NEXT_VALUE는 시리얼 값을 증가시킨 후 값을 반환한다.

```
CREATE TABLE athlete idx( code INT, name VARCHAR(40) );
CREATE SERIAL order no START WITH 10000 INCREMENT BY 2 MAXVALUE 20000;
INSERT INTO athlete idx VALUES (order no.NEXT VALUE, 'Park');
INSERT INTO athlete idx VALUES (order no.NEXT VALUE, 'Kim');
INSERT INTO athlete idx VALUES (order no.NEXT VALUE, 'Choo');
INSERT INTO athlete idx VALUES (order no.CURRENT VALUE, 'Lee');
SELECT * FROM athlete idx;

=== <Result of SELECT Command in Line 1> ===
```

code	name
10000	'Park'
10002	'Kim'


```
10004 'Choo'
10004 'Lee'
```

ALTER SERIAL

설명

ALTER SERIAL 문을 이용하면 시리얼 값의 증가치를 갱신하고 시작 값, 최소 값, 최대 값을 설정하거나 제거할 수 있으며, 순환 속성을 설정할 수 있다.

구문

```
ALTER SERIAL serial_identifier
[ INCREMENT BY interval ]
[ START WITH initial_value ]
[ MINVALUE min | NOMINVALUE ]
[ MAXVALUE max | NOMAXVALUE ]
[ CACHE integer | NOCACHE ]
```

- ***serial_identifier***: 생성할 시리얼의 이름을 지정한다.
- **INCREMENT BY *interval***: 시리얼 숫자간의 간격을 지정한다. *interval* 값으로 0을 제외한 38자리 이하의 어떤 정수도 지정할 수 있다. *interval*의 절대값은 **MAXVALUE**와 **MINVALUE**의 차이보다 작아야 한다. 음수가 설정되면 시리얼은 내림차순이 되고 양수가 설정되면 오름차순이 된다. 디폴트 값은 1이다.
- **START WITH *initial_value***: 시리얼의 시작 값을 변경한다.
- **MINVALUE**: 시리얼의 최소값을 지정한다. 이 값은 38자리 이하의 숫자이다. **MINVALUE**는 초기값보다 작거나 같아야 하고 최대값보다 작아야 한다.
- **NOMINVALUE**: 오름차순 시리얼에 대해서는 1, 내림차순 시리얼에 대해서는 $-(10)^{36}$ 이 최소값으로 자동 지정된다.
- **MAXVALUE**: 시리얼의 최대값을 지정한다. 이 값은 38자리 이하의 숫자이다. **MAXVALUE**는 초기값보다 작거나 같아야 하고 최소값보다 커야 한다.
- **NOMAXVALUE**: 오름차순 시리얼에 대해서는 $(10)^{37}$, 내림차순 시리얼에 대해서는 -1이 최대값으로 자동 지정된다.
- **CYCLE**: 시리얼 값이 최대 또는 최소값에 도달한 후에 연속적으로 값을 생성하도록 지정한다. 오름차순 시리얼은 최대값에 도달한 후에 다음 값으로 최소값이 생성된다. 내림차순 시리얼은 최소값에 도달한 후에 다음 값으로 최대값이 생성된다.
- **NOCYCLE**: 시리얼이 최대 또는 최소값에 도달한 후에 시리얼 값이 더 이상 생성되지 않도록 지정한다. 디폴트 값은 **NOCYCLE**이다.
- **CACHE**: 시리얼 성능을 향상시키기 위하여 *integer*에 지정된 개수만큼의 시리얼을 캐시에 저장하고, 시리얼 값을 요청받으면 캐시된 시리얼 값을 가져온다. **CACHE** 키워드 뒤에 *integer*는 생략할 수 없으며, 1 이하의 숫자가 지정되면 시리얼 캐시가 적용되지 않는다.
- **NOCACHE**: 시리얼 캐시 기능을 사용하지 않으며, 매번 시리얼 값이 업데이트되고, 요청 시마다 디스크로부터 시리얼 값을 가져온다.

주의 CUBRID 2008 R1.x 버전에서는 시스템 카탈로그인 db_serial 테이블을 업데이트하는 방식으로 시리얼 값을 변경할 수 있었으나, CUBRID 2008 R2.0 이상 버전부터는 db_serial 테이블의 수정은 허용되지 않고 **ALTER SERIAL** 구문을 이용하는 방식만 허용된다. 따라서 CUBRID 2008 R2.0 이상 버전에서

내보내기(unloadb)한 데이터에 **ALTER SERIAL** 구문이 포함된 경우에는 이를 CUBRID 2008 R1.x 이하 버전에서 가져오기(loadb)할 수 없다.

예제

```
--altering serial by changing start and incremental values
ALTER SERIAL order no START WITH 100 INCREMENT BY 2;

--altering serial to operate in cache mode
ALTER SERIAL order no CACHE 5;

--altering serial to operate in common mode
ALTER SERIAL order_no NOCACHE;
```

DROP SERIAL

설명

DROP SERIAL 문으로 시리얼 객체를 데이터베이스에서 삭제할 수 있다.

구문

```
DROP SERIAL serial_identifier
```

- *serial_identifier* : 삭제할 시리얼의 이름을 지정한다.

예제

다음 예는 *order_no* 시리얼을 삭제하는 예제이다.

```
DROP SERIAL order_no;
```

시리얼 사용

설명

시리얼은 시리얼 이름과 예약어를 질의 안에 삽입하여 읽고 갱신할 수 있다.

구문

```
serial_identifier.CURRENT_VALUE  
serial_identifier.NEXT_VALUE
```

- *serial_identifier*.**CURRENT_VALUE** : 시리얼의 현재 값을 반환한다.
- *serial_identifier*.**NEXT_VALUE** : 시리얼 값을 증가시키고 그 값을 반환한다.

예제

다음은 선수 번호와 이름을 저장하는 athlete_idx 테이블을 생성하고 order_no 시리얼을 이용하여 인스턴스를 생성하는 예제이다.

```
CREATE TABLE athlete_idx( code INT, name VARCHAR(40) );  
INSERT INTO athlete_idx VALUES (order no.NEXT VALUE, 'Park');  
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Kim');  
INSERT INTO athlete_idx VALUES (order no.NEXT_VALUE, 'Choo');  
INSERT INTO athlete_idx VALUES (order_no.NEXT_VALUE, 'Lee');SELECT * FROM athlete_idx;
```

```
=== <Result of SELECT Command in Line 1> ===
```

```
      code  name
=====
    10000  'Park'
    10002  'Kim'
    10004  'Choo'
    10006  'Lee'
```

주의 사항

- 시리얼을 생성하고 처음 사용할 때 **NEXT_VALUE**를 이용하면 초기 값을 반환한다. 그 이후에는 현재 값에 증가 값이 추가되어 반환된다.

연산자와 함수

논리 연산자

설명

논리 연산자(logical operator)는 피연산자로 항상 부울린(boolean) 연산식이 지정되며, 연산 결과로 **TRUE**, **FALSE**, **NULL**을 반환한다. CUBRID가 지원하는 논리 연산자의 종류 및 진리표는 아래와 같다.

CUBRID가 지원하는 논리 연산자

논리 연산자 설명	조건식
AND, && 피연산자가 모두 TRUE 이면 TRUE 를 반환한다.	a AND b
OR, 피연산자가 모두 NULL 이 아니고, 하나 이상의 피연산자가 TRUE 이면 TRUE 를 반환한다. SQL 구문 관련 파라미터인 pipes_as_concat 파라미터가 no 이면, 이중 파이프 기호()를 OR 연산자로 사용할 수 있다.	a OR b
XOR 피연산자가 모두 NULL 이 아니고, 두 피연산자의 값이 다르면 TRUE 를 반환한다.	a XOR b
NOT, ! 단항 연산자이며, 피연산자가 FALSE 이면 TRUE , 피연산자가 TRUE 이면 NOT a FALSE 를 반환한다.	NOT a

논리 연산자의 진리표

a	b	a AND b	a OR b	NOT a	a XOR b
TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
TRUE	NULL	NULL	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
FALSE	NULL	FALSE	NULL	TRUE	NULL

비교 연산자

설명

비교 연산자(comparison operator)는 왼쪽 피연산자와 오른쪽 피연산자를 비교하여 **TRUE**(1) 또는 **FALSE**(0)를 반환한다. 비교 연산의 피연산자들은 같은 데이터 타입이어야 하므로, 시스템에 의해서

묵시적으로 타입이 변환되거나 사용자에 의해 명시적으로 타입이 변환되어야 한다. 다음은 CUBRID에서 지원되는 비교 연산자의 설명 및 리턴 값을 나타낸 표이다.

CUBRID가 지원하는 비교 연산자

비교 연산자	설명	조건식	리턴 값
=	일반 등호이며, 두 피연산자의 값이 같은지 비교한다. 1=2 하나 이상의 피연산자가 NULL 이면 NULL 을 반환한다.	1=2 1=NULL	0 NULL
<=>	NULL safe 등호이며, NULL 을 포함하여 두 피연산자의 값이 같은지 비교한다. 피연산자가 모두 NULL 이면 1 을 반환한다.	1<=>2 1<=>NULL	0 0
<>, !=	두 피연산자의 값이 다른지 비교한다. 하나 이상의 피연산자가 NULL 이면 NULL 을 반환한다.	1<>2	1
>	왼쪽 피연산자가 오른쪽 피연산자보다 값이 큰지 비교한다. 하나 이상의 피연산자가 NULL 이면 NULL 을 반환한다.	1>2	0
<	왼쪽 피연산자가 오른쪽 피연산자보다 값이 작은지 비교한다. 하나 이상의 피연산자가 NULL 이면 NULL 을 반환한다.	1<2	1
>=	왼쪽 피연산자가 오른쪽 피연산자보다 값이 크거나 같은지 비교한다. 하나 이상의 피연산자가 NULL 이면 NULL 을 반환한다.	1>=2	0
<=	왼쪽 피연산자가 오른쪽 피연산자보다 값이 작거나 같은지 비교한다. 하나 이상의 피연산자가 NULL 이면 NULL 을 반환한다.	1<=2	1
IS boolean_value	왼쪽 피연산자가 오른쪽 불리언 값과 같은지 비교한다. 불리언 값은 TRUE , FALSE , NULL 이 될 수 있다.	1 IS FALSE	0
IS NOT boolean_value	왼쪽 피연산자가 오른쪽 불리언 값과 다른지 비교한다. 불리언 값은 TRUE , FALSE , NULL 이 될 수 있다.	1 IS NOT FALSE	1

구문 1

```
expression comparison operator expression
```

```
expression :
• bit string
• character string
• numeric value
• date-time value
• collection value
• NULL
```

```
comparison operator :
```

```
=
```

```
| <=>
| <>
| !=
| >
| <
| >=
| <=
```

구문 2

```
expression IS [NOT] boolean_value
```

```
expression :
• bit string
• character string
• numeric value
• date-time value
• collection value
• NULL
```

```
boolean value :
< UNKNOWN | NULL>
| TRUE
| FALSE
```

- *expression* : 비교할 수식을 선언한다.
- *bit string* : 비트열에 대하여 부울린(boolean) 연산을 수행할 수 있으며, 모든 비교 연산자를 비트열을 비교하는데 사용할 수 있다. 길이가 같지 않은 두 수식을 비교할 때는 길이가 짧은 비트열의 오른쪽 끝에 0이 추가된다.
- *character string* : 비교 연산자를 통해 비교할 두 문자열은 같은 문자 세트를 가져야 한다. 문자 코드와 연관된 정렬 체계(collation)에 의해 비교 순서가 결정된다. 서로 다른 길이의 문자열을 비교할 때는, 비교 전에 길이가 긴 문자열의 길이와 같아지도록 길이가 짧은 문자열 뒤에 공백을 추가한다.
- *numeric value* : 모든 숫자 값에 대해 부울린(Boolean)을 수행할 수 있으며, 모든 비교 연산자를 이용하여 비교 연산을 수행할 수 있다. 서로 다른 숫자 타입을 비교할 때에는 시스템이 묵시적으로 타입을 변환한다. 예를 들어, **INTEGER** 값을 **DECIMAL** 값과 비교할 때 시스템은 먼저 **INTEGER**를 **DECIMAL**로 변환한 후 비교한다. **FLOAT**에 대해서 비교할 때에는 **FLOAT**는 시스템 종속적으로 처리되므로 정확한 값이 아니라 범위를 지정해야 한다.
- *date-time value* : 날짜/시간 값을 같은 타입 간에 비교할 때에 값의 순서는 연대기 순으로 결정된다. 즉, 두 개의 날짜와 시간 값을 비교할 때, 이전 날짜가 나중 날짜보다 작은 것으로 간주된다. 서로 다른 타입의 날짜/시간 값에 대한 비교 연산은 허용되지 않으므로 명시적 타입 변환이 필요하지만, **DATE**, **TIMESTAMP**, **DATETIME** 타입 간에는 묵시적 타입 변환이 수행되어 비교 연산이 가능하다.
- *collection value* : 두 순차집합을 비교할 때에는 순차집합이 생성되었을 때 사용자가 명시한 순서대로 원소 간의 비교가 이루어진다. 집합과 다중집합을 포함하는 비교는 적절한 집합 연산자로 오버로딩 된다. 집합, 다중집합, 리스트 또는 순차집합에 대한 비교 연산을 이 장의 뒷부분에서 설명하는 포함 연산자를 이용하여 수행할 수 있다. 자세한 정보는 [포함 연산자](#)를 참조한다.
- **NULL** : **NULL** 값은 모든 데이터 타입의 값 범위 내에 포함되지 않는다. 따라서, **NULL** 값의 비교는 주어진 값이 **NULL** 값인지 아닌지에 대한 비교만 가능하다. **NULL** 값이 다른 데이터 타입으로 할당될 때 묵시적인 타입 변경은 일어나지 않는다. 예를 들어, **INTEGER** 타입의 컬럼이 **NULL** 값을 가지고 있고 부동 소수점 타입과 비교할 때, 비교하기 전에 **NULL** 값을 **FLOAT** 형으로 변환하지 않는다. **NULL** 값에 대한 비교 연산은 결과를 반환하지 않는다.

예제

```
EVALUATE (1 <> 0); -- TRUE이므로 1을 출력한다.
EVALUATE (1 != 0); -- TRUE이므로 1을 출력한다.
EVALUATE (0.01 = '0.01'); -- 숫자 타입과 문자열 타입을 비교했으므로 에러가 발생한다.
EVALUATE (1 = NULL); -- NULL을 출력한다.
EVALUATE (1 <=> NULL); -- FALSE이므로 0을 출력한다.
EVALUATE (1.000 = 1); -- TRUE이므로 1을 출력한다.
EVALUATE ('cubrid' = 'CUBRID'); -- 대소문자를 구분하므로 0을 출력한다.
EVALUATE ('cubrid' = 'cubrid'); -- TRUE이므로 1을 출력한다.
EVALUATE (SYSTIMESTAMP = CAST(SYSDATETIME AS TIMESTAMP)); -- 명시적으로 타입을 변환하여 비교 연산을 수행한 결과, 1을 출력한다.
EVALUATE (SYSTIMESTAMP = SYSDATETIME); -- 묵시적으로 타입을 변환하여 비교 연산을 수행한 결과, 0을 출력한다.
EVALUATE (SYSTIMESTAMP <> NULL); -- NULL의 비교 연산을 수행하지 않고 NULL을 반환한다.
EVALUATE (SYSTIMESTAMP IS NOT NULL); -- NULL이 아니므로 1을 반환한다.
```

산술 연산자

산술 연산자

설명

산술 연산자는 덧셈, 뺄셈, 곱셈, 나눗셈을 위한 이항(binary) 연산자와 양수, 음수를 나타내기 위한 단항(unary) 연산자가 있다. 양수/음수의 부호를 나타내는 단항 연산자의 연산 우선순위가 이항 연산자보다 높다.

CUBRID가 지원하는 산술 연산자

산술 연산자 설명	연산식	리턴 값
+	더하기 연산	1+2 3
-	빼기 연산	1-2 -1
*	곱하기 연산	1*2 2
/	나누기 연산 후, 몫을 반환한다.	1/2.0 0.500000000
DIV	나누기 연산 후, 몫을 반환한다. 피연산자는 정수 타입이어야 하며, 정수를 반환한다.	1 DIV 2 0
%, MOD	나누기 연산 후, 나머지를 반환한다. 피연산자는 정수 타입이어야 하며, 정수를 반환한다. 피연산자가 실수이면 MOD 함수를 이용한다.	1 % 2 1 1 MOD 2

구문

```
expression mathematical operator expression
expression :
```

```

• bit string
• character string
• numeric value
• date-time value
• collection value
• NULL

mathematical operator :
• set_arithmetic_operator
• arithmetic_operator

arithmetic_operator :
• +
• -
• *
• /, DIV
• %, MOD

set arithmetic_operator :
• UNION (합집합)
• DIFFERENCE (차집합)
• INTERSECT | INTERSECTION (교집합)

```

- *expression*: 연산을 수행할 수식을 선언한다.
- *mathematical_operator*: 수학적 연산을 지정하는 연산자로서, 산술 연산자와 집합 연산자가 있다.
- *set_arithmetic_operator*: 컬렉션 타입의 피연산자에 대해 합집합, 차집합, 교집합을 수행하는 집합 산술 연산자이다.
- *arithmetic_operator*: 사칙 연산을 수행하기 위한 연산자이다.

수치형 데이터 타입의 산술 연산과 타입 변환

설명

모든 수치형 데이터 타입을 산술 연산에 사용할 수 있으며, 연산 결과 타입은 피연산자의 데이터 타입과 연산의 종류에 따라 다르다. 아래는 피연산자 타입별 덧셈/뺄셈/곱셈 연산의 결과 데이터 타입을 정리한 표이다.

피연산자의 타입별 결과 데이터 타입

	INT	NUMERIC	FLOAT	DOUBLE	MONETARY
INT	INT (BIGINT 범위 까지)	NUMERIC	FLOAT	DOUBLE	MONETARY
NUMERIC	NUMERIC	NUMERIC (p 와 s 도 변환 됨)	DOUBLE	DOUBLE	MONETARY
FLOAT	FLOAT	DOUBLE	FLOAT	DOUBLE	MONETARY
DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	MONETARY
MONETARY	MONETARY	MONETARY	MONETARY	MONETARY	MONETARY

피연산자가 모두 동일한 데이터 타입이면 연산 결과의 타입이 변환되지 않으나, 나누기 연산의 경우 예외적으로 타입이 변환되므로 주의해야 한다. 분모, 즉 제수(divisor)가 0이면 에러가 발생한다.

피연산자 중 하나가 **MONETARY** 타입인 경우, **DOUBLE** 타입과 같은 연산 방식을 채택하고 있으므로 연산 결과는 모두 **MONETARY** 타입으로 반환된다.

아래는 피연산자가 모두 **NUMERIC** 타입인 경우, 연산 결과의 전체 자릿수(p)와 소수점 아래 자릿수(s)를 정리한 표이다.

NUMERIC 타입의 연산 결과

연산	결과의 최대 자릿수	결과의 소수점 이하 자릿수
$N(p_1, s_1) + N(p_2, s_2)$	$\max(p_1-s_1, p_2-s_2)+\max(s_1, s_2)+1$	$\max(s_1, s_2)$
$N(p_1, s_1) - N(p_2, s_2)$	$\max(p_1-s_1, p_2-s_2)+\max(s_1, s_2)$	$\max(s_1, s_2)$
$N(p_1, s_1) * N(p_2, s_2)$	p_1+p_2+1	s_1+s_2
$N(p_1, s_1) / N(p_2, s_2)$	$s_2 > 0$ 이면 $P_t = p_1+\max(s_1, s_2) + s_2 - s_1$, 그 외에는 $P_t = p_1$ 라 하고, $s_1 > s_2$ 이면 $S_t = s_1$, 그 외에는 s_2 라 하면, 소수점 이하 자릿수는 $S_t < 9$ 이면 $\min(9-S_t, 38-P_t) + S_t$, 그 외에는 S_t	

예제

```
--int * int
SELECT 123*123;
=====
      15129

-- int * int returns overflow error
SELECT (1234567890123*1234567890123);

-- int * numeric returns numeric type
SELECT (1234567890123*CAST(1234567890123 AS NUMERIC(15,2)));
=====
1524157875322755800955129.00

-- int * float returns float type
SELECT (1234567890123*CAST(1234567890123 AS FLOAT));
=====
1.524158e+024

-- int * double returns double type
SELECT (1234567890123*CAST(1234567890123 AS DOUBLE));
=====
1.524157875322756e+024

-- numeric * numeric returns numeric type
SELECT (CAST(1234567890123 AS NUMERIC(15,2))*CAST(1234567890123 AS NUMERIC(15,2)));
=====
1524157875322755800955129.0000

-- numeric * float returns double type
SELECT (CAST(1234567890123 AS NUMERIC(15,2))*CAST(1234567890123 AS FLOAT));
=====
1.524157954716582e+024

-- numeric * double returns double type
SELECT (CAST(1234567890123 AS NUMERIC(15,2))*CAST(1234567890123 AS DOUBLE));
=====
```

```

1.524157875322756e+024

-- float * float returns float type
SELECT (CAST(1234567890123 AS FLOAT)*CAST(1234567890123 AS FLOAT));
=====
1.524158e+024

-- float * double returns float type
SELECT (CAST(1234567890123 AS FLOAT)*CAST(1234567890123 AS DOUBLE));
=====
1.524157954716582e+024

-- double * double returns float type
SELECT (CAST(1234567890123 AS DOUBLE)*CAST(1234567890123 AS DOUBLE));
=====
1.524157875322756e+024

-- int / int returns int type without type conversion or rounding
SELECT 100100/100000;
=====
1

-- int / int returns int type without type conversion or rounding
SELECT 100100/200200;
=====
0

-- int / zero returns error
SELECT 100100/(100100-100100);
ERROR: Attempt to divide by zero.

```

날짜/시간 데이터 타입의 산술 연산과 타입 변환

설명

피연산자가 모두 날짜/시간 데이터 타입이면 뺄셈 연산이 가능하며, 리턴 값의 타입은 INT이다. 이때 피연산자의 타입에 따라 연산 단위가 다르므로 주의한다. 날짜/시간 데이터 타입과 정수는 덧셈 및 뺄셈 연산이 가능하며, 이때 연산 단위와 리턴 값의 타입은 날짜/시간 데이터 타입을 따른다.

아래는 피연산자 타입별 허용 연산과 결과 데이터 타입을 정리한 표이다.

피연산자의 타입별 허용 연산과 결과 데이터 타입

	TIME (초 단위)	DATE (일 단위)	TIMESTAMP (초 단위)	DATETIME (밀리초 단위)	INT
TIME	뺄셈만 허용	X	X	X	덧셈, 뺄셈 허용
INT					TIME
DATE	X	뺄셈만 허용	뺄셈만 허용	뺄셈만 허용	덧셈, 뺄셈 허용
		INT	INT	INT	DATE
TIMESTAMP	X	뺄셈만 허용	뺄셈만 허용	뺄셈만 허용	덧셈, 뺄셈 허용
		INT	INT	INT	TIMESTAMP
DATETIME	X	뺄셈만 허용	뺄셈만 허용	뺄셈만 허용	덧셈, 뺄셈 허용
		INT	INT	INT	DATETIME
INT	덧셈, 뺄셈 허용	덧셈, 뺄셈 허용	덧셈, 뺄셈 허용	덧셈, 뺄셈 허용	모든 산술 연산 허용

TIME	DATE	TIMESTAMP	DATETIME
------	------	-----------	----------

참고 사항

날짜/시간 산술 연산의 인자 중 하나라도 **NULL**이 포함되어 있으면 수식의 결과로 **NULL**이 반환된다.

예제

```
-- initial systimestamp value
SELECT SYSDATETIME;
=====
07:09:52.115 PM 01/14/2010

-- time type + 10(seconds) returns time type
SELECT (CAST (SYSDATETIME AS TIME) + 10);
=====
07:10:02 PM

-- date type + 10 (days) returns date type
SELECT (CAST (SYSDATETIME AS DATE) + 10);
=====
01/24/2010

-- timestamp type + 10(seconds) returns timestamp type
SELECT (CAST (SYSDATETIME AS TIMESTAMP) + 10);
=====
07:10:02 PM 01/14/2010

-- systimestamp type + 10(milliseconds) returns systimestamp type
SELECT (SYSDATETIME + 10);
=====
07:09:52.125 PM 01/14/2010

SELECT DATETIME '09/01/2009 03:30:30.001 pm'- TIMESTAMP '08/31/2009 03:30:30 pm';
=====
86400001

SELECT TIMESTAMP '09/01/2009 03:30:30 pm'- TIMESTAMP '08/31/2009 03:30:30 pm';
=====
86400
```

집합 연산자

집합 산술 연산자

설명

집합형 데이터(**SET**, **MULTISET**, **LIST(=SEQUENCE)**) 타입에 대해 합집합, 차집합, 교집합을 구하기 위해서 각각 **+**, **-**, ***** 연산자를 사용할 수 있다. 다음은 집합형 데이터 타입이 피연산자인 경우, 연산별 결과 데이터 타입을 나타낸 표이다.

피연산자의 타입별 결과 데이터 타입

	SET	MULTISET	LIST (=SEQUENCE)
SET	+ , - , * : SET	+ , - , * : MULTISET	+ , - , * : MULTISET
MULTISET	+ , - , * : MULTISET	+ , - , * : MULTISET	+ , - , * : MULTISET

LIST	+ : MULTISSET	+ : MULTISSET	+ : LIST
(=SEQUENCE)	- : MULTISSET	- : MULTISSET	- : MULTISSET
	* : MULTISSET	* : MULTISSET	* : MULTISSET

구문

value expression set arithmetic operator value expression

value expression :

- *collection value*
- **NULL**

set arithmetic operator :

- **+** (합집합)
- **-** (차집합)
- ***** (교집합)

예제

```
SELECT ((CAST ({3,3,3,2,2,1} AS SET))+(CAST ({4,3,3,2} AS MULTISSET)))
FROM db_root;
=====
{1, 2, 2, 3, 3, 3, 4}

SELECT ((CAST ({3,3,3,2,2,1} AS MULTISSET))+(CAST ({4,3,3,2} AS MULTISSET)))
FROM db_root;
=====
{1, 2, 2, 2, 3, 3, 3, 3, 3, 4}

SELECT ((CAST ({3,3,3,2,2,1} AS LIST))+(CAST ({4,3,3,2} AS MULTISSET)))
FROM db_root;
=====
{1, 2, 2, 2, 3, 3, 3, 3, 3, 4}

SELECT ((CAST ({3,3,3,2,2,1} AS SET))-(CAST ({4,3,3,2} AS MULTISSET)))
FROM db_root;
=====
{1}

SELECT ((CAST ({3,3,3,2,2,1} AS MULTISSET))-(CAST ({4,3,3,2} AS MULTISSET)))
FROM db_root;
=====
{1, 2, 3}

SELECT ((CAST ({3,3,3,2,2,1} AS LIST))-(CAST ({4,3,3,2} AS MULTISSET)))
FROM db_root;
=====
{1, 2, 3}

SELECT ((CAST ({3,3,3,2,2,1} AS SET))*(CAST ({4,3,3,2} AS MULTISSET)))
FROM db_root;
=====
{2, 3}

SELECT ((CAST ({3,3,3,2,2,1} AS MULTISSET))*(CAST ({4,3,3,2} AS MULTISSET)))
FROM db_root;
=====
{2, 3, 3}

SELECT ((CAST ({3,3,3,2,2,1} AS LIST))*(CAST ({4,3,3,2} AS MULTISSET)))
FROM db_root;
=====
{2, 3, 3}
```

변수에 집합 데이터 값 할당

컬렉션 값을 변수에 할당하기 위해서는 외부 질의가 하나의 행만을 반환해야 한다.

다음은 컬렉션 값을 변수에 할당하는 방법을 나타내는 예제이다. 다음과 같이 외부 질의는 하나의 행만을 반환해야 한다.

```
SELECT SET(SELECT name
FROM people
WHERE ssn in {'1234', '5678'})
TO : "names"
FROM TABLE people;
```

문장 집합 연산자

설명

피연산자로 지정된 하나 이상의 질의문의 결과에 대해 합집합(**UNION**), 차집합(**DIFFERENCE**), 교집합(**INTERSECTION**)을 구하기 위하여 문장 집합 연산자(Statement Set Operator)를 이용한다. 단, 두 질의문의 대상 테이블에서 조회하고자 하는 데이터 타입이 동일하거나, 묵시적으로 변환 가능해야 한다. 다음은 CUBRID가 지원하는 문장 집합 연산자와 예제를 나타낸 표이다.

CUBRID가 지원하는 문장 집합 연산자

문장 집합 연산자 설명	비고
UNION 합집합 중복을 허용하지 않음	UNION ALL 이면 중복된 값을 포함한 모든 결과 인스턴스 출력
DIFFERENCE 차집합 중복을 허용하지 않음	EXCEPT 연산자와 동일 DIFFERENCE ALL 이면 중복된 값을 포함한 모든 결과 인스턴스 출력
INTERSECTION 교집합 중복을 허용하지 않음	INTERSECT 연산자와 동일 INTERSECTION ALL 이면 중복된 값을 포함한 모든 결과 인스턴스 출력

서식 있음: 글꼴: (영어) 맑은 고딕, 9 pt, (Intl) Verdana

구문

```
query term statement set operator[qualifier] query term
[statement_set_operator[qualifier] query_term];

query_term :
• query_specification
• subquery

qualifier :
• DISTINCT 또는 DISTINCTROW (결과로 반환되는 인스턴스가 서로 다르다는 것을 보장)
• UNIQUE (결과로 반환되는 인스턴스가 서로 다르다는 것을 보장)
• ALL (모든 인스턴스가 반환, 중복 허용)

statement_set_operator :
• UNION (합집합)
```

- **DIFFERENCE** (차집합)
- **INTERSECTION** | **INTERSECT** (교집합)

예제

```
CREATE TABLE nojoin_tbl_1 (ID INT, Name VARCHAR(32));

INSERT INTO nojoin_tbl_1 VALUES (1,'Kim');
INSERT INTO nojoin_tbl_1 VALUES (2,'Moy');
INSERT INTO nojoin_tbl_1 VALUES (3,'Jonas');
INSERT INTO nojoin_tbl_1 VALUES (4,'Smith');
INSERT INTO nojoin_tbl_1 VALUES (5,'Kim');
INSERT INTO nojoin_tbl_1 VALUES (6,'Smith');
INSERT INTO nojoin_tbl_1 VALUES (7,'Brown');

CREATE TABLE nojoin_tbl_2 (id INT, Name VARCHAR(32));

INSERT INTO nojoin_tbl_2 VALUES (5,'Kim');
INSERT INTO nojoin_tbl_2 VALUES (6,'Smith');
INSERT INTO nojoin_tbl_2 VALUES (7,'Brown');
INSERT INTO nojoin_tbl_2 VALUES (8,'Lin');
INSERT INTO nojoin_tbl_2 VALUES (9,'Edwin');
INSERT INTO nojoin_tbl_2 VALUES (10,'Edwin');

--Using UNION to get only distinct rows
SELECT id, name FROM nojoin_tbl_1
UNION
SELECT id,name FROM nojoin_tbl_2;

      id  name
=====
      1  'Kim'
      2  'Moy'
      3  'Jonas'
      4  'Smith'
      5  'Kim'
      6  'Smith'
      7  'Brown'
      8  'Lin'
      9  'Edwin'
     10  'Edwin'

--Using UNION ALL not eliminating duplicate selected rows
SELECT id, name FROM nojoin_tbl_1
UNION ALL
SELECT id,name FROM nojoin_tbl_2;

      id  name
=====
      1  'Kim'
      2  'Moy'
      3  'Jonas'
      4  'Smith'
      5  'Kim'
      6  'Smith'
      7  'Brown'
      8  'Kim'
      9  'Smith'
     10  'Brown'
     11  'Lin'
     12  'Edwin'
     13  'Edwin'

--Using DEFFERENCE to get only rows returned by the first query but not by the second
SELECT id, name FROM nojoin_tbl_1
DEFFERENCE
SELECT id,name FROM nojoin_tbl_2;

      id  name
=====
      1  'Kim'
```

```
2 'Moy'
3 'Jonas'
4 'Smith'

--Using INTERSECTION to get only those rows returned by both queries
SELECT id, name FROM nojoin tbl 1
INTERSECT
SELECT id,name FROM nojoin tbl 2;

id name
=====
5 'Kim'
6 'Smith'
7 'Brown'
```

포함 연산자

개요

설명

집합형 데이터 타입인 피연산자 간 비교 연산을 수행하여 포함(containment) 관계를 확인하기 위해 포함 연산자가 사용된다. 피연산자로 집합형 데이터 타입 또는 부질의(subquery)를 지정할 수 있으며, 두 피연산자의 포함 관계(동일하다/다르다/부분집합이다/진부분집합이다)에 따라 **TRUE** 또는 **FALSE**를 반환한다.

다음은 CUBRID가 지원하는 포함 연산자에 관한 설명 및 리턴 값을 나타낸 표이다.

CUBRID가 지원하는 포함 연산자			
포함 연산자	설명	조건식	리턴 값
A SETEQ B	A = B 집합 A와 집합 B의 원소가 서로 같다.	{1,2} SETEQ {1,2,2}	0
A SETNEQ B	A ≠ B 집합 A와 집합 B의 원소가 같지 않다.	{1,2} SETNEQ {1,2,3}	1
A SUPERSET B	A ⊃ B 집합 B는 집합 A의 진 부분집합이다.	{1,2} SUPERSET {1,2,3}	0
A SUBSET B	A ⊂ B 집합 A는 집합 B의 진 부분집합이다.	{1,2} SUBSET {1,2,3}	1
A SUPERSETEQ B	A ≥ B 집합 B는 집합 A의 부분 집합이다.	{1,2} SUPERSETEQ {1,2,3}	0
A SUBSETEQ B	A ≤ B 집합 A는 집합 B의 부분 집합이다.	{1,2} SUBSETEQ {1,2,3}	1

다음은 포함 연산자를 이용하는 경우, 피연산자의 타입별 연산 가능 여부 및 타입 변환 여부를 나타낸 표이다.

포함 연산자의 피연산자 타입별 연산 가능 여부

	SET	MULTISET	LIST(=SEQUENCE)
SET	연산 가 능	연산 가능	연산 가능
MULTISET	연산 가 능	연산 가능	연산 가능 (LIST 타입은 MULTISET 타입으로 변환됨)
LIST(=SEQUENCE)	연산 가 능	연산 가능 (LIST 타입은 MULTISET 타입 으로 변환됨)	일부 연산만 가능 (SETEQ, SETNEQ) 나머지 연산은 에러 발생

구문

```
collection_operand containment_operator collection_operand
```

```
collection_operand:
• set
• multiset
• sequence(또는 list)
• subquery
• NULL
```

```
containment_operator:
```

```
• SETEQ
• SETNEQ
• SUPERSET
• SUBSET
• SUPERSETEQ
• SUBSETEQ
```

- *collection_operand*: 피연산자로 지정될 수 있는 수식은 하나의 집합 값 속성(SET-valued attribute)이거나, 집합 연산자(SET operator)를 지닌 산술 수식(arithmetic expression)이거나, 중괄호로 둘러싸인 집합 값이다. 이때, 중괄호로 둘러싸인 집합 값은 타입을 명시하지 않을 경우 기본적으로 **LIST** 타입으로 처리한다.

피연산자로 부질의가 지정될 수 있으며, 집합형 데이터 타입이 아닌 컬럼을 조회하는 경우에는 **SET(subquery)**과 같이 해당 부질의에 집합형 데이터 타입 키워드를 붙여야 한다. 부질의에서 조회하는 컬럼은 하나의 집합만 결과로 반환해야 나머지 피연산자 집합과 비교할 수 있다.

집합 원소의 타입이 오브젝트이면, 오브젝트의 내용이 아닌 객체 식별자(OID, object identifier)에 대해 비교한다. 예를 들어, 같은 속성 값을 갖고 OID가 다른 두 오브젝트는 서로 다른 것으로 간주한다.

- **NULL**: 비교 대상이 되는 피연산자 중 어느 하나가 **NULL**인 경우, **NULL**이 반환된다.

예제

```
--empty set is a subset of any set
EVALUATE ({} SUBSETEQ (CAST ({3,1,2} AS SET)));
=====
1

--operation between set type and null returns null
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ NULL);
=====
NULL

--{1,2,3} seteq {1,2,3} returns true
```



```

EVALUATE ((CAST ({3,1,2} AS SET)) SETEQ (CAST ({1,2,3,3} AS SET)));
=====
1

--{1,2,3} seteq {1,2,3,3} returns false
EVALUATE ((CAST ({3,1,2} AS SET)) SETEQ (CAST ({1,2,3,3} AS MULTISSET)));
=====
0

--{1,2,3} setneq {1,2,3,3} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SETNEQ (CAST ({1,2,3,3} AS MULTISSET)));
=====
1

--{1,2,3} subseteq {1,2,3,4} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ (CAST ({1,2,4,4,3} AS SET)));
=====
1

--{1,2,3} subseteq {1,2,3,4,4} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ (CAST ({1,2,4,4,3} AS MULTISSET)));
=====
1

--{1,2,3} subseteq {1,2,4,4,3} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ (CAST ({1,2,4,4,3} AS LIST)));
=====
0

--{1,2,3} subseteq {1,2,3,4,4} returns true
EVALUATE ((CAST ({3,1,2} AS SET)) SUBSETEQ (CAST ({1,2,3,4,4} AS LIST)));
=====
1

--{3,1,2} seteq {3,1,2} returns true
EVALUATE ((CAST ({3,1,2} AS LIST)) SETEQ (CAST ({3,1,2} AS LIST)));
=====
1
--error occurs because LIST subseteq LIST is not supported
EVALUATE ((CAST ({3,1,2} AS LIST)) SUBSETEQ (CAST ({3,1,2} AS LIST)));
=====
error

```

SETEQ 연산자

설명

SETEQ 연산자는 첫 번째 피연산자와 두 번째 피연산자가 동일한 경우 **TRUE(1)**을 반환한다. 모든 집합형 데이터 타입에 대해 비교 연산을 수행할 수 있다.

구문

```
collection_operand SETEQ collection_operand
```

예제

```

--creating a table with SET type address column and LIST type zip code column

CREATE TABLE contain_tbl (id int primary key, name char(10), address SET varchar(20),
zip_code LIST int);
INSERT INTO contain_tbl VALUES(1, 'Kim', {'country', 'state'},{1, 2, 3});
INSERT INTO contain_tbl VALUES(2, 'Moy', {'country', 'state'},{3, 2, 1});
INSERT INTO contain_tbl VALUES(3, 'Jones', {'country', 'state', 'city'},{1,2,3,4});
INSERT INTO contain_tbl VALUES(4, 'Smith', {'country', 'state', 'city',
'street'},{1,2,3,4});
INSERT INTO contain_tbl VALUES(5, 'Kim', {'country', 'state', 'city', 'street'},{1,2,3,4});

```

```

INSERT INTO contain_tbl VALUES(6, 'Smith', {'country', 'state', 'city',
'street'}, {1,2,3,5});
INSERT INTO contain_tbl VALUES(7, 'Brown', {'country', 'state', 'city', 'street'}, {});

--selecting rows when two collection operands are same in the WEHRE clause
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SETEQ {'country','state',
'city'};

=== <Result of SELECT Command in Line 1> ===

=====
id name address zip_code
=====
3 'Jones' {'city', 'country', 'state'} {1, 2, 3, 4}

1 row selected.

--selecting rows when two collection operands are same in the WEHRE clause
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SETEQ {1,2,3};

=== <Result of SELECT Command in Line 1> ===

=====
id name address zip_code
=====
1 'Kim' {'country', 'state'} {1, 2, 3}

1 rows selected.

```

SETNEQ 연산자

설명

SETNEQ 연산자는 첫 번째 피연산자와 두 번째 피연산자가 동일하지 않은 경우에 **TRUE**(1)을 반환한다. 모든 집합형 데이터 타입에 대해 비교 연산을 수행할 수 있다.

구문

```
collection_operand SETNEQ collection_operand
```

예제

```

--selecting rows when two collection operands are not same in the WEHRE clause
SELECT id, name, address, zip code FROM contain_tbl WHERE address SETNEQ
{'country','state', 'city'};

=== <Result of SELECT Command in Line 1> ===

=====
id name address zip code
=====
1 'Kim' {'country', 'state'} {1, 2, 3}
2 'Moy' {'country', 'state'} {3, 2, 1}
4 'Smith' {'city', 'country', 'state', 'street'} {1, 2, 3, 4}
5 'Kim' {'city', 'country', 'state', 'street'} {1, 2, 3, 4}
6 'Smith' {'city', 'country', 'state', 'street'} {1, 2, 3, 5}
7 'Brown' {'city', 'country', 'state', 'street'} {}

6 rows selected.

--selecting rows when two collection operands are not same in the WEHRE clause
SELECT id, name, address, zip code FROM contain_tbl WHERE zip code SETNEQ {1,2,3};

=== <Result of SELECT Command in Line 1> ===

=====
id name address zip code
=====
2 'Moy' {'country', 'state'} {3, 2, 1}
3 'Jones' {'city', 'country', 'state'} {1, 2, 3, 4}
4 'Smith' {'city', 'country', 'state', 'street'} {1, 2, 3, 4}

```

```

5 'Kim      ' ('city', 'country', 'state', 'street') {1, 2, 3, 4}
6 'Smith    ' ('city', 'country', 'state', 'street') {1, 2, 3, 5}
7 'Brown    ' ('city', 'country', 'state', 'street') {}
6 rows selected.

```

SUPERSET 연산자

설명

SUPERSET 연산자는 첫 번째 피연산자가 두 번째 피연산자의 모든 원소를 포함하는 경우, 즉 두 번째 피연산자가 첫 번째 피연산자의 진부분집합인 경우 **TRUE(1)**을 반환한다. 피연산자 집합이 서로 동일한 경우에는 **FALSE(0)**을 반환한다. 단, 피연산자가 모두 **LIST** 타입인 경우에는 **SUPERSET** 연산을 지원하지 않는다.

구문

```
collection_operand SUPERSET collection_operand
```

예제

```

--selecting rows when the first operand is a superset of the second operand and they are
not same
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SUPERSET
{'country','state','city'};

=== <Result of SELECT Command in Line 1> ===

      id name      address                                zip_code
=====
      4 'Smith    ' ('city', 'country', 'state', 'street') {1, 2, 3, 4}
      5 'Kim      ' ('city', 'country', 'state', 'street') {1, 2, 3, 4}
      6 'Smith    ' ('city', 'country', 'state', 'street') {1, 2, 3, 5}
      7 'Brown    ' ('city', 'country', 'state', 'street') {}
4 rows selected.

--SUPERSET operator cannot be used for comparison between LIST and LIST type values
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUPERSET {1,2,3};

ERROR: ' superset ' operator is not defined on types sequence and sequence.

--Comparing operands with a SUPERSET operator after casting LIST type as SET type
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUPERSET (CAST ({1,2,3}
AS SET));

=== <Result of SELECT Command in Line 1> ===

      id name      address                                zip_code
=====
      3 'Jones    ' ('city', 'country', 'state') {1, 2, 3, 4}
      4 'Smith    ' ('city', 'country', 'state', 'street') {1, 2, 3, 4}
      5 'Kim      ' ('city', 'country', 'state', 'street') {1, 2, 3, 4}
      6 'Smith    ' ('city', 'country', 'state', 'street') {1, 2, 3, 5}
4 rows selected.

```

SUPERSETEQ 연산자

설명

SUPERSETEQ 연산자는 첫 번째 피연산자가 두 번째 피연산자의 모든 원소를 포함하거나 서로 동일한 경우, 즉 두 번째 피연산자가 첫 번째 피연산자의 부분집합인 경우 **TRUE(1)**를 반환한다. 단, 피연산자가 모두 **LIST** 타입인 경우에는 **SUPERSETEQ** 연산을 지원하지 않는다.

구문

```
collection_operand SUPERSETEQ collection_operand
```

예제

```
--selecting rows when the first operand is a superset of the second operand
SELECT id, name, address, zip code FROM contain_tbl WHERE address SUPERSETEQ
{'country','state','city'};

=== <Result of SELECT Command in Line 1> ===

      id  name      address      zip_code
=====
      3  'Jones    '      {'city', 'country', 'state'} {1, 2, 3, 4}
      4  'Smith    '      {'city', 'country', 'state', 'street'} {1, 2, 3, 4}
      5  'Kim      '      {'city', 'country', 'state', 'street'} {1, 2, 3, 4}
      6  'Smith    '      {'city', 'country', 'state', 'street'} {1, 2, 3, 5}
      7  'Brown    '      {'city', 'country', 'state', 'street'} {}

5 rows selected.

--SUPERSETEQ operator cannot be used for comparison between LIST and LIST type values
SELECT id, name, address, zip code FROM contain_tbl WHERE zip code SUPERSETEQ {1,2,3};

ERROR: ' superseteq ' operator is not defined on types sequence and sequence.

--Comparing operands with a SUPERSETEQ operator after casting LIST type as SET type
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUPERSETEQ (CAST
({1,2,3} AS SET));

=== <Result of SELECT Command in Line 1> ===

      id  name      address      zip code
=====
      1  'Kim      '      {'country', 'state'} {1, 2, 3}
      3  'Jones    '      {'city', 'country', 'state'} {1, 2, 3, 4}
      4  'Smith    '      {'city', 'country', 'state', 'street'} {1, 2, 3, 4}
      5  'Kim      '      {'city', 'country', 'state', 'street'} {1, 2, 3, 4}
      6  'Smith    '      {'city', 'country', 'state', 'street'} {1, 2, 3, 5}

5 rows selected.
```

SUBSET 연산자

설명

SUBSET 연산자는 두 번째 피연산자가 첫 번째 피연산자의 모든 원소를 포함하는 경우, 즉 첫 번째 피연산자가 두 번째 피연산자의 진부분집합인 경우 **TRUE(1)**을 반환한다. 피연산자 집합이 서로 동일한 경우에는 **FALSE(0)**를 반환한다. 단, 피연산자가 모두 **LIST** 타입인 경우에는 **SUBSET** 연산을 지원하지 않는다.

구문

```
collection_operand SUBSET collection_operand
```

예제

```
--selecting rows when the first operand is a subset of the second operand and they are not
same
SELECT id, name, address, zip_code FROM contain_tbl WHERE address SUBSET
{'country','state','city'};

=== <Result of SELECT Command in Line 1> ===

=====
id  name      address                                zip code
=====
1   'Kim      '    {'country', 'state'} {1, 2, 3}
2   'Moy      '    {'country', 'state'} {3, 2, 1}

2 rows selected.

--SUBSET operator cannot be used for comparison between LIST and LIST type values
SELECT id, name, address, zip code FROM contain_tbl WHERE zip code SUBSET {1,2,3};

ERROR: ' subset ' operator is not defined on types sequence and sequence.

--Comparing operands with a SUBSET operator after casting LIST type as SET type
SELECT id, name, address, zip code FROM contain_tbl WHERE zip code SUBSET (CAST ({1,2,3}
AS SET));

=== <Result of SELECT Command in Line 1> ===

=====
id  name      address                                zip code
=====
7   'Brown    '    {'city', 'country', 'state', 'street'} {}

1 rows selected.
```

SUBSETEQ 연산자

설명

SUBSETEQ 연산자는 두 번째 피연산자가 첫 번째 피연산자의 모든 원소를 포함하거나 서로 동일한 경우, 즉 첫 번째 피연산자가 두 번째 피연산자의 부분집합인 경우 **TRUE(1)**을 반환한다. 단, 피연산자가 모두 **LIST** 타입인 경우에는 **SUBSETEQ** 연산을 지원하지 않는다.

구문

```
collection_operand SUBSETEQ collection_operand
```

예제

```
--selecting rows when the first operand is a subset of the second operand
SELECT id, name, address, zip code FROM contain_tbl WHERE address SUBSETEQ
{'country','state','city'};

=== <Result of SELECT Command in Line 1> ===

=====
id  name      address                                zip code
=====
1   'Kim      '    {'country', 'state'} {1, 2, 3}
2   'Moy      '    {'country', 'state'} {3, 2, 1}
3   'Jones    '    {'city', 'country', 'state'} {1, 2, 3, 4}

3 rows selected.
```

```
--SUBSETEQ operator cannot be used for comparison between LIST and LIST type values
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUBSETEQ {1,2,3};

ERROR: ' subseteq ' operator is not defined on types sequence and sequence.

--Comparing operands with a SUBSETEQ operator after casting LIST type as SET type
SELECT id, name, address, zip_code FROM contain_tbl WHERE zip_code SUBSETEQ (CAST ({1,2,3}
AS SET));

=== <Result of SELECT Command in Line 1> ===

=====
id  name      address                                zip_code
=====
1   'Kim      '   {'country', 'state'} {1, 2, 3}
7   'Brown    '   {'city', 'country', 'state', 'street'} {}
=====

2 rows selected.
```

비트 함수와 연산자

비트 연산자

비트 연산자(Bitwise operator)는 비트 단위로 연산을 수행하며 산술 연산식에서 이용될 수 있다. 피연산자로 정수 타입이 지정되며, **BIT** 타입은 지정될 수 없다. 연산 결과로 **BIGINT** 타입 정수(64비트 정수)를 반환한다. 이때, 하나 이상의 피연산자가 **NULL**이면 **NULL**을 반환한다.

아래는 CUBRID가 지원하는 비트 연산자의 종류에 관한 표이다.

CUBRID가 지원하는 비트 연산자

비트 연산자 설명	조건식	리턴 값
& 비트 단위로 AND 연산을 수행하고, BIGINT 정수를 반환한다.	17 & 3	1
비트 단위로 OR 연산을 수행하고, BIGINT 정수를 반환한다.	17 3	19
^ 비트 단위로 XOR 연산을 수행하고, BIGINT 정수를 반환한다.	17 ^ 3	18
~ 단항 연산자이며, 피연산자의 비트를 역으로 전환(INVERT)하는 보수 연산을 수행하고, BIGINT 정수를 반환한다.	~17	-18
<< 왼쪽 피연산자의 비트를 오른쪽 피연산자만큼 왼쪽으로 이동시키는 연산을 수행하고, BIGINT 정수를 반환한다.	17 << 3	136
>> 왼쪽 피연산자의 비트를 오른쪽 피연산자만큼 오른쪽으로 이동시키는 연산을 수행하고, BIGINT 정수를 반환한다.	17 >> 3	2

BIT_AND 함수

설명

집계 함수로서, *expr*의 모든 비트에 대해 비트 단위 **AND** 연산을 수행한다. 리턴 값은 **BIGINT** 타입이다. 조건절을 만족하는 행이 없는 경우, **NULL**을 반환한다.

구문

BIT_AND(*expr*)

- expr*: 정수 타입의 임의의 연산식이다.

예제

```
CREATE TABLE bit_tbl(id int);
INSERT INTO bit_tbl VALUES (1), (2), (3), (4), (5);
SELECT 1&3&5, BIT_AND(id) FROM bit_tbl WHERE id in(1,3,5);
;xr
```

=== <Result of SELECT Command in Line 1> ===

1&3&5	bit and(id)
1	1

BIT_OR 함수

설명

집계 함수로서, *expr*의 모든 비트에 대해 비트 단위 **OR** 연산을 수행한다. 리턴 값은 **BIGINT** 타입이다. 조건절을 만족하는 행이 없는 경우, **NULL**을 반환한다.

구문

BIT_OR(*expr*)

- expr*: 정수 타입의 임의의 연산식이다.

예제

```
SELECT 1|3|5, BIT_OR(id) FROM bit_tbl WHERE id in(1,3,5);
```

=== <Result of SELECT Command in Line 1> ===

1 3 5	bit or(id)
7	7

BIT_XOR 함수

설명

집계 함수로서, *expr*의 모든 비트에 대해 비트 단위 **XOR** 연산을 수행한다. 리턴 값은 **BIGINT** 타입이다. 조건절을 만족하는 행이 없는 경우, **NULL**을 반환한다.

구문

BIT_XOR (*expr*)

- *expr*: 정수 타입의 임의의 연산식이다.

예제

```
SELECT 1^2^3, BIT_XOR(id) FROM bit_tbl WHERE id in(1,3,5);
```

=== <Result of SELECT Command in Line 1> ===

1^3^5	bit xor(id)
7	7

BIT_COUNT 함수

설명

*expr*의 모든 비트 중 1로 설정된 비트의 개수를 반환하는 함수이며, 집계 함수는 아니다. 리턴 값은 **BIGINT** 타입이다.

구문

BIT_COUNT (*expr*)

- *expr*: 정수 타입의 임의의 연산식이다.

예제

```
SELECT BIT COUNT(id) FROM bit_tbl WHERE id in(1,3,5);
```

=== <Result of SELECT Command in Line 1> ===

bit_count(id)
1
2
2

스트링 함수와 연산자

병합 연산자

설명

병합 연산자는 피연산자로 문자열 또는 비트열 데이터 타입이 지정되며, 병합(concatenation)된 스트링을 반환한다. 문자열 데이터의 병합 연산자로 덧셈 기호(+)와 두 개의 파이프 기호(||)가 제공된다. 피연산자로 **NULL**이 지정된 경우는 **NULL** 값이 반환된다.

SQL 구문 관련 파라미터인 **pipes_as_concat** 파라미터가 no이면, 이중 파이프 기호(||)가 부울린(Boolean) OR 연산자로 해석되므로, + 연산자 또는 **CONCAT** 함수를 사용하여 스트링을 병합하는 것이 좋다.

구문

```
concat operand1 + concat operand1
concat_operand2 || concat_operand2
concat_operand1 :
• bit string
• NULL

concat_operand2 :
• bit string
• character string
• NULL
```

- *concat_operand1*: 병합 후 왼쪽에 위치할 스트링이며, 문자열 또는 비트열 타입이다.
- *concat_operand2*: 병합 후 오른쪽에 위치할 스트링이며, 문자열 또는 비트열 타입이다.

예제

```
SELECT 'CUBRID' || ',' + '2008';
=====
'CUBRID,2008'

SELECT 'cubrid' || ',' || B'0010' || B'0000' || B'0000' || B'1000' FROM db root;
=====
'cubrid,2008'

SELECT ((EXTRACT(YEAR FROM SYS_TIMESTAMP)) || (EXTRACT(MONTH FROM SYS_TIMESTAMP)));
=====
'200812'

SELECT 'CUBRID' || ',' + NULL;
=====
NULL
```

BIT_LENGTH 함수

설명

BIT_LENGTH 함수는 문자열 또는 비트열의 길이(bit)를 정수값으로 반환한다. 단, 문자열의 경우 데이터 입력 환경의 문자 세트(character set)에 따라 한 문자가 차지하는 바이트 수가 다르므로, **BIT_LENGTH** 함수의 리턴 값 역시 문자 세트에 따라 다를 수 있다(예: EUC-KR: 2*8 bits). CUBRID가 지원하는 문자 세트에 관한 상세한 설명은 [문자 스트링 데이터 타입](#)을 참고한다.

구문

```
BIT_LENGTH ( string )

string :
• bit string
• character string
• NULL
```

- *string*: 비트 단위로 길이를 구할 문자열 또는 비트열을 지정한다. **NULL**이 지정된 경우는 **NULL** 값이 반환된다.

예제

```
SELECT BIT_LENGTH('');
=====
0
```

```

SELECT BIT_LENGTH('CUBRID');
=====
                        48

SELECT BIT_LENGTH('큐브리드');
=====
                        64

SELECT BIT_LENGTH(B'010101010');
=====
                        9

CREATE TABLE bit_length_tbl (char_1 CHAR, char_2 CHAR(5), varchar_1 VARCHAR, bit_var_1 BIT
VARYING);
INSERT INTO bit_length_tbl VALUES(' ', ' ', ' ', B''); --Length of empty string
INSERT INTO bit_length_tbl VALUES('a', 'a', 'a', B'010101010'); --English character
INSERT INTO bit_length_tbl VALUES(NULL, '큐', '큐', B'010101010'); --Korean character and
NULL
INSERT INTO bit_length_tbl VALUES(' ', ' 큐', ' 큐', B'010101010'); --Korean character and
space

SELECT BIT_LENGTH(char_1), BIT_LENGTH(char_2), BIT_LENGTH(varchar_1), BIT_LENGTH(bit_var_1)
FROM bit_length_tbl;

=== <Result of SELECT Command in Line 15> ===

=====
8                        40                        0                        0
8                        40                        8                        9
NULL                     40                        16                       9
8                        40                        24                       9

```

CHAR_LENGTH, CHARACTER_LENGTH, LENGTHB, LENGTH 함수

설명

CHAR_LENGTH, CHARACTER_LENGTH, LENGTHB, LENGTH 함수는 동일하다.

문자열의 길이(bytes)를 정수 값으로 반환하며, 리턴 값은 문자 세트에 따라 다를 수 있다(예: EUC-KR: 2바이트). CUBRID가 지원하는 문자 세트에 관한 상세한 설명은 [문자 스트링 데이터 타입](#)을 참고한다.

구문

```

CHAR_LENGTH( string )
CHARACTER_LENGTH( string )
LENGTHB( string )
LENGTH( string )

```

string :
• character string
• **NULL**

- *string*: 바이트 단위로 길이를 구할 문자열을 지정한다. **NULL**이 지정된 경우는 **NULL** 값이 반환된다.

참고 사항

- 문자열 내에 포함된 공백 문자(space)의 길이는 1바이트이다.
- 멀티바이트 스트링의 경우, 데이터 입력 환경에서의 문자 세트에 따라 단위 문자의 길이가 2바이트 또는 3바이트로 계산된다.

- 공백 문자를 표현하기 위한 빈 따옴표('')의 길이는 0이다. 단, **CHAR(*n*)** 타입에서는 공백 문자의 길이가 *n*이고, *n*이 생략되는 경우 1로 처리되므로 주의한다.

예제

```
--character set is euc-kr for Korean characters
SELECT LENGTH('');
=====
0

SELECT LENGTH('CUBRID');
=====
6

SELECT LENGTH('큐브리드');
=====
8

CREATE TABLE length_tbl (char 1 CHAR, char 2 CHAR(5), varchar 1 VARCHAR, varchar 2
VARCHAR);
INSERT INTO length_tbl VALUES(' ', ' ', ' ', ' '); --Length of empty string
INSERT INTO length_tbl VALUES('a', 'a', 'a', 'a'); --English character
INSERT INTO length_tbl VALUES(NULL, '큐', '큐', '큐'); --Korean character and NULL
INSERT INTO length_tbl VALUES(' ', ' 큐', ' 큐', ' 큐'); --Korean character and space

SELECT LENGTH(char 1), LENGTH(char 2), LENGTH(varchar 1), LENGTH(varchar 2) FROM
length_tbl;

=== <Result of SELECT Command in Line 1> ===

=====
1          5          0          0
1          5          1          1
NULL       5          2          2
1          5          3          3
```

CHR 함수

설명

CHR 함수는 인자로 지정된 연산식의 리턴 값에 대응하는 문자를 반환하는 함수이다. 문자 코드 범위를 초과하면 '0'을 반환한다.

구문

```
CHR( number_operand )
```

- number_operand*: 수치값을 반환하는 임의의 연산식을 지정한다.

예제

```
SELECT CHR(68) || CHR(68-2);

=== <Result of SELECT Command in Line 1> ===

chr(68)|| chr(68-2)
=====
'DB'
```

CONCAT 함수

설명

CONCAT 함수는 두 개 이상의 인자가 지정되며, 모든 인자 값을 연결한 스트링을 결과로 반환한다. 지정 가능한 인자의 개수는 제한이 없으며, 스트링 타입이 아닌 인자가 지정되는 경우 자동으로 타입 변환이 수행된다. 인자 중에 **NULL**이 포함되면 결과로 **NULL**을 반환한다.

인자로 지정된 스트링 사이에 구분자(separator)를 삽입하여 연결하려면, [CONCAT_WS 함수](#)를 사용한다.

구문

```
CONCAT( string1, string2 [,string3 [, ... [, stringN]...]])

string :
• character string
• NULL
```

예제

```
SELECT CONCAT('CUBRID', '2008' , 'R3.0');

=====
'CUBRID2008R3.0'

--it returns null when null is specified for one of parameters
SELECT CONCAT('CUBRID', '2008' , 'R3.0', NULL);

=====
NULL

--it converts number types and then returns concatenated strings
SELECT CONCAT(2008, 3.0);

=====
'20083.0'
```

CONCAT_WS 함수

설명

CONCAT_WS 함수는 두 개 이상의 인자가 지정되며, 첫 번째 인자 값을 구분자로 이용하여 나머지 인자 값을 연결한 스트링을 결과로 반환한다. 지정 가능한 인자의 개수에는 제한이 없으며, 스트링 타입이 아닌 인자가 지정되는 경우 자동으로 타입 변환이 수행된다. 만약, 구분자로 **NULL**이 지정되면 **NULL**을 반환하고, 구분자 다음에 위치하는 나머지 인자에 **NULL**이 지정되면 이를 무시하고 문자열을 반환한다.

구문

```
CONCAT_WS( string1, string2 [,string3 [, ... [, stringN]...]])

string :
• character string
• NULL
```

예제

```
SELECT CONCAT_WS(' ', 'CUBRID', '2008' , 'R3.0');
```

```
=====
'CUBRID 2008 R3.0'

--it returns strings even if null is specified for one of parameters
SELECT CONCAT WS(' ', 'CUBRID', '2008', NULL, 'R3.0');
=====
'CUBRID 2008 R3.0'

--it converts number types and then returns concatenated strings with separator
SELECT CONCAT WS(' ',2008, 3.0);
=====
'2008 3.0'
```

FIELD 함수

설명

FIELD 함수는 *string1*, *string2* 등의 인자 중 *search_string*과 동일한 인자의 위치 인덱스 값(포지션)을 반환한다. *search_string*과 동일한 인자가 없으면 0을 반환한다. *search_string*이 **NULL**이면 다른 인자와 비교 연산을 수행할 수 없으므로 0을 반환한다.

FIELD 함수에서 지정된 모든 인자가 스트링 타입이면 스트링 비교 연산을 수행하고, 모두 수치 타입이면 수치 비교 연산을 수행한다. 어느 한 인자의 타입이 나머지와 다른 경우, 모든 인자를 첫 번째 인자의 타입으로 변환하여 비교 연산을 수행한다. 각 인자와의 비교 연산 도중 타입 변환에 실패하면 비교 연산의 결과를 **FALSE**로 간주하고, 나머지 연산을 계속 진행한다.

구문

```
FIELD( search_string, string1 [,string2 [, ... [, stringN]...]])

string :
• character string
• NULL
```

예제

```
SELECT FIELD('abc', 'a', 'ab', 'abc', 'abcd', 'abcde');

=====
3

--it returns 0 when no same string is found in the list
SELECT FIELD('abc', 'a', 'ab', NULL);

=====
0

--it returns 0 when null is specified in the first parameter
SELECT FIELD(NULL, 'a', 'ab', NULL);

=====
0

SELECT FIELD('123', 1, 12, 123.0, 1234, 12345);

=====
0

SELECT FIELD(123, 1, 12, '123.0', 1234, 12345);

=====
3
```

INSTR 함수

설명

INSTR 함수는 **POSITION** 함수와 유사하게 문자열 *string* 내에서 문자열 *substring*의 위치를 반환한다. 단, **INSTR** 함수는 *substring*의 검색을 시작할 위치를 지정할 수 있으므로 중복된 *substring*을 검색할 수 있다.

문자 단위가 아닌 바이트 단위로 위치를 반환한다는 점을 주의한다. 멀티바이트 문자 세트에서는 한 문자를 표현하는 바이트 수가 다르므로 반환되는 결과 값이 다를 수 있다.

구문

```
INSTR( string , substring [, position] )

string , substring :
• character string
• NULL
position :
• INT
• NULL
```

- *string* : 입력 문자열을 지정한다.
- *substring* : 위치를 반환할 문자열을 지정한다.
- *position* : 선택 사항으로 탐색을 시작할 *string*의 위치를 나타내며, 바이트 단위로 지정된다. 이 인자가 생략되면 디폴트 값인 1이 적용된다. *string*의 첫 번째 위치는 1로 지정된다. 값이 음수이면 *string*의 끝에서부터 지정된 값만큼 떨어진 위치에서 역방향으로 *string*을 탐색한다.

예제

```
--character set is euc-kr for Korean characters
--it returns position of the first 'b'
SELECT INSTR ('12345abcdeabcde','b');

=====
7

-- it returns position of the first '나' on double byte charset
SELECT INSTR ('12345가나다라마가나다라마', '나' );
=====
8

-- it returns position of the second 'ㄴ' on double byte charset
SELECT INSTR ('12345가나다라마가나다라마', '나', 16 );
=====
18

--it returns position of the 'b' searching from the 8th position
SELECT INSTR ('12345abcdeabcde','b', 8);

=====
12

--it returns position of the 'b' searching backwardly from the end
SELECT INSTR ('12345abcdeabcde','b', -1);

=====
12

--it returns position of the 'b' searching backwardly from a specified position
SELECT INSTR ('12345abcdeabcde','b', -8);
```

```
=====
7
```

LCASE, LOWER 함수

설명

LCASE 함수와 **LOWER** 함수는 동일하며, 문자열에 포함된 대문자를 소문자로 변환한다. 단, CUBRID가 지원하지 않는 문자 세트에서는 정상 동작하지 않을 수 있으므로 주의한다. CUBRID가 지원하는 문자 세트에 관한 상세한 설명은 [문자 스트링 데이터 타입](#)을 참고한다.

구문

```
LCASE ( string )
LOWER ( string )

string :
• character string
• NULL
```

- *string*: 소문자로 변환할 문자열을 지정한다. 값이 **NULL**이면 결과는 **NULL**이 반환된다.

예제

```
SELECT LOWER('');
=====
''

SELECT LOWER(NULL);
=====
NULL

SELECT LOWER('Cubrid');
=====
'cubrid'
```

LEFT 함수

설명

LEFT 함수는 *string*의 가장 왼쪽에서부터 *length* 개의 문자를 반환한다. 어느 하나의 인자가 **NULL**인 경우 **NULL**이 반환되고, *string* 길이보다 큰 값이나 음수가 *length*로 지정되면 스트링 전체를 반환한다.

문자열의 가장 오른쪽에서부터 *length* 길이의 문자열을 추출하려면 [RIGHT 함수](#)를 사용한다.

구문

```
LEFT( string , length )

string :
• character string
• NULL

length :
• INT
• NULL
```

예제

```
SELECT LEFT('CUBRID', 3);
=====
'CUB'
=====

SELECT LEFT('CUBRID', 10);
=====
'CUBRID'
=====
```

LOCATE 함수

설명

LOCATE 함수는 문자열 *string* 내에서 문자열 *substring*의 위치 인덱스 값을 반환한다. 세 번째 인자 *position*은 생략할 수 있으며, 이 인자가 지정되면 해당 위치에서부터 *substring*을 검색하여 처음 검색한 위치 인덱스 값을 반환한다. *substring*이 *string* 내에서 검색되지 않으면 0을 반환한다.

LOCATE 함수는 [POSITION 함수](#)와 유사하게 동작하지만, 비트열에 대해서는 **LOCATE** 함수를 적용할 수 없다.

구문

```
LOCATE ( substring, string [, position] )

string :
• character string
• NULL
```

예제

```
--it returns 1 when substring is empty space
SELECT LOCATE ('', '12345abcdeabcde');
=====
1

--it returns position of the first 'abc'
SELECT LOCATE ('abc', '12345abcdeabcde');
=====
6

--it returns position of the second 'abc'
SELECT LOCATE ('abc', '12345abcdeabcde', 8);
=====
11

--it returns 0 when no substring found in the string
SELECT LOCATE ('ABC', '12345abcdeabcde');
=====
0
```

LPAD 함수

설명

LPAD 함수는 문자열이 일정 길이(byte)가 될 때까지 왼쪽에 특정 문자를 덧붙인다.

구문

```
LPAD( char1, n, [, char2 ] )
```

char1 :

- character string
- string valued column
- **NULL**

n :

- integer
- **NULL**

char2 :

- character string
- **NULL**

- *char1*: 덧붙이는 대상 문자열을 지정한다. *char1*의 길이보다 작은 *n*이 지정되면, 패딩을 수행하지 않고 *char1*을 길이 *n*으로 잘라내어 반환한다. 단, 멀티바이트 문자 세트 환경에서는 한 문자를 2바이트 또는 3바이트로 처리하는데, *n* 값에 의해 한 문자를 표현하는 첫 번째 바이트까지 *char1*을 잘라내는 경우, 마지막 문자를 정상 표현할 수 없으므로 마지막 바이트를 제거하고 왼쪽에 공백 문자 하나(1바이트)를 덧붙인다. 값이 **NULL**이면 결과는 **NULL**이 반환된다.
- *n*: *char1*의 전체 길이를 바이트 단위로 지정한다. 단, 멀티바이트 문자 세트 환경에서는 문자열의 개수와 문자열의 길이가 다를 수 있으므로 주의한다. 값이 **NULL**이면 결과는 **NULL**이 반환된다.
- *char2*: *char1*의 길이가 *n*이 될 때까지 왼쪽에 덧붙일 문자열을 지정한다. 이를 지정하지 않으면 공백 문자(' ')가 *char2*의 디폴트 값으로 사용된다. 값이 **NULL**이면 결과는 **NULL**이 반환된다.

예제

```
--character set is euc-kr for Korean characters

--it returns only 3 characters if not enough length is specified
SELECT LPAD ('CUBRID', 3, '?');

=====
' CUB'

--on multi-byte charset, it returns the first character only with a left padded space
SELECT LPAD ('큐브리드', 3, '?');

=====
' 큐'

--padding spaces on the left till char length is 10
SELECT LPAD ('CUBRID', 10);

=====
'      CUBRID'

--padding specific characters on the left till char length is 10
SELECT LPAD ('CUBRID', 10, '?');

=====
'????CUBRID'

--padding specific characters on the left till char length is 10
SELECT LPAD ('큐브리드', 10, '?');

=====
'??큐브리드'

--padding 4 characters on the left
```

```
SELECT LPAD ('큐브리드', LENGTH('큐브리드')+4, '?');

=====
'????큐브리드'
```

LTRIM 함수

설명

LTRIM 함수는 문자열의 왼쪽(앞 부분)에 위치한 특정 문자를 제거한다.

구문

```
LTRIM( string [, trim_string])

string :
• character string
• string valued column
• NULL

trim string :
• character string
• NULL
```

- *string*: 트리밍할 문자열 또는 문자열 타입의 컬럼을 입력하며, 이 값이 **NULL**이면 결과는 **NULL**이 반환된다.
- *trim_string*: *string*의 왼쪽에서 제거하고자 하는 특정 문자열을 지정할 수 있으며, 이를 지정하지 않으면 공백 문자(' ')가 자동으로 지정되어 대상 문자열의 왼쪽에 위치한 공백이 제거된다.

예제

```
--trimming spaces on the left
SELECT LTRIM ('    Olympic    ');

=====
'Olympic    '

--If NULL is specified, it returns NULL
SELECT LTRIM ('iiiiOlympiciiii', NULL);

=====
NULL

-- trimming specific strings on the left
SELECT LTRIM ('iiiiOlympiciiii', 'i');

=====
'Olympiciiii'
```

MID 함수

설명

MID 함수는 문자열 *string* 내의 *position* 위치로부터 *substring_length* 길이의 문자열을 추출하여 반환한다.

만약, *position* 값으로 음수가 지정되면, 문자열의 끝에서부터 역방향으로 위치를 산정한다.

*substring_length*는 생략할 수 없으며, 음수가 지정되는 경우 이를 0으로 간주하여 공백 스트링을 반환한다.

MID 함수는 [SUBSTR 함수](#)와 유사하게 동작하나, 비트 스트링에 대해서는 적용할 수 없고, *substring_length* 인자를 생략할 수 없으며, *substring_length*에 음수가 지정되면 공백 스트링을 반환한다는 차이점이 있다.

구문

```
MID( string, position, substring_length )
```

string :

- character string
- **NULL**

position :

- integer
- **NULL**

substring_length :

- integer
- **NULL**

- *string*: 입력 문자열을 지정한다. 입력 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *position*: 문자열을 추출할 시작 위치를 지정한다. 첫 번째 문자의 위치는 1이며, 0으로 지정되더라도 1로 간주된다. 입력 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *substring_length*: 추출할 문자열의 길이를 지정한다. 0 또는 음수가 지정되는 경우 공백 스트링이 반환되고, 입력 값이 **NULL**이면 결과로 **NULL**이 반환된다.

예제

```
CREATE TABLE mid_tbl(a VARCHAR);
INSERT INTO mid_tbl VALUES('12345abcdeabcde');

--it returns empty string when substring_length is 0
SELECT MID(a, 6, 0), SUBSTR(a, 6, 0), SUBSTRING(a, 6, 0) FROM mid_tbl;

=====
''          ''          ''

--it returns 4-length substrings counting from the 6th position
SELECT MID(a, 6, 4), SUBSTR(a, 6, 4), SUBSTRING(a, 6, 4) FROM mid_tbl;

=====
'abcd'      'abcd'      'abcd'

--it returns a empty string when substring length < 0
SELECT MID(a, 6, -4), SUBSTR(a, 6, -4), SUBSTRING(a, 6, -4) FROM mid_tbl;

=====
''          NULL        'abcdeabcde'

--it returns 4-length substrings at 6th position counting backward from the end
SELECT MID(a, -6, 4), SUBSTR(a, -6, 4), SUBSTRING(a, -6, 4) FROM mid_tbl;

=====
'eabc'      'eabc'      '1234'
```

OCTET_LENGTH 함수

설명

OCTET_LENGTH 함수는 문자열 또는 비트열의 길이(byte)를 정수로 반환한다. 따라서, 비트열의 길이가 8비트인 경우에는 1(byte)을 반환하지만, 9비트인 경우에는 2(byte)를 반환한다.

구문

```
OCTET_LENGTH ( string )
```

```
string :  
• bit string  
• character string  
• NULL
```

- *string*: 바이트 단위로 길이를 구할 문자열 또는 비트열을 지정한다. **NULL**이 지정된 경우는 **NULL** 값이 반환된다.

예제

```
--character set is euc-kr for Korean characters  
  
SELECT OCTET_LENGTH('');  
=====  
0  
  
SELECT OCTET_LENGTH('CUBRID');  
=====  
6  
  
SELECT OCTET_LENGTH('큐브리드');  
=====  
8  
  
SELECT OCTET_LENGTH(B'010101010');  
=====  
2  
  
CREATE TABLE octet_length_tbl (char_1 CHAR, char_2 CHAR(5), varchar_1 VARCHAR, bit_var_1  
BIT VARYING);  
INSERT INTO octet_length_tbl VALUES(' ', ' ', ' ', B''); --Length of empty string  
INSERT INTO octet_length_tbl VALUES('a', 'a', 'a', B'010101010'); --English character  
INSERT INTO octet_length_tbl VALUES(NULL, '큐', '큐', B'010101010'); --Korean character and  
NULL  
INSERT INTO octet_length_tbl VALUES(' ', ' 큐', ' 큐', B'010101010'); --Korean character  
and space  
  
SELECT OCTET_LENGTH(char_1), OCTET_LENGTH(char_2), OCTET_LENGTH(varchar_1),  
OCTET_LENGTH(bit_var_1) FROM octet_length_tbl;  
=== <Result of SELECT Command in Line 16> ===  
  
=====
```

1	5	0	0
1	5	1	2
NULL	5	2	2
1	5	3	2

POSITION 함수

설명

POSITION 함수는 문자열 *string* 내에서 문자열 *substring*의 위치를 반환한다. 문자 단위가 아닌 바이트 단위로 위치를 반환한다는 점을 주의한다. 멀티바이트 문자 세트에서는 한 문자를 표현하는 바이트 수가 다르므로 반환되는 결과 값이 다를 수 있다.

이 함수의 인자로 문자열 또는 비트열을 반환하는 임의의 연산식을 지정할 수 있으며, 리턴 값은 0 이상의 정수이다. 문자열에 대해서는 바이트 단위로 위치 값을 반환하고, 비트열에 대해서는 비트 단위로 위치 값을 반환한다.

POSITION 함수는 가끔 다른 함수와 연결되어서 사용된다. 예를 들어, 특정 스트링에서 일부 스트링을 추출하고 싶은 경우에 **POSITION** 함수의 결과를 **SUBSTRING** 함수의 입력으로 사용할 수 있다.

구문

```
POSITION ( substring IN string )
```

```
substring :  
• bit string  
• character string  
• NULL
```

- *substring*: 위치를 반환할 문자열을 지정한다. 값이 공백 스트링이면 1이 반환된다. **NULL**이면 **NULL**이 반환된다.

예제

```
--character set is euc-kr for Korean characters  
  
--it returns 1 when substring is empty space  
SELECT POSITION ('' IN '12345abcdeabcde');  
=====  
1  
  
--it returns position of the first 'b'  
SELECT POSITION ('b' IN '12345abcdeabcde');  
=====  
7  
  
-- it returns position of the first '나' on double byte charset  
SELECT POSITION ('나' IN '12345가나다라마가나다라마');  
=====  
8  
  
--it returns 0 when no substring found in the string  
SELECT POSITION ('f' IN '12345abcdeabcde');  
=====  
0  
  
SELECT POSITION (B'1' IN B'000011110000');  
=====  
5
```

REPLACE 함수

설명

REPLACE 함수는 주어진 문자열 *string* 내에서 문자열 *search_string*을 검색하여 이를 문자열 *replacement_string*으로 대체한다. 이때, 대체할 문자열 *replacement_string*이 생략되면 *string* 내에서 검색된 *search_string*이 모두 제거된다. 만약, 인자에 **NULL**이 지정되면, **NULL**이 반환된다.

구문

```
REPLACE( string, search_string [, replacement_string ] )
```

```
string :
• character string
• NULL

search string :
• character string
• NULL

replacement string :
• character string
• NULL
```

- *string*: 원본 문자열을 지정한다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *search_string*: 검색할 문자열을 지정한다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *replacement_string*: *search_string*을 대체할 문자열을 지정한다. 값이 생략되면 *string*에서 *search_string*을 제거하여 반환한다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.

예제

```
--it returns NULL when an argument is specified with NULL value
SELECT REPLACE('12345abcdeabcde','abcde',NULL);

=====
NULL

--not only the first substring but all substrings into 'ABCDE' are replaced
SELECT REPLACE('12345abcdeabcde','abcde','ABCDE');

=====
'12345ABCDEABCDE'

--it removes all of substrings when replace string is omitted
SELECT REPLACE('12345abcdeabcde','abcde');

=====
'12345'
```

REVERSE 함수

설명

REVERSE 함수는 문자열 *string*을 역순으로 변환한 후 반환한다.

구문

```
REVERSE( string )

string :
• character string
• NULL
```

- *string*: 입력 문자열을 지정한다. 입력 값이 공백 스트링이면 공백 스트링을 반환하고, **NULL**이면 **NULL**을 반환한다.

예제

```
SELECT REVERSE('CUBRID');

=====
'DIRBUC'
```

RIGHT 함수

설명

RIGHT 함수는 *string*의 가장 오른쪽에서부터 *length* 개의 문자를 반환한다. 어느 하나의 인자가 **NULL**인 경우 **NULL**이 반환되고, *string* 길이보다 큰 값이나 음수가 *length*로 지정되면 문자열 전체를 반환한다.

문자열의 가장 왼쪽에서부터 *length* 길이의 문자열을 추출하려면 [LEFT 함수](#)를 사용한다.

구문

```
RIGHT( string , length )
```

string :

- *character string*
- **NULL**

length :

- *INT*
- **NULL**

예제

```
SELECT RIGHT('CUBRID', 3);
=====
      'RID'
=====

SELECT RIGHT ('CUBRID', 10);
=====
'CUBRID'
=====
```

RPAD 함수

설명

RPAD 함수는 문자열이 일정 길이(byte)가 될때까지 오른쪽에 특정 문자를 덧붙인다.

구문

```
RPAD( char1, n, [, char2 ] )
```

char1 :

- *character string*
- *string valued column*
- **NULL**

n :

- *integer*
- **NULL**

char2 :

- *character string*
- **NULL**

- *char1*: 덧붙이는 대상 문자열을 지정한다. *char1*의 길이보다 작은 *n*이 지정되면, 패딩을 수행하지 않고 *char1*을 길이 *n*으로 잘라내어 반환한다. 단, 멀티바이트 문자 세트 환경에서는 한 문자를 2바이트 또는 3바이트로 처리하는데, *n* 값에 의해 한 문자를 표현하는 첫 번째 바이트까지 *char1*을 잘라내는 경우,

마지막 문자를 정상 표현할 수 없으므로 마지막 바이트를 제거하고 오른쪽에 공백 문자 하나(1바이트)를 덧붙인다. 값이 **NULL**이면 결과는 **NULL**이 반환된다.

- *n*: *char1*의 전체 길이를 바이트 단위로 지정한다. 단, 멀티바이트 문자 세트 환경에서는 문자열의 개수와 문자열의 길이가 다를 수 있으므로 주의한다. 값이 **NULL**이면 결과는 **NULL**이 반환된다.
- *char2*: *char1*의 길이가 *n*이 될 때까지 오른쪽에 덧붙일 문자열을 지정한다. 이를 지정하지 않으면 공백 문자(' ')가 *char2*의 디폴트 값으로 사용된다. 값이 **NULL**이면 결과는 **NULL**이 반환된다.

예제

```
--character set is euc-kr for Korean characters

--it returns only 3 characters if not enough length is specified
SELECT RPAD ('CUBRID', 3, '?');

=====
'CUB'

--on multi-byte charset, it returns the first character only with a right-padded space
SELECT RPAD ('큐브리드', 3, '?');
=====
'큐 '

--padding spaces on the right till char_length is 10
SELECT RPAD ('CUBRID', 10);

=====
'CUBRID      '

--padding specific characters on the right till char_length is 10
SELECT RPAD ('CUBRID', 10, '?');
=====
'CUBRID????'

--padding specific characters on the right till char_length is 10
SELECT RPAD ('큐브리드', 10, '?');
=====
'큐브리드??'

--padding 4 characters on the right
SELECT RPAD ('큐브리드', LENGTH('큐브리드')+4, '?');
=====
'큐브리드????'
```

RTRIM 함수

설명

RTRIM 함수는 문자열의 오른쪽(뒷 부분)에 위치한 특정 문자를 제거한다.

구문

```
RTRIM( string [, trim_string])

string :
• character string
• string valued column
• NULL

trim_string :
• character string
```


• NULL

- *string*: 트리밍할 문자열 또는 문자열 타입의 컬럼을 입력하며, 이 값이 **NULL**이면 결과는 **NULL**이 반환된다.
- *trim_string*: *string*의 오른쪽에서 제거하고자 하는 특정 문자열을 지정할 수 있으며, 이를 지정하지 않으면 공백 문자(' ')가 자동으로 지정되어 대상 문자열의 오른쪽에 위치한 공백이 제거된다.

예제

```
SELECT RTRIM ('      Olympic      ');
=====
'      Olympic'

--If NULL is specified, it returns NULL
SELECT RTRIM ('iiiiOlympiciiii', NULL);
=====
NULL

-- trimming specific strings on the right
SELECT RTRIM ('iiiiOlympiciiii', 'i');
=====
'iiiiOlympic'
```

STRCMP 함수

설명

STRCMP 함수는 두 개의 문자열 *string1*, *string2*을 비교하여 동일하면 0을 반환하고, *string1*이 더 크면 1을 반환하고, *string1*이 더 작은 경우에는 -1을 반환한다. 어느 하나의 인자가 **NULL**이면 결과는 **NULL**을 반환한다.

구문

```
STRCMP( string1 , string2 )

string :
• character string
• NULL
```

예제

```
SELECT STRCMP('abc', 'abc');
=====
0
SELECT STRCMP ('acc', 'abc');
=====
1

--STRCMP works case-insensitively
SELECT STRCMP ('ABC','abc');
=====
0
```

SUBSTR 함수

설명

SUBSTR 함수는 문자열 *string* 내의 *position* 위치로부터 *substring_length* 길이의 문자열을 추출하여 반환한다. 만약, *position* 값으로 음수가 지정되면, 문자열의 끝에서부터 역방향으로 위치를 산정한다. 또한, *substring_length*가 생략되는 경우, 주어진 *position* 위치로부터 마지막까지 문자열을 추출하여 반환한다.

문자 단위가 아닌 바이트 단위로 시작 위치와 문자열의 길이를 산정한다는 점을 주의한다. 멀티바이트 문자 세트에서는 한 문자를 표현하는 바이트 수를 고려하여 인자를 지정해야 한다.

구문

```
SUBSTR( string, position [, substring_length])
```

string :

- *character string*
- *bit string*
- **NULL**

position :

- *integer*
- **NULL**

substring_length :

- *integer*

- *string*: 입력 문자열을 지정한다. 입력 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *position*: 문자열을 추출할 시작 위치를 바이트 단위로 지정한다. 첫 번째 문자의 위치는 1이며, 0으로 지정되더라도 1로 간주된다. *string* 길이보다 큰 값을 지정하거나 **NULL**을 지정하면 결과로 **NULL**이 반환된다.
- *substring_length*: 추출할 문자열의 길이를 바이트 단위로 지정한다. 이 인자가 생략되면 *position* 위치로부터 마지막까지 문자열을 추출한다. 이 인자의 값으로 **NULL**이 지정될 수 없으며, 0이 지정되는 경우 공백 스트링이 반환되고, 음수가 지정되는 경우 **NULL**이 반환된다.

예제

```
--character set is euc-kr for Korean characters

--it returns empty string when substring_length is 0
SELECT SUBSTR('12345abcdeabcde',6, 0);

=====
''

--it returns 4-length substrings counting from the position
SELECT SUBSTR('12345abcdeabcde', 6, 4), SUBSTR('12345abcdeabcde', -6, 4);

=====
'abcd'          'eabc'

--it returns substrings counting from the position to the end
SELECT SUBSTR('12345abcdeabcde', 6), SUBSTR('12345abcdeabcde', -6);
=====
'abcdeabcde'    'eabcde'

-- it returns 4-length substrings counting from 16th position on double byte charset
SELECT SUBSTR ('12345가나다라마가나다라마', 16 , 4);
=====
```

'가나'

SUBSTRING 함수

설명

SUBSTRING 함수는 **SUBSTR** 함수와 유사하며, 문자열 *string* 내의 *position* 위치로부터 *substring_length* 길이의 문자열을 추출하여 반환한다.

position 값에 음수가 지정되면, **SUBSTRING** 함수는 문자열의 처음으로 검색 위치를 산정하고, **SUBSTR** 함수는 문자열의 끝에서부터 역방향으로 위치를 산정한다. *substring_length* 값에 음수가 지정되면,

SUBSTRING 함수는 해당 인자가 생략된 것으로 처리하지만, **SUBSTR** 함수는 **NULL**을 반환한다.

구문

```
SUBSTRING( string, position [, substring_length])
SUBSTRING( string FROM position [FOR substring_length] )

string :
• bit string
• character string
• NULL

position :
• integer
• NULL

substring length :
• integer
```

- *string*: 입력 문자열을 지정한다. 입력 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *position*: 문자열을 추출할 시작 위치를 바이트 단위로 지정한다. 0이나 음수가 지정되면, 첫 번째 문자의 위치인 1로 간주된다. *string* 길이보다 큰 값을 지정하면 공백 스트링이 반환되고, **NULL**을 지정하면 **NULL**이 반환된다.
- *substring_length*: 추출할 문자열의 길이를 바이트 단위로 지정한다. 이 인자가 생략되면 *position* 위치로부터 마지막까지 문자열을 추출한다. 이 인자의 값으로 **NULL**을 지정될 수 없으며, 0을 지정하면 공백 스트링이 반환되고, 음수를 지정하면 무시한다.

예제

```
SELECT SUBSTRING('12345abcdeabcde', -6 ,4), SUBSTR('12345abcdeabcde', -6 ,4);
=====
'1234'                                'eabc'

SELECT SUBSTRING('12345abcdeabcde', 16), SUBSTR('12345abcdeabcde', 16);
=====
' '                                    NULL

SELECT SUBSTRING('12345abcdeabcde', 6, -4), SUBSTR('12345abcdeabcde', 6, -4);
=====
'abcdeabcde'                          NULL
```

TRANSLATE 함수

설명

TRANSLATE 함수는 지정된 문자열 *string* 내에 문자열 *from_substring*에 지정된 문자가 존재한다면, 이를 *to_substring*에 지정된 문자로 대체한다. 이때, *from_substring*과 *to_substring*에 지정되는 문자의 순서에 따라 대응 관계를 가지며, *to_substring*과 1:1 대응되지 않는 나머지 *from_substring* 문자는 문자열 *string* 내에서 모두 제거된다. **REPLACE** 함수와 유사하게 동작하나, **TRANSLATE** 함수에서는 *to_substring* 인자를 생략할 수 없다.

구문

```
TRANSLATE( string, from_substring, to_substring )
```

string :

- character string
- **NULL**

from_substring :

- character string
- **NULL**

to_substring :

- character string
- **NULL**

- *string*: 원본 문자열을 지정한다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *from_substring*: 검색할 문자열을 지정한다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *to_substring*: *from_substring*에 지정된 문자열을 대체할 문자열을 지정하며, 생략할 수 없다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.

예제

```
--it returns NULL when an argument is specified with NULL value
SELECT TRANSLATE('12345abcdeabcde','abcde', NULL);

=====
NULL

--it translates 'a','b','c','d','e' into '1', '2', '3', '4', '5' respectively
SELECT TRANSLATE('12345abcdeabcde', 'abcde', '12345');

=====
'123451234512345'

--it translates 'a','b','c' into '1', '2', '3' respectively and removes 'd's and 'e's
SELECT TRANSLATE('12345abcdeabcde','abcde', '123');

=====
'12345123123'

--it removes 'a's,'b's,'c's,'d's, and 'e's in the string
SELECT TRANSLATE('12345abcdeabcde','abcde', '');
=====
'12345'

--it only translates 'a','b','c' into '3', '4', '5' respectively
SELECT TRANSLATE('12345abcdeabcde','ABabc', '12345');

=====
'12345345de345de'
```

TRIM 함수

설명

TRIM 함수는 문자열의 앞, 뒤 또는 앞뒤에 위치한 특정 문자들을 제거한다.

구문

```
TRIM ( [ [ LEADING | TRAILING | BOTH ] [ trim_string ] FROM ] string )
```

trim_string :

- *character string*
- **NULL**

string :

- *character string literal*
- *string valued column*
- **NULL**

- *trim_string*: 대상 문자열의 앞, 뒤 또는 앞뒤에서 제거하고자 하는 특정 문자열을 지정할 수 있으며, 이를 지정하지 않으면 공백 문자(' ')가 자동으로 지정되어 대상 문자열의 앞, 뒤 또는 앞뒤에 위치한 공백이 제거된다.
- *string*: 트리밍할 문자열 또는 문자열 타입의 컬럼을 입력하며, 이 값이 **NULL**이면 **NULL**이 반환된다.
- [**LEADING** | **TRAILING** | **BOTH**]: 대상 문자열의 어느 위치에서 지정된 문자열을 트리밍할 것인지를 옵션으로 명시할 수 있다. **LEADING**은 문자열의 앞 부분에서 트리밍을 수행하고, **TRAILING**은 문자열의 뒷 부분에서 트리밍을 수행하며, **BOTH**는 앞뒤에서 지정된 문자열을 트리밍한다. 옵션을 명시하지 않으면 디폴트 값은 **BOTH**이다.
- *trim_string*과 *string*의 문자열은 같은 문자 세트를 가져야 한다.

예제

```
--trimming NULL returns NULL
SELECT TRIM (NULL);

=====
NULL

--trimming spaces on both leading and trailing parts
SELECT TRIM ('    Olympic    ');

=====
'Olympic'

--trimming specific strings on both leading and trailing parts
SELECT TRIM ('i' FROM 'iiiiOlympiciiii');

=====
'Olympic'

--trimming specific strings on the leading part
SELECT TRIM (LEADING 'i' FROM 'iiiiOlympiciiii');

=====
'Olympiciiii'

--trimming specific strings on the trailing part
SELECT TRIM (TRAILING 'i' FROM 'iiiiOlympiciiii');

=====
'iiiiOlympic'
```

UCASE, UPPER 함수

설명

UCASE 함수와 **UPPER** 함수는 동일하며, 문자열에 포함된 소문자를 대문자로 변환한다. 단, CUBRID가 지원하지 않는 문자 세트에서는 정상 동작하지 않을 수 있으므로 주의한다. CUBRID가 지원하는 문자 세트에 관한 상세한 설명은 [문자 스트링 데이터 타입](#)을 참고한다.

구문

```
UCASE ( string )
UPPER ( string )

string :
• character string
• NULL
```

- *string*: 대문자로 변환할 문자열을 지정한다. 값이 **NULL**이면 결과는 **NULL**이 반환된다.

예제

```
SELECT UPPER('');

=====
''

SELECT UPPER(NULL);

=====
NULL

SELECT UPPER('Cubrid');

=====
'CUBRID'
```

수치 연산 함수

ABS 함수

설명

ABS 함수는 지정된 인자 값의 절대값을 반환하며, 리턴 값의 타입은 주어진 인자의 타입과 같다.

구문

```
ABS( number_operand )
```

- *number_operand*: 수치 값을 반환하는 임의의 연산식이다.

예제

```
--it returns the absolute value of the argument
SELECT ABS(12.3), ABS(-12.3), ABS(-12.3000), ABS(0.0);

=====
12.3          12.3          12.3000          .0
```

ACOS 함수

설명

ACOS 함수는 인자의 아크 코사인(arc cosine) 값을 반환한다. 즉, 코사인이 x 인 값을 라디안 단위로 반환하며, 리턴 값은 **DOUBLE** 타입이다. x 는 -1 이상 1 이하의 값이어야 하며, 그 외의 경우 에러를 반환한다.

구문

ACOS (x)

- x : 수치 값을 반환하는 임의의 연산식이다.

예제

```
SELECT ACOS(1), ACOS(0), ACOS(-1);
```

```
=====
0.000000000000000e+00    1.570796326794897e+00    3.141592653589793e+00
```

ASIN 함수

설명

ASIN 함수는 인자의 아크 사인(arc sine) 값을 반환한다. 즉, 사인이 x 인 값을 라디안 단위로 반환하며, 리턴 값은 **DOUBLE** 타입이다. x 는 -1 이상 1 이하의 값이어야 하며, 그 외의 경우 에러를 반환한다.

구문

ASIN (x)

- x : 수치 값을 반환하는 임의의 연산식이다.

예제

```
SELECT ASIN(1), ASIN(0), ASIN(-1);
```

```
=====
1.570796326794897e+00    0.000000000000000e+00    -1.570796326794897e+00
```

ATAN 함수

설명

ATAN 함수는 탄젠트가 x 인 값을 라디안 단위로 리턴한다. 인자 y 는 생략될 수 있으며, y 가 지정되는 경우 함수는 y/x 의 아크 탄젠트 값을 계산한다. 리턴 값은 **DOUBLE** 타입이다.

구문

ATAN ([y ,] x)

- x , y : 수치 값을 반환하는 임의의 연산식이다.

예제

```
SELECT ATAN(1), ATAN(-1), ATAN(1,-1);

=== <Result of SELECT Command in Line 1> ===

          atan(1)          atan(-1)          atan2(1, -1)
=====
7.853981633974483e-01    -7.853981633974483e-01    2.356194490192345e+00
```

ATAN2 함수

설명

ATAN2 함수는 y/x 의 아크 탄젠트 값을 라디안 단위로 리턴하며, [ATAN 함수](#)와 유사하게 동작한다. 인자 x , y 가 모두 지정되어야 한다. 리턴 값은 **DOUBLE** 타입이다.

구문

ATAN2 (y , x)

- x , y : 수치 값을 반환하는 임의의 연산식이다.

예제

```
SELECT ATAN2(1,1), ATAN2(-1,-1), ATAN2(Pi(),0);

=== <Result of SELECT Command in Line 1> ===

atan2(1, 1)          atan2(-1, -1)          atan2( pi(), 0)
=====
7.853981633974483e-01    -2.356194490192345e+00    1.570796326794897e+00
```

CEIL 함수

설명

CEIL 함수는 인자보다 크거나 같은 최소 정수 값을 인자의 타입으로 반환한다. 리턴 값은 *number_operand* 인자로 지정된 값의 유효 자릿수를 따른다.

구문

CEIL(*number_operand*)

- number_operand*: 수치 값을 반환하는 임의의 연산식이다.

예제

```
SELECT CEIL(34567.34567), CEIL(-34567.34567);
=====
34568.00000          -34567.00000

SELECT CEIL(34567.1), CEIL(-34567.1);
=====
34568.0          -34567.0
```


COS 함수

설명

COS 함수는 인자의 코사인(cosine) 값을 반환하며, 인자 x 는 라디안 값이어야 한다. 리턴 값은 **DOUBLE** 타입이다.

구문

```
COS ( x )
```

- x : 수치 값을 반환하는 임의의 연산식이다.

예제

```
SELECT COS(pi()/6), COS(pi()/3), COS(pi());  
=====
```

8.660254037844387e-01	5.000000000000001e-01	-1.000000000000000e+00
-----------------------	-----------------------	------------------------

COT 함수

설명

COT 함수는 인자 x 의 코탄젠트(cotangent) 값을 반환한다. 즉, 탄젠트가 x 인 값을 라디안 단위로 반환하며, 리턴 값은 **DOUBLE** 타입이다.

구문

```
COT ( x )
```

- x : 수치 값을 반환하는 임의의 연산식이다.

예제

```
SELECT COT(1), COT(-1), COT(0);  
=====
```

6.420926159343306e-01	-6.420926159343306e-01	NULL
-----------------------	------------------------	------

DEGREES 함수

설명

DEGREES 함수는 라디안 단위로 지정된 인자 x 를 각도로 환산하여 리턴한다. 리턴 값은 **DOUBLE** 타입이다.

구문

```
DEGREES ( x )
```

- x : 수치 값을 반환하는 임의의 연산식이다.

예제

```
SELECT DEGREES(pi()/6), DEGREES(pi()/3), DEGREES (pi());
```

```
=====
3.000000000000000e+01      5.999999999999999e+01      1.800000000000000e+02
=====
```

DRANDOM/DRAND 함수

설명

DRANDOM/DRAND 함수는 구간 0.0 이상 1.0 미만의 구간에서 임의의 이중 정밀도(double-precision) 부동 소수점 값을 반환하며, *seed* 인자를 지정할 수 있다. *seed* 인자의 타입은 **INTEGER**이며, 실수가 지정되면 반올림하고, **INTEGER** 범위를 초과하면 에러를 반환한다.

DRAND 함수는 연산을 출력하는 행(row)의 개수와 관계없이 한 문장 내에서 1회만 연산을 수행하여 오직 한 개의 임의값만 생성하는 반면, **DRANDOM** 함수는 함수가 호출될 때마다 매번 연산을 수행하므로 한 문장 내에서 여러 개의 다른 임의 값을 생성한다. 따라서, 무작위 순서로 행을 출력하기 위해서는 **ORDER BY** 절에 **DRANDOM** 함수를 이용해야 한다.

무작위 정수값을 구하기 위해서는 [RANDOM/RAND 함수](#)를 사용한다.

구문

```
DRANDOM( [seed] )
DRAND( [seed] )
```

예제

```
SELECT DRAND(), DRAND(1), DRAND(1.4);
=== <Result of SELECT Command in Line 1> ===

              drand()              drand(1)              drand(1.4)
=====
2.849646518006921e-001      4.163034446537495e-002      4.163034446537495e-002

SELECT * FROM rand tbl;
=== <Result of SELECT Command in Line 1> ===
      id  name
=====
      1  'a'
      2  'b'
      3  'c'
      4  'd'
      5  'e'
      6  'f'
      7  'g'
      8  'h'
      9  'i'
     10  'j'

--drandom() returns random values on every row
SELECT DRAND(), DRANDOM() FROM rand tbl;
=== <Result of SELECT Command in Line 1> ===

      drand()              drandom()
=====
7.638782921842098e-001      1.018707846308786e-001
7.638782921842098e-001      3.191320535905026e-001
7.638782921842098e-001      3.461714529862361e-001
7.638782921842098e-001      6.791894283883175e-001
7.638782921842098e-001      4.533829767754143e-001
7.638782921842098e-001      1.714224677266762e-001
7.638782921842098e-001      1.698049867244484e-001
7.638782921842098e-001      4.507583849604786e-002
7.638782921842098e-001      5.279091769157994e-001
```

```

7.638782921842098e-001    7.021088290047914e-001

--selecting rows in random order
SELECT * FROM rand tbl ORDER BY DRANDOM();

=== <Result of SELECT Command in Line 1> ===

      id  name
=====
        6  'f'
        2  'b'
        7  'g'
        8  'h'
        1  'a'
        4  'd'
       10  'j'
        9  'i'
        5  'e'
        3  'c'

```

FLOOR 함수

설명

FLOOR 함수는 인자보다 작거나 같은 최대 정수 값을 반환하며, 리턴 값의 타입은 인자의 타입과 같다.

구문

```
FLOOR( number_operand )
```

- number_operand*: 수치 값을 반환하는 임의의 연산식이다.

예제

```

--it returns the largest integer less than or equal to the arguments
SELECT FLOOR(34567.34567), FLOOR(-34567.34567);

=====
  34567.00000          -34568.00000

SELECT FLOOR(34567), FLOOR(-34567);
=====
    34567             -34567

```

FORMAT 함수

설명

FORMAT 함수는 숫자 *x*의 포맷이 '#,###,###.#####'이 되도록, 소수점 위 세 자리마다 콤마로 구분하고 소수점 아래 숫자가 *dec*만큼 표현되도록 *dec*의 아랫자리에서 반올림을 수행하여 결과를 반환한다. 리턴 값은 스트링 타입이다.

구문

```
FORMAT ( x , dec )
```

- x*, *dec*: 수치 값을 반환하는 임의의 연산식이다.

예제

```
SELECT FORMAT(12000.123456,3), FORMAT(12000.123456,0);
```

```
=====
'12,000.123'      '12,000'
```

GREATEST 함수

설명

GREATEST 함수는 인자로 지정된 하나 이상의 연산식을 서로 비교하여 가장 큰 값을 반환한다. 만약, 하나의 연산식만 지정되면 서로 비교할 대상이 없으므로 해당 연산식의 값을 그대로 반환한다.

따라서, 인자로 지정되는 하나 이상의 연산식은 서로 비교 가능한 타입이어야 한다. 지정된 인자의 타입이 동일하면 리턴 값의 타입도 동일하고, 인자의 타입이 다르면 리턴 값의 타입은 변환 가능(convertible)한 공통의 데이터 타입이 된다.

즉, **GREATEST** 함수는 같은 행(row) 내에서 컬럼 1, 컬럼 2, 컬럼 3의 값을 서로 비교하여 최대 값을 반환하며, **MAX** 함수는 모든 결과 행들의 컬럼 1 값을 서로 비교하여 최대 값을 반환한다.

구문

```
GREATEST ( expression [, expression]* )
```

- *expression* : 하나 이상의 연산식을 지정하며, 서로 비교 가능한 타입이어야 한다. 인자 중 어느 하나가 **NULL** 값이면 **NULL**을 반환한다.

예제

다음은 한국이 획득한 각 메달의 수와 최대 메달의 수를 반환하는 예제이다.(demodb)

```
SELECT gold, silver , bronze, GREATEST (gold, silver, bronze) FROM participant
WHERE nation code = 'KOR';
```

=== <Result of SELECT Command in Line 2> ===

gold	silver	bronze	greatest(gold, silver, bronze)
9	12	9	12
8	10	10	10
7	15	5	15
12	5	12	12
12	10	11	12

LEAST 함수

설명

LEAST 함수는 인자로 지정된 하나 이상의 연산식을 비교하여 가장 작은 값을 반환한다. 만약, 하나의 연산식만 지정되면 서로 비교할 대상이 없으므로 해당 연산식의 값을 그대로 반환한다.

따라서, 인자로 지정되는 하나 이상의 연산식은 서로 비교 가능한 타입이어야 한다. 만약, 지정된 인자의 타입이 동일하면 리턴 값의 타입도 동일하고, 인자의 타입이 다르면 리턴 값의 타입은 변환 가능(convertible)한 공통의 데이터 타입이 된다.

즉, **LEAST** 함수는 같은 행(row) 내에서 컬럼 1, 컬럼 2, 컬럼 3의 값을 서로 비교하여 최소 값을 반환하며, **MIN** 함수는 모든 결과 행들의 컬럼 1 값을 서로 비교하여 최소 값을 반환한다.

구문

```
LEAST( expression [, expression]* )
```

- *expression*: 하나 이상의 연산식을 지정하며, 서로 비교 가능한 타입이어야 한다. 인자 중 어느 하나가 **NULL** 값이면 **NULL**을 반환한다.

예제

다음은 한국이 획득한 각 메달의 수와 최소 메달의 수를 반환하는 예제이다. (demodb)

```
SELECT gold, silver , bronze, LEAST(gold, silver, bronze) FROM participant
WHERE nation_code = 'KOR';
```

=== <Result of SELECT Command in Line 2> ===

gold	silver	bronze	least(gold, silver, bronze)
9	12	9	9
8	10	10	8
7	15	5	5
12	5	12	5
12	10	11	10

LN 함수

설명

LN 함수는 진수 x 의 자연 로그(밑수가 e 인 로그) 값을 반환하며, 리턴 값은 **DOUBLE** 타입이다. 진수 x 가 0이거나 음수인 경우, 에러를 반환한다.

구문

```
LN ( x )
```

- x : 양수 값을 반환하는 임의의 연산식이다.

예제

```
SELECT ln(1), ln(2.72);
```

```
=====
0.0000000000000000e+00    1.000631880307906e+00
```

LOG2 함수

설명

LOG2 함수는 진수가 x 이고, 밑수가 2인 로그 값을 반환하며, 리턴 값은 **DOUBLE** 타입이다. 진수 x 가 0이거나 음수인 경우, 에러를 반환한다.

구문

LOG2 (*x*)

- x*: 양수 값을 반환하는 임의의 연산식이다.

예제

```
SELECT log2(1), log2(8);
=====
0.0000000000000000e+00    3.0000000000000000e+00
```

LOG10 함수

설명

LOG10 함수는 진수 *x*의 상용 로그 값을 반환하며, 리턴 값은 **DOUBLE** 타입이다. 진수 *x*가 0이거나 음수인 경우, 에러를 반환한다.

구문

LOG10 (*x*)

- x*: 양수 값을 반환하는 임의의 연산식이다.

예제

```
SELECT log10(1), log10(1000);
=====
0.0000000000000000e+00    3.0000000000000000e+00
```

MOD 함수

설명

MOD 함수는 첫 번째 인자 *m*을 두 번째 인자 *n*으로 나눈 나머지 값을 정수로 반환하며, 만약 *n*이 0이면, 나누기 연산을 수행하지 않고 *m* 값을 그대로 반환한다.

주의할 점은 피제수, 즉 **MOD** 함수의 인자 *m*이 음수인 경우, 전형적인 연산(classical modulus) 방식과 다르게 동작한다는 점이다. 아래의 표를 참고한다.

MOD 함수의 결과

m	n	MOD(m, n)	Classical Modulus m-n*FLOOR(m/n)
11	4	3	3
11	-4	3	-1
-11	4	-3	1
-11	-4	-3	-3

11	0	11	0으로 나누기 예러
----	---	----	------------

구문

MOD (*m*, *n*)

- *m*: 피제수를 나타내며, 수치 값을 반환하는 연산식이다.
- *n*: 제수를 나타내며, 수치 값을 반환하는 연산식이다.

예제

```
--it returns the remainder of m divided by n
SELECT MOD(11, 4), MOD(11, -4), MOD(-11, 4), MOD(-11, -4), MOD(11,0);

=====
3          3          -3          -3          11

SELECT MOD(11.0, 4), MOD(11.000, 4), MOD(11, 4.0), MOD(11, 4.000);

=====
3.0          3.000          3.0          3.000
```

PI 함수

설명

PI 함수는 π 값을 반환하며, 이중 정밀도(double-precision)로 표현된다.

구문

PI ()

예제

```
SELECT PI(), PI()/2;

=====
3.141592653589793e+00    1.570796326794897e+00
```

POW, POWER 함수

설명

POW 함수와 POWER 함수는 동일하며, 지정된 밑수 *x*를 지수 *y*만큼 거듭제곱한 값을 반환한다. 리턴 값은 DOUBLE 타입이다.

구문

POW (*x*, *y*)
POWER (*x*, *y*)

- *x*: 밑수를 나타내며, 수치 값을 반환하는 연산식이다.
- *y*: 지수를 나타내며, 수치 값을 반환하는 연산식이다. 밑수가 음수인 경우, 지수는 반드시 정수가 지정되어야 한다.

예제

```
SELECT POWER(2, 5), POWER(-2, 5), POWER(0, 0), POWER(1,0);
=====
3.2000000000000000e+01      -      1.0000000000000000e+00      1.0000000000000000e+00
3.2000000000000000e+01      1.0000000000000000e+00      1.0000000000000000e+00

--it returns an error when the negative base is powered by a non-int exponent
SELECT POWER(-2, -5.1), POWER(-2, -5.1);
ERROR
```

RADIANS 함수

설명

RADIANS 함수는 각도 단위로 지정된 인자 *x*를 라디안 단위로 환산하여 리턴한다. 리턴 값은 **DOUBLE** 타입이다.

구문

```
RADIANS ( x )
```

- *x*: 수치 값을 반환하는 임의의 연산식이다.

예제

```
SELECT RADIANS(90), RADIANS(180), RADIANS(360);
=====
1.570796326794897e+00      3.141592653589793e+00      6.283185307179586e+00
```

RANDOM/RAND 함수

설명

RANDOM/RAND 함수는 0 이상 2^{31} 미만 구간에서 임의의 정수 값을 반환하며, *seed* 인자를 지정할 수 있다. *seed* 인자의 타입은 **INTEGER**이며, 실수가 지정되면 반올림하고 **INTEGER** 범위를 초과하면 에러를 반환한다.

RAND 함수는 연산을 출력하는 행(row)의 개수와 관계없이 한 문장 내에서 1회만 연산을 수행하여 오직 한 개의 임의값만 생성하는 반면, **RANDOM** 함수는 함수가 호출될 때마다 매번 연산을 수행하므로 한 문장 내에서 여러 개의 다른 임의 값을 생성한다. 따라서, 무작위 순서로 행을 출력하기 위해서는 **RANDOM** 함수를 이용해야 한다.

무작위 실수 값을 구하기 위해서는 **DRANDOM/DRAND** [함수](#)를 사용한다.

구문

```
RANDOM( [seed] )
RAND ( [seed] )
```

예제

```
SELECT RAND(), RAND(1), RAND(1.4);
=== <Result of SELECT Command in Line 1> ===
```



```

      rand()      rand(1)      rand(1.4)
=====
1526981144      89400484      89400484

--creating a new table
SELECT * FROM rand_tbl;
=== <Result of SELECT Command in Line 1> ===

      id  name
=====
      1  'a'
      2  'b'
      3  'c'
      4  'd'
      5  'e'
      6  'f'
      7  'g'
      8  'h'
      9  'i'
     10  'j'

--random() returns random values on every row
SELECT RAND(),RANDOM() FROM rand_tbl;
=== <Result of SELECT Command in Line 1> ===

      rand()      random()
=====
2078876566      1753698891
2078876566      1508854032
2078876566      625052132
2078876566      279624236
2078876566      1449981446
2078876566      1360529082
2078876566      1563510619
2078876566      1598680194
2078876566      1160177096
2078876566      2075234419

--selecting rows in random order
SELECT * FROM rand_tbl ORDER BY RANDOM();
=== <Result of SELECT Command in Line 1> ===

      id  name
=====
      6  'f'
      1  'a'
      5  'e'
      4  'd'
      2  'b'
      7  'g'
     10  'j'
      9  'i'
      3  'c'
      8  'h'

```

ROUND 함수

설명

ROUND 함수는 지정된 인자 *number_operand*를 소수점 아래 *integer* 자리까지 반올림한 값을 반환한다.

반올림할 자릿수를 지정하는 *integer* 인자가 생략되거나 0인 경우에는 소수점 아래 첫째자리에서

반올림한다. 그리고 *integer* 인자가 음수이면, 소수점 위 자리, 즉 정수부에서 반올림한다.

구문

```
ROUND ( number_operand, integer )
```

- *number_operand*: 수치 값을 반환하는 임의의 연산식이다.
- *integer*: 반올림 처리할 위치를 지정한다. 양의 정수 *n*이 지정되면 소수점 아래 *n* 자리까지 표현되고, 음의 정수 *n*이 지정되면 소수점 위 *n* 자리에서 반올림한다.
- 리턴 값의 타입은 *number_operand*와 같은 타입이다.

예제

```
--it rounds a number to one decimal point when the second argument is omitted
SELECT ROUND(34567.34567), ROUND(-34567.34567);

=====
34567.00000          -34567.00000

--it rounds a number to three decimal point
SELECT ROUND(34567.34567, 3), ROUND(-34567.34567, 3) FROM db root;

=====
34567.34600          -34567.34600

--it rounds a number three digit to the left of the decimal point
SELECT ROUND(34567.34567, -3), ROUND(-34567.34567, -3);

=====
35000.00000          -35000.00000
```

SIGN 함수

설명

SIGN 함수는 지정된 인자 값의 부호를 반환한다. 양수이면 1을, 음수이면 -1을, 0이면 0을 반환한다.

구문

```
SIGN (number_operand)
```

- *number_operand*: 수치 값을 반환하는 임의의 연산식이다.

예제

```
--it returns the sign of the argument

SELECT SIGN(12.3), SIGN(-12.3), SIGN(0);

=====
1          -1          0
```

SIN 함수

설명

SIN 함수는 인자의 사인(sine) 값을 반환하며, 인자 *x*는 라디안 값이어야 한다. 리턴 값은 **DOUBLE** 타입이다.

구문

```
SIN ( x )
```

- x : 수치 값을 반환하는 임의의 연산식이다.

예제

```
SELECT SIN(pi()/6), SIN(pi()/3), SIN(pi());
=====
4.999999999999999e-01      8.660254037844386e-01      1.224646799147353e-16
```

SQRT 함수

설명

SQRT 함수는 x 의 제곱근(square root) 값을 **DOUBLE** 타입으로 반환한다.

구문

SQRT (x)

- x : 수치 값을 반환하는 연산식이다. 만약, 음수이면 에러를 반환한다.

예제

```
SELECT SQRT(4), SQRT(16.0);
=====
2.000000000000000e+00      4.000000000000000e+00
```

TAN 함수

설명

TAN 함수는 인자의 탄젠트(tangent) 값을 반환하며, 인자 x 는 라디안 값이어야 한다. 리턴 값은 **DOUBLE** 타입이다.

구문

TAN (x)

- x : 수치 값을 반환하는 임의의 연산식이다.

예제

```
SELECT TAN(pi()/6), TAN(pi()/3), TAN(pi()/4);
tan( pi()/6)      tan( pi()/3)      tan( pi()/4)
=====
5.773502691896257e-01      1.732050807568877e+00      9.999999999999999e-01
```

TRUNC, TRUNCATE 함수

설명

TRUNC 함수와 **TRUNCATE** 함수는 지정된 인자 x 의 소수점 아래 숫자가 dec 자리까지 표현되도록 버림(truncation)한 값을 반환한다. 단, **TRUNC** 함수의 dec 인자는 생략할 수 있지만, **TRUNCATE** 함수의

dec 인자는 생략할 수 없다. 버림할 위치를 지정하는 *dec* 인자가 음수이면 정수부의 소수점 위 *dec* 번째 자리까지 0으로 표시한다. 리턴 값의 표현 자릿수는 인자 *x*를 따른다.

구문

```
TRUNC ( x [, dec] )  
TRUNCATE ( x [, dec] )
```

- *x*: 수치 값을 반환하는 임의의 연산식이다.
- *dec*: 버림할 위치를 지정한다. 양의 정수 *n*이 지정되면 소수점 아래 *n* 자리까지 표현되고, 음의 정수 *n*이 지정되면 소수점 위 *n* 자리까지 0으로 표시한다. *dec* 인자가 0이거나 생략되면 소수부를 버림한다. 단, **TRUNCATE** 함수에서는 *dec* 인자를 생략할 수 없다.

예제

```
--it returns a number truncated to 0 places  
SELECT TRUNC(34567.34567), TRUNCATE(34567.34567, 0);  
  
=====
```

34567.00000	34567.00000
-------------	-------------

```
--it returns a number truncated to three decimal places  
SELECT TRUNC(34567.34567, 3), TRUNC(-34567.34567, 3);  
  
=====
```

34567.34500	-34567.34500
-------------	--------------

```
--it returns a number truncated to three digits left of the decimal point  
SELECT TRUNC(34567.34567, -3), TRUNC(-34567.34567, -3);  
  
=====
```

34000.00000	-34000.00000
-------------	--------------

TRUNC, TRUNCATE 함수

설명

TRUNC 함수와 **TRUNCATE** 함수는 지정된 인자 *x*의 소수점 아래 숫자가 *dec* 자리까지 표현되도록 버림(truncation)한 값을 반환한다. 단, **TRUNC** 함수의 *dec* 인자는 생략할 수 있지만, **TRUNCATE** 함수의 *dec* 인자는 생략할 수 없다. 버림할 위치를 지정하는 *dec* 인자가 음수이면 정수부의 소수점 위 *dec* 번째 자리까지 0으로 표시한다. 리턴 값의 표현 자릿수는 인자 *x*를 따른다.

구문

```
TRUNC ( x [, dec] )  
TRUNCATE ( x [, dec] )
```

- *x*: 수치 값을 반환하는 임의의 연산식이다.
- *dec*: 버림할 위치를 지정한다. 양의 정수 *n*이 지정되면 소수점 아래 *n* 자리까지 표현되고, 음의 정수 *n*이 지정되면 소수점 위 *n* 자리까지 0으로 표시한다. *dec* 인자가 0이거나 생략되면 소수부를 버림한다. 단, **TRUNCATE** 함수에서는 *dec* 인자를 생략할 수 없다.

예제

```
--it returns a number truncated to 0 places  
SELECT TRUNC(34567.34567), TRUNCATE(34567.34567, 0);
```

```
=====
34567.00000      34567.00000
--it returns a number truncated to three decimal places
SELECT TRUNC(34567.34567, 3), TRUNC(-34567.34567, 3);
=====
34567.34500      -34567.34500
--it returns a number truncated to three digits left of the decimal point
SELECT TRUNC(34567.34567, -3), TRUNC(-34567.34567, -3);
=====
34000.00000      -34000.00000
```

날짜/시간 함수와 연산자

EXTRACT 연산자

설명

EXTRACT 연산자는 날짜/시간 값을 반환하는 연산식 *date-time_argument* 중 일부분을 추출하여 **INTEGER** 타입으로 반환한다.

구문

```
EXTRACT ( field FROM date-time_argument )
```

field :

- **YEAR**
- **MONTH**
- **DAY**
- **HOURL**
- **MINUTE**
- **SECOND**
- **MILLISECOND**

date-time_argument :

- *expression*

- *field*: 날짜/시간 수식에서 추출할 값을 지정한다.
- *date-time_argument*: 날짜/시간 값을 반환하는 연산식이다. 이 연산식의 값은 **TIME**, **DATE**, **TIMESTAMP**, **DATETIME** 타입 중 하나여야 하며, **NULL**이 지정된 경우에는 **NULL** 값이 반환된다.

예제

```
SELECT EXTRACT(MONTH FROM DATETIME '2008-12-25 10:30:20:123' );
=====
12

SELECT EXTRACT(HOUR FROM DATETIME '2008-12-25 10:30:20:123' );
=====
10

SELECT EXTRACT(MILLISECOND FROM DATETIME '2008-12-25 10:30:20:123' );
=====
123
```

ADDDATE, DATE_ADD 함수

설명

ADDDATE 함수와 **DATE_ADD** 함수는 동일하며, 특정 **DATE** 값에 대해 덧셈 또는 뺄셈을 실행한다.

리턴 값은 **DATE** 타입 또는 **DATETIME** 타입이다. **DATETIME** 타입을 반환하는 경우는 다음과 같다.

- 첫 번째 인자가 **DATETIME** 타입 또는 **TIMESTAMP** 타입인 경우
- 첫 번째 인자가 **DATE** 타입이고 **INTERVAL** 값의 단위가 날짜 단위 미만으로 지정된 경우

따라서 위의 경우 외에 **DATETIME** 타입의 결과 값을 반환하려면 **CAST()** 함수를 이용하여 첫 번째 인자 값의 타입을 변환해야 한다. 연산 결과의 날짜가 해당 월의 마지막 날짜를 초과하면, 해당 월의 말일을 적용하여 유효한 **DATE** 값을 반환한다.

구문

```
ADDDATE(date, INTERVAL expr unit)
DATE_ADD(date, INTERVAL expr unit)
ADDDATE(date, days)
```

- *date*: **DATE**, **DATETIME** 또는 **TIMESTAMP** 타입의 연산식이며, 시작 날짜를 의미한다. 만약, '2006-07-00'와 같이 유효하지 않은 **DATE** 값이 지정되면, 에러를 반환한다.
- *expr*: 시작 날짜로부터 더할 시간 간격 값(interval value)을 의미하며, **INTERVAL** 키워드 뒤에 음수가 명시되면 시작 날짜로부터 시간 간격 값을 뺀다.
- *unit*: *expr* 수식에 명시된 시간 간격 값의 단위를 의미하며, 아래의 테이블을 참고하여 시간 간격 값 해석을 위한 포맷을 지정할 수 있다. *expr* 값이 *unit*에서 요구하는 값의 개수보다 적을 경우 가장 작은 단위부터 채운다. 예를 들어, **HOURL_SECONDS**의 경우 'HOURS:MINUTES:SECONDS'와 같이 3개의 값이 요구되는데, "1:1" 처럼 2개의 값만 주어진다면 'MINUTES:SECONDS'로 간주한다.

unit 값에 대한 expr 값

Unit 값	expr 값
MILLISECOND	MILLISECONDS
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS
YEAR	YEARS
SECOND_MILLISECOND	'SECONDS.MILLISECONDS'

MINUTE_MILLISECOND	'MINUTES:SECONDS.MILLISECONDS'
MINUTE_SECOND	'MINUTES:SECONDS'
HOUR_MILLISECOND	'HOURS:MINUTES:SECONDS.MILLISECONDS'
HOUR_SECOND	'HOURS:MINUTES:SECONDS'
HOUR_MINUTE	'HOURS:MINUTES'
DAY_MILLISECOND	'DAYS HOURS:MINUTES:SECONDS.MILLISECONDS'
DAY_SECOND	'DAYS HOURS:MINUTES:SECONDS'
DAY_MINUTE	'DAYS HOURS:MINUTES'
DAY_HOUR	'DAYS HOURS'
YEAR_MONTH	'YEARS-MONTHS'

예제

```
SELECT SYSDATE, ADDDATE(SYSDATE,INTERVAL 24 HOUR), ADDDATE(SYSDATE, 1);

=== <Result of SELECT Command in Line 1> ===

  SYS_DATE      date_add( SYS_DATE , INTERVAL 24 HOUR)      adddate( SYS_DATE , 1)
=====
03/30/2010 12:00:00.000 AM 03/31/2010                        03/31/2010

--it subtracts days when argument < 0
SELECT SYSDATE, ADDDATE(SYSDATE,INTERVAL -24 HOUR), ADDDATE(SYSDATE, -1);

=== <Result of SELECT Command in Line 1> ===

  SYS_DATE      date_add( SYS_DATE , INTERVAL -24 HOUR)      adddate( SYS_DATE , -1)
=====
03/30/2010 12:00:00.000 AM 03/29/2010                        03/29/2010

--when expr is not fully specified for unit
select sys datetime, adddate(sys datetime, interval '1:20' HOUR SECOND);

=== <Result of SELECT Command in Line 1> ===

  SYS DATETIME      date add( SYS DATETIME , INTERVAL '1:20' HOUR SECOND)
=====
06:18:24.149 PM 06/28/2010      06:19:44.149 PM 06/28/2010
```

ADD_MONTH 함수

설명

ADD_MONTHS 함수는 **DATE** 타입의 연산식 *date_argument*에 *month*를 더한 후, **DATE** 타입의 값을 반환한다. 인자로 지정된 값의 일(*dd*)이 연산 결과값의 월에 존재하면 해당 일(*dd*)을 반환하고, 존재하지 않으면 해당 월의 마지막 날(<*dd*)을 반환한다. 또한, 연산 결과값이 **DATE** 타입의 표현 범위를 초과하는 경우, 에러를 반환한다.

구문

```
ADD_MONTHS ( date_argument , month )
```

```

date_argument :
• date
• NULL

month :
• integer
• NULL

```

- *date_argument*: **DATE** 타입의 연산식을 지정한다. **TIMESTAMP**나 **DATETIME** 값을 지정하려면 **DATE** 타입으로 명시적 변환을 해야 한다. 값이 **NULL**이면 **NULL**을 반환한다.
- *month*: *date_argument*에 더할 개월 수를 지정하며, 양수와 음수 모두 지정될 수 있다. 만약, 정수가 아닌 타입의 값이 주어지면 묵시적으로 변환(소수점 아래 첫째자리를 반올림 처리)하여 정수형 타입으로 변환한다. 값이 **NULL**이면 **NULL**을 반환한다.

예제

```

--it returns DATE type value by adding month to the first argument

SELECT ADD_MONTHS (DATE '2008-12-25', 5), ADD_MONTHS (DATE '2008-12-25', -5);

=====
05/25/2009                                07/25/2008

SELECT ADD_MONTHS (DATE '2008-12-31', 5.5), ADD_MONTHS (DATE '2008-12-31', -5.5);

=====
06/30/2009                                06/30/2008

SELECT ADD_MONTHS (CAST (SYS DATETIME AS DATE), 5), ADD_MONTHS (CAST (SYS TIMESTAMP AS DATE),
5);

=====
07/03/2010                                07/03/2010

```

CURDATE, CURRENT_DATE, CURRENT_DATE(), SYS_DATE, SYSDATE

설명

CURDATE(), **CURRENT_DATE**, **CURRENT_DATE()**, **SYS_DATE**, **SYSDATE**는 모두 동일하며, 현재 날짜를 **DATE** 타입(*MM/DD/YYYY*)으로 반환한다. 산술 연산의 단위는 일(day)이다.

구문

```

CURDATE()
CURRENT_DATE()
CURRENT_DATE
SYS_DATE
SYSDATE

```

예제

```

--it returns the current date in DATE type
SELECT CURDATE(), CURRENT DATE(), CURRENT DATE, SYS DATE, SYSDATE;

=== <Result of SELECT Command in Line 1> ===

SYS DATE    SYS DATE    SYS DATE    SYS DATE    SYS DATE
=====
04/01/2010  04/01/2010  04/01/2010  04/01/2010  04/01/2010

--it returns the date 60 days added to the current date

```



```
SELECT CURDATE()+60;

=== <Result of SELECT Command in Line 1> ===

      SYS DATE +60
=====
      05/31/2010
```

CURRENT_DATETIME, CURRENT_DATETIME(), NOW(), SYS_DATETIME, SYSDATETIME

설명

CURRENT_DATETIME, CURRENT_DATETIME(), NOW() SYS_DATETIME, SYSDATETIME는 동일하며, 현재 날짜를 **DATETIME** 타입으로 반환한다. 산술 연산의 단위는 밀리초(milli-sec)다.

테이블 생성 시 컬럼 초기값 설정을 위해 **DEFAULT** 속성을 정의하고 **SYS_DATETIME**를 초기값으로 설정하면, 테이블 생성 시점의 타임스탬프 값이 디폴트 값으로 지정된다. 즉, 데이터 **INSERT** 시점의 타임스탬프 값이 디폴트로 입력되지 않으므로 주의한다. 타임스탬프 값을 입력하려면 데이터 입력 시 **INSERT** 구문의 **VALUES**에 **SYS_DATETIME** 값을 넣어야 한다.

구문

```
CURRENT_DATETIME
CURRENT_DATETIME ()
NOW ()
SYS_DATETIME
SYSDATETIME
```

예제

```
--it returns the current date and time in DATETIME type
SELECT NOW(), SYS_DATETIME;

=== <Result of SELECT Command in Line 1> ===

      SYS_DATETIME          SYS_DATETIME
=====
      04:08:09.829 PM 02/04/2010      04:08:09.829 PM 02/04/2010

--it returns the timestamp value 1 hour added to the current sys_datetime value
SELECT TO_CHAR(SYSDATETIME+3600*1000, 'YYYY-MM-DD HH:MI');

=====
      '2010-02-04 04:08'
```

CURTIME(), CURRENT_TIME, CURRENT_TIME(), SYS_TIME, SYSTIME

설명

CURTIME(), CURRENT_TIME, CURRENT_TIME(), SYS_TIME, SYSTIME는 모두 동일하며, 현재 시간을 **TIME** 타입(**HH:MI:SS**)으로 반환한다. 산술 연산의 단위는 초(sec)다.

구문

```
CURTIME ()
CURRENT_TIME
CURRENT_TIME ()
SYS_TIME
```

SYSTIME

예제

```
--it returns the current time in TIME type
SELECT CURTIME(), CURRENT_TIME(), CURRENT_TIME, SYS_TIME, SYSTIME;

=== <Result of SELECT Command in Line 1> ===

  SYS TIME      SYS TIME      SYS TIME      SYS TIME      SYS TIME
=====
  04:37:34 PM    04:37:34 PM    04:37:34 PM    04:37:34 PM    04:37:34 PM

--it returns the time value 1 hour added to the current sys time
SELECT CURTIME()+3600;

=== <Result of SELECT Command in Line 1> ===

  SYS TIME +3600
=====
  05:37:34 PM
```

CURRENT_TIMESTAMP, CURRENT_TIMESTAMP(), SYS_TIMESTAMP, SYSTIMESTAMP

설명

CURRENT_TIMESTAMP, **CURRENT_TIMESTAMP()**, **SYS_TIMESTAMP**, **SYSTIMESTAMP**는 동일하며, 현재 날짜와 시간을 **TIMESTAMP** 타입으로 반환한다. [LOCALTIME](#), [LOCALTIME\(\)](#), [LOCALTIMESTAMP](#), [LOCALTIMESTAMP\(\)](#)도 같은 기능을 제공한다. 산술 연산의 단위는 초(sec)다.

테이블 생성 시 컬럼 초기값 설정을 위해 **DEFAULT** 속성을 정의하고 **SYS_TIMESTAMP**를 초기값으로 설정하면, 테이블 생성 시점의 타임스탬프 값이 디폴트 값으로 지정된다. 즉, 데이터 **INSERT** 시점의 타임스탬프 값이 디폴트로 입력되지 않으므로 주의한다. 타임스탬프 값을 입력하려면 데이터 입력 시 **INSERT** 구문의 **VALUES**에 **SYS_TIMESTAMP** 값을 넣어야 한다.

구문

```
CURRENT_TIMESTAMP
CURRENT_TIMESTAMP()
SYS_TIMESTAMP
SYSTIMESTAMP
LOCALTIME
LOCALTIME()
LOCALTIMESTAMP
LOCALTIMESTAMP()
```

예제

```
--it returns the current date and time in TIMESTAMP type
SELECT LOCALTIME, SYS_TIMESTAMP;
=====
  07:00:48 PM 04/01/2010    07:00:48 PM 04/01/2010

--it returns the timestamp value 1 hour added to the current sys_timestamp value
SELECT CURRENT_TIMESTAMP()+3600;
=====
  08:02:42 PM 04/01/2010
```

DATE 함수

설명

DATE 함수는 지정된 인자로부터 날짜 부분을 추출하여 'MM/DD/YYYY' 포맷 스트링으로 반환한다. 지정 가능한 인자는 **DATE**, **TIMESTAMP**, **DATETIME** 타입이며, 리턴 값은 **VARCHAR** 타입이다.

구문

DATE (date)

- *date*: **DATE**, **TIMESTAMP**, **DATETIME** 타입이 지정될 수 있다.

예제

```
SELECT DATE('2010-02-27 15:10:23');
=====
'02/27/2010'

SELECT DATE(NOW());
=====
'04/01/2010'
```

DATEDIFF 함수

설명

DATEDIFF 함수는 주어진 두 개의 인자로부터 날짜 부분을 추출하여 두 값의 차이를 일 단위 정수로 반환한다. 지정 가능한 인자는 **DATE**, **TIMESTAMP**, **DATETIME** 타입이며, 리턴 값의 타입은 **INTEGER**이다.

구문

DATEDIFF (date1, date2)

- *date1, date2*: 날짜/시간 타입(**DATE**, **TIMESTAMP**, **DATETIME**) 또는 날짜/시간 포맷 스트링이 지정될 수 있다. 유효하지 않은 스트링이 지정되면 에러를 반환한다.

예제

```
SELECT DATEDIFF('2010-2-28 23:59:59','2010-03-02');
=====
-2

SELECT DATEDIFF('2010/12/31', SYSDATETIME);

In the command from line 1,
ERROR: Conversion error in date format.
```

DATE_SUB(), SUBDATE()

설명

DATE_SUB()와 **SUBDATE()**는 동일하며, 특정 **DATE** 값에 대해 뺄셈 또는 덧셈을 실행한다. 리턴 값은 **DATE** 타입 또는 **DATETIME** 타입이다. 연산 결과의 날짜가 해당 월의 마지막 날짜를 초과하면, 해당 월의 말일을 적용하여 유효한 **DATE** 값을 반환한다.

구문

```
DATE_SUB (date, INTERVAL expr unit)
SUBDATE(date, INTERVAL expr unit)
SUBDATE(date, days)
```

- *date*: **DATE**, **DATETIME** 또는 **TIMESTAMP** 타입의 연산식이며, 시작 날짜를 의미한다. 만약, '2006-07-00'와 같이 유효하지 않은 **DATE** 값이 지정되면, 에러를 반환한다.
- *expr*: 시작 날짜로부터 뺄 시간 간격 값(interval value)을 의미하며, **INTERVAL** 키워드 뒤에 음수가 명시되면 시작 날짜로부터 시간 간격 값을 더한다.
- *unit*: *expr* 수식에 명시된 시간 간격 값의 단위를 의미하며, *unit* 값에 대한 *expr* 인자의 값은 [ADDDATE](#), [DATE ADD 함수](#)의 표를 참고한다.

예제

```
SELECT SYSDATE, SUBDATE(SYSDATE,INTERVAL 24 HOUR), SUBDATE(SYSDATE, 1);

=== <Result of SELECT Command in Line 1> ===

  SYS_DATE      date_sub( SYS_DATE , INTERVAL 24 HOUR)      subdate( SYS_DATE , 1)
=====
03/30/2010  12:00:00.000 AM 03/29/2010                      03/29/2010

--it adds days when argument < 0
SELECT SYSDATE, SUBDATE(SYSDATE,INTERVAL -24 HOUR), SUBDATE(SYSDATE, -1);

=== <Result of SELECT Command in Line 1> ===

  SYS_DATE      date_sub( SYS_DATE , INTERVAL -24 HOUR)      subdate( SYS_DATE , -1)
=====
03/30/2010  12:00:00.000 AM 03/31/2010                      03/31/2010
```

LAST_DAY 함수

설명

LAST_DAY 함수는 인자로 지정된 **DATE** 값에서 해당 월의 마지막 날짜 값을 **DATE** 타입으로 반환한다.

구문

```
LAST_DAY ( date_argument )
```

date_argument :

- **date**
- **NULL**

- *date_argument*: **DATE** 타입의 연산식을 지정한다. **TIMESTAMP**나 **DATETIME** 값을 지정하려면 **DATE** 타입으로 명시적 변환을 해야 한다. 값이 **NULL**이면 **NULL**을 반환한다.

예제

```
--it returns last day of the momth in DATE type
SELECT LAST_DAY (DATE '1980-02-01'), LAST_DAY (DATE '2010-02-01');

=====
02/28/1980                02/28/2010

--it returns last day of the momth when explicitly casted to DATE type
SELECT LAST_DAY (CAST (SYS_TIMESTAMP AS DATE)), LAST_DAY (CAST (SYS_DATETIME AS DATE));

=====
02/28/2010                02/28/2010
```

LOCALTIME, LOCALTIME(), LOCALTIMESTAMP, LOCALTIMESTAMP()

설명

LOCALTIME, LOCALTIME(), LOCALTIMESTAMP, LOCALTIMESTAMP()는 동일하며, 현재 날짜와 시간을 **TIMESTAMP** 타입으로 반환한다. [CURRENT_TIMESTAMP, CURRENT_TIMESTAMP\(\), NOW\(\), SYS_TIMESTAMP, SYSTIMESTAMP](#)도 같은 기능을 제공한다. 산술 연산의 단위는 초(sec)다.

구문

```
LOCALTIME
LOCALTIME ()
LOCALTIMESTAMP
LOCALTIMESTAMP ()
CURRENT_TIMESTAMP
CURRENT_TIMESTAMP ()
SYS_TIMESTAMP
SYSTIMESTAMP
```

예제

```
--it returns the current date and time in TIMESTAMP type
SELECT LOCALTIME, SYS_TIMESTAMP;

=====
07:00:48 PM 04/01/2010    07:00:48 PM 04/01/2010

--it returns the timestamp value 1 hour added to the current sys_timestamp value
SELECT CURRENT_TIMESTAMP ()+3600;

=====
08:02:42 PM 04/01/2010
```

MONTHS_BETWEEN 함수

설명

MONTHS_BETWEEN 함수는 주어진 두 개의 **DATE** 값 간의 차이를 월 단위로 반환하며, 리턴 값은 **DOUBLE** 타입이다. 인자로 지정된 두 날짜가 동일하거나, 해당 월의 말일인 경우에는 정수 값을 반환하지만, 그 외의 경우에는 날짜 차이를 31로 나눈 값을 반환한다.

구문

```
MONTHS_BETWEEN (date_argument, date_argument)

date_argument :
• date
```

- **NULL**

- *date_argument*: **DATE** 타입의 연산식을 지정한다. **TIMESTAMP**나 **DATETIME** 값을 지정하려면 **DATE** 타입으로 명시적 변환을 해야 한다. 값이 **NULL**이면 **NULL**을 반환한다.

예제

```
--it returns the negative months when the first argument is the previous date
SELECT MONTHS BETWEEN (DATE '2008-12-31', DATE '2010-6-30');
=====
-1.8000000000000000e+001

--it returns integer values when each date is the last dat of the month
SELECT MONTHS BETWEEN (DATE '2010-6-30', DATE '2008-12-31');
=====
1.8000000000000000e+001

--it returns months between two arguments when explicitly casted to DATE type
SELECT MONTHS BETWEEN (CAST (SYS_TIMESTAMP AS DATE), DATE '2008-12-25');
=====
1.332258064516129e+001

--it returns months between two arguments when explicitly casted to DATE type
SELECT MONTHS BETWEEN (CAST (SYS_DATETIME AS DATE), DATE '2008-12-25');
=====
1.332258064516129e+001
```

DATE_SUB(), SUBDATE()

설명

DATE_SUB()와 **SUBDATE()**는 동일하며, 특정 **DATE** 값에 대해 뺄셈 또는 덧셈을 실행한다. 리턴 값은 **DATE** 타입 또는 **DATETIME** 타입이다. 연산 결과의 날짜가 해당 월의 마지막 날짜를 초과하면, 해당 월의 말일을 적용하여 유효한 **DATE** 값을 반환한다.

구문

```
DATE_SUB (date, INTERVAL expr unit)
SUBDATE (date, INTERVAL expr unit)
SUBDATE (date, days)
```

- *date*: **DATE**, **DATETIME** 또는 **TIMESTAMP** 타입의 연산식이며, 시작 날짜를 의미한다. 만약, '2006-07-00'와 같이 유효하지 않은 **DATE** 값이 지정되면, 에러를 반환한다.
- *expr*: 시작 날짜로부터 뺄 시간 간격 값(interval value)을 의미하며, **INTERVAL** 키워드 뒤에 음수가 명시되면 시작 날짜로부터 시간 간격 값을 더한다.
- *unit*: *expr* 수식에 명시된 시간 간격 값의 단위를 의미하며, *unit* 값에 대한 *expr* 인자의 값은 [ADDDATE, DATE ADD 함수](#)의 표를 참고한다.

예제

```
SELECT SYSDATE, SUBDATE(SYSDATE,INTERVAL 24 HOUR), SUBDATE(SYSDATE, 1);

=== <Result of SELECT Command in Line 1> ===

SYS DATE      date sub( SYS DATE , INTERVAL 24 HOUR)      subdate( SYS DATE , 1)
=====
03/30/2010    12:00:00.000 AM 03/29/2010                    03/29/2010

--it adds days when argument < 0
SELECT SYSDATE, SUBDATE(SYSDATE,INTERVAL -24 HOUR), SUBDATE(SYSDATE, -1);
```

```

=== <Result of SELECT Command in Line 1> ===
      SYS DATE      date sub( SYS DATE , INTERVAL -24 HOUR)      subdate( SYS DATE , -1)
=====
03/30/2010  12:00:00.000 AM 03/31/2010                        03/31/2010

```

TIMESTAMP 함수

설명

TIMESTAMP 함수는 인자로 날짜/시간 포맷의 스트링이 지정되고, 이를 **DATETIME** 타입으로 반환한다.

단일 인자로 **DATE** 포맷 스트링('YYYY-MM-DD' 또는 'MM/DD/YYYY' 또는 **TIMESTAMP** 포맷 스트링('YYYY-MM-DD HH.MI.SS' 또는 'HH.MI.SS MM/DD/YYYY')이 지정되면 이를 **DATETIME** 타입으로 반환한다.

두 번째 인자로 **TIME** 포맷 스트링('HH.MI.SS')이 주어지면 이를 첫 번째 인자 값에 더한 결과를 **DATETIME** 타입으로 반환한다. 두 번째 인자가 명시되지 않으면, 디폴트로 **12:00:00.000 AM**이 설정된다.

구문

```
TIMESTAMP(date [,time])
```

- *date* : 'YYYY-MM-DD', 'MM/DD/YYYY', 'YYYY-MM-DD HH.MI.SS', 'HH.MI.SS MM/DD/YYYY' 포맷 스트링이 지정될 수 있다.
- *time* : 'HH.MI.SS' 포맷 스트링이 지정될 수 있다.

예제

```

SELECT TIMESTAMP('2009-12-31'), TIMESTAMP('2009-12-31','12:00:00');
=====
12:00:00.000 AM 12/31/2009      12:00:00.000 PM 12/31/2009

SELECT TIMESTAMP('2010-12-31 12:00:00','12:00:00');
=====
12:00:00.000 AM 01/01/2011

SELECT TIMESTAMP('13:10:30 12/25/2008');
=====
01:10:30.000 PM 12/25/2008

```

UNIX_TIMESTAMP 함수

설명

UNIX_TIMESTAMP 함수는 인자를 생략할 수 있으며, 인자를 생략하면 '1970-01-01 00:00:00' UTC 이후 현재 시스템 날짜/시간까지의 초 단위 시간 간격(interval)을 **INTEGER** 타입의 리턴 값을 반환한다.

date 인자가 지정되면 '1970-01-01 00:00:00' UTC 이후 지정된 날짜/시간까지의 초 단위 시간 간격을 반환한다.

구문

```
UNIX_TIMESTAMP( [date] )
```

- *date*: **DATE** 타입, **TIMESTAMP** 타입, **DATE** 포맷 스트링('YYYY-MM-DD' 또는 'MM/DD/YYYY'), **TIMESTAMP** 포맷 스트링('YYYY-MM-DD HH:MI:SS', 'HH:MI:SS MM/DD/YYYY') 또는 'YYYYMMDD' 포맷 스트링이 지정될 수 있다.

예제

```
SELECT UNIX_TIMESTAMP('1970-01-02'), UNIX_TIMESTAMP();

      unix_timestamp('1970-01-02')      unix_timestamp()
=====
                54000                1270196737
```

데이터 타입 변환 함수와 연산자

CAST 연산자

설명

CAST 연산자를 **SELECT** 문에서 어떤 값의 데이터 타입을 다른 데이터 타입으로 명시적으로 변환하는 데 사용할 수 있다. 조회 리스트 또는 **WHERE** 절의 값 수식을 다른 데이터 타입으로 변환할 수 있다.

CUBRID에서 **CAST** 연산자를 사용한 명시적인 타입 변환에 대해서 정리하면 다음의 표와 같다.

	EN	AN	VC	FC	VB	FB	D	T	UT	DT	S	MS	SQ
EN	예	예	예	예	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요
AN	예	예	예	예	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요
VC	예	예	예[1]	예*	예	예	예	예	예	예	아니요	아니요	아니요
FC	예	예	예*	예*	예	예	예	예	예	예	아니요	아니요	아니요
VB	아니요	아니요	예	예	예	예	아니요	아니요	아니요	아니요	아니요	아니요	아니요
FB	아니요	아니요	예	예	예	예	아니요	아니요	아니요	아니요	아니요	아니요	아니요
D	아니요	아니요	예	예	아니요	아니요	예	아니요	예	예	아니요	아니요	아니요
T	아니요	아니요	예	예	아니요	아니요	아니요	예	아니요	아니요	아니요	아니요	아니요
UT	아니요	아니요	예	예	아니요	아니요	예	예	예	예	아니요	아니요	아니요
DT	아니요	아니요	예	예	아니요	아니요	예	예	예	예	아니요	아니요	아니요
S	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요	예	예	예
MS	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요	예	예	예
SQ	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요	아니요	예	예	예

[1] 이 경우에 **CAST** 연산은 값 수식과 변환할 데이터 타입이 같은 문자 세트를 가질 경우에만 허용된다.

데이터 타입 키

- **EN** : 정확한 수치(**INTEGER, SMALLINT, BIGINT, NUMERIC, DECIMAL**)
- **AN** : 근사값 수치(**FLOAT/REAL, DOUBLE PRECISION, MONETARY**)
- **VC** : 가변 길의 문자열(**VARCHAR(*n*), NCHAR VARYING(*n*)**)
- **FC** : 고정 길의 문자열(**CHAR(*n*), NCHAR(*n*)**)
- **VB** : 가변 길의 비트열(**BIT VARYING(*n*)**)
- **FB** : 고정 길의 비트열(**BIT(*n*)**)
- **D** : 날짜(**DATE**)
- **T** : 시간(**TIME**)
- **DT** : 날짜 시간(**DATETIME**)
- **UT** : 타임스탬프(**TIMESTAMP**)
- **S** : 집합(**SET**)
- **MS** : 다중 집합(**MULTISET**)
- **SQ** : 순차집합(**LIST, SEQUENCE**)

구문

CAST (*cast operand AS cast target*)

cast operand :

- *value expression*
- **NULL**

cast target :

- *data type*

- *cast_operand* : 다른 타입으로 변환할 값을 선언한다.
- *cast_target* : 변환할 타입을 지정한다.

예제

```
--operation after casting character as INT type returns 2
SELECT (1+CAST ('1' AS INT));
=====
2

--cannot cast the string which is out of range as SMALLINT
SELECT (1+CAST ('1234567890' AS SMALLINT));

ERROR

--operation after casting returns 1+1234567890
SELECT (1+CAST ('1234567890' AS INT));
=====
1234567891

--'1234.567890' is casted to 1235 after rounding up
SELECT (1+CAST ('1234.567890' AS INT));
=====
1236

--'1234.567890' is casted to string containing only first 5 letters.
SELECT (CAST ('1234.567890' AS CHAR(5)));
=====
'1234.'
```

```
--numeric type can be casted to CHAR type only when enough length is specified
SELECT (CAST(1234.567890 AS CHAR(5)));

ERROR

--numeric type can be casted to CHAR type only when enough length is specified
SELECT (CAST(1234.567890 AS CHAR(11)));
=====
'1234.567890'

--numeric type can be casted to CHAR type only when enough length is specified
SELECT (CAST(1234.567890 AS VARCHAR));
=====
'1234.567890'

--string can be casted to time/date types only when its literal is correctly specified
SELECT (CAST('2008-12-25 10:30:20' AS TIMESTAMP));
=====
10:30:20 AM 12/25/2008

SELECT (CAST('10:30:20' AS TIME));
=====
10:30:20 AM

--string can be casted to TIME type when its literal is same as TIME's.
SELECT (CAST('2008-12-25 10:30:20' AS TIME));

ERROR

--string can be casted to TIME type after specifying its type of the string
SELECT (CAST(TIMESTAMP'2008-12-25 10:30:20' AS TIME));
=====
10:30:20 AM
```

주의 사항

- **CAST** 변환은 같은 문자 세트를 가지는 데이터 타입끼리만 허용된다.
- 근사치 데이터 타입이 정수형으로 변환되는 경우, 소수점 아래 자리가 반올림 처리된다.
- 수치 데이터 타입을 문자열 타입으로 변환하는 경우, 문자열의 길이가 (모든 유효 숫자 자리 + 소수점) 이상이 되도록 충분히 지정해야 한다. 그렇지 않으면 에러가 발생된다.
- 문자열 타입 A를 문자열 타입 B로 변환하는 경우, A 길이 이상이 되도록 충분히 지정되지 않으면 문자열 끝 부분이 삭제(truncate)되어 저장된다.
- 문자열 타입 A를 날짜/시간 데이터 타입 B로 변환하는 경우, A의 리터럴이 B 타입과 일치하는 경우에만 변환된다. 그렇지 않을 경우 에러가 발생된다.
- 문자열로 저장된 수치 데이터는 명시적으로 타입 변환을 해주어야 산술 연산이 가능하다.

DATE_FORMAT 함수

설명

DATE_FORMAT 함수는 **DATE** 포맷('YYYY-MM-DD' 또는 'MM/DD/YYYY')를 포함하는 스트링 또는 날짜/시간 타입(**DATE**, **TIMESTAMP**, **DATETIME**) 값을 지정된 날짜/시간 포맷으로 변환하여 스트링으로 출력하며, 리턴 값은 **VARCHAR** 타입이다.

구문

```
DATE_FORMAT(date, format)
```

- *date* : **DATE** 포맷('YYYY-MM-DD' 또는 'MM/DD/YYYY')를 포함하는 스트링 또는 날짜/시간 타입(**DATE**, **TIMESTAMP**, **DATETIME**) 값이 지정될 수 있다.
- *format* : 출력 포맷을 지정한다. '%'로 시작하는 포맷 지정자(specifier)를 사용한다. 아래의 표를 참고한다. 아래의 [날짜/시간 포맷 2](#)는 **DATE_FORMAT** 함수, [TIME_FORMAT](#) 함수, [STR_TO_DATE](#) 함수에서 사용된다.

디폴트 날짜/시간 출력 포맷

날짜/시간 타입	디폴트 출력 포맷
DATE	'MM/DD/YYYY'
TIME	'HH:MI:SS AM'
TIMESTAMP	'HH:MI:SS AM MM/DD/YYYY'
DATETIME	'HH:MI:SS.FF AM MM/DD/YYYY'

날짜/시간 포맷 2

format 값 의미	
%a	Weekday, 영문 약어 (Sun, ..., Sat)
%b	Month, 영문 약어 (Jan, ..., Dec)
%c	Month(1, ..., 12)
%D	Day of the month, 서수 영문 스트링(1st, 2nd, 3rd, ...)
%d	Day of the month, 두 자리 숫자(01, ..., 31)
%e	Day of the month (1, ..., 31)
%f	Milliseconds, 세 자리 숫자 (000, ..., 999)
%H	Hour, 24 시간 기준, 두 자리 수 이상 (00, ..., 23, ..., 100, ...)
%h	Hour, 12 시간 기준 두 자리 숫자 (01, ..., 12)
%I	Hour, 12 시간 기준 두 자리 숫자 (01, ..., 12)
%i	Minutes , 두 자리 숫자(00, ..., 59)
%j	Day of year, 세 자리 숫자 (001, ..., 366)
%k	Hour, 24 시간 기준, 한 자리 수 이상 (0, ..., 23, ..., 100, ...)
%l	Hour , 12 시간 기준 (1, ..., 12)
%M	Month, 영문 스트링 (January, ..., December)
%m	Month, 두 자리 숫자 (01, ..., 12)
%p	AM or PM
%r	Time, 12 시간 기준, 시:분:초 (hh:mm:ss AM or hh:mm:ss PM)

%S	Seconds, 두 자리 숫자 (00, ..., 59)
%s	Seconds , 두 자리 숫자(00, ..., 59)
%T	Time, 24 시간 기준, 시:분:초 (hh:mm:ss)
%U	Week, 두 자리 숫자, 일요일이 첫날인 주 단위(00, ..., 53)
%u	Week, 두 자리 숫자, 월요일이 첫날인 주 단위(00, ..., 53)
%V	Week, 두 자리 숫자, 일요일이 첫날인 주 단위(01, ..., 53) %X 와 결합되어 사용 가능
%v	Week, 두 자리 숫자, 월요일이 첫날인 주 단위(01, ..., 53) %x 와 결합되어 사용 가능
%W	Weekday, 영문 스트링 (Sunday, ..., Saturday)
%w	Day of the week, 숫자 인덱스 (0=Sunday, ..., 6=Saturday)
%X	Year, 네 자리 숫자, 일요일이 첫날인 주 단위로 계산(0000, ..., 9999) %V 와 결합되어 사용 가능
%x	Year, 네 자리 숫자, 월요일이 첫날인 주 단위로 계산(0000, ..., 9999) %v 와 결합되어 사용 가능
%Y	Year, 네 자리 숫자(0001, ..., 9999)
%y	Year, 두 자리 숫자(00, 01, ..., 99)
%%	특수문자 "%"를 그대로 출력하는 경우
%x	포맷 지정자로 쓰이지 않는 영문자 중 임의의 문자 x를 그대로 출력하는 경우

예제

```
SELECT DATE_FORMAT('2009-10-04 22:23:00', '%W %M %Y');
=====
'Sunday October 2009'

SELECT DATE_FORMAT('2007-10-04 22:23:00', '%H:%i:%s');
=====
'22:23:00'

SELECT DATE_FORMAT('1900-10-04 22:23:00', '%D %y %a %d %m %b %j');
=====
'4th 00 Thu 04 10 Oct 277'

SELECT DATE_FORMAT('1999-01-01', '%X %V');
=====
'1998 52'
```

STR_TO_DATE 함수

설명

STR_TO_DATE 함수는 인자로 주어진 스트링을 지정된 포맷에 따라 해석한 후 이를 날짜/시간 값으로 변환하며, [DATE_FORMAT 함수](#)와 반대로 동작한다. 리턴 값은 스트링에 포함된 날짜 또는 시간 부분에 따라 타입이 결정되며, **DATETIME**, **DATE**, **TIME** 타입 중 하나이다. 만약, *string*에 유효하지 않은 날짜나 시간 값이 포함되거나, *format*에 지정된 포맷 지정자를 적용하여 스트링을 해석할 수 없는 경우, 에러를 리턴한다.

구문

```
STR_TO_DATE(string, format)
```

- *string*: 모든 스트링 타입이 지정될 수 있다.
- *format*: 스트링 해석을 위한 포맷을 지정한다. '%'를 포함하는 스트링을 포맷 지정자(specifier)로 사용하며, 표 [날짜/시간 포맷 2](#)를 참고한다.

예제

```
SELECT STR_TO_DATE('01,5,2013','%d,%m,%Y');
=====
05/01/2013

SELECT STR_TO_DATE('May 1, 2013','%M %d,%Y');
=====
05/01/2013

SELECT STR_TO_DATE('a09:30:17','a%h:%i:%s');
=====
09:30:17 AM

SELECT STR_TO_DATE('09:30:17a','%h:%i:%s');
=====
09:30:17 AM
```

TIME_FORMAT 함수

설명

TIME_FORMAT 함수는 **TIME** 포맷(*HH:MM:SS*)을 포함하는 스트링 또는 **TIME**을 포함하는 날짜/시간 타입(**TIME**, **TIMESTAMP**, **DATETIME**) 값을 지정된 시간 포맷으로 변환하여 스트링으로 출력하며, 리턴 값은 **VARCHAR** 타입이다.

구문

```
TIME_FORMAT(time, format)
```

- *time*: **TIME** 포맷(*HH:MM:SS*)을 포함하는 스트링, **TIME**을 포함하는 날짜/시간 타입(**TIME**, **TIMESTAMP**, **DATETIME**) 값을 지정할 수 있다.
- *format*: 출력 포맷을 지정한다. '%'를 포함하는 스트링을 지정자(specifier)로 사용한다. 표 [날짜/시간 포맷 2](#)를 참고한다. 시간과 관련 없는 포맷 지정자가 사용되면, 영문자가 그대로 출력된다.

예제

```
SELECT TIME FORMAT('22:23:00', '%H %i %s');
=====
'22 23 00'

SELECT TIME FORMAT('25:59:00', '%H %h %i %s %f');
=====
'25 01 59 00 000'

SELECT SYSTIME, TIME FORMAT(SYSTIME, '%T');
=====
08:46:53 PM '20:46:53'
```

TO_CHAR 함수(date_time)

설명

TO_CHAR 함수는 **TIME** 포맷(*HH:MI:SS*)을 포함하는 스트링 또는 **TIME**을 포함하는 날짜/시간 타입(**TIME**, **TIMESTAMP**, **DATETIME**) 값을 [날짜/시간 포맷 1](#)에 따라 스트링으로 변환하여 이를 반환하며, 리턴 값의 타입은 **VARCHAR**이다. 포맷 인자가 생략되면, 디폴트 포맷에 따라 문자열로 변환한다. 주어진 스트링과 대응하지 않는 포맷 지정자가 지정되면 에러를 반환한다.

구문

```
TO_CHAR( date_time [, format[, date_lang_string_literal ]] )

date_time :
• date
• time
• timestamp
• datetime
• NULL

format :
• character strings (날짜/시간 포맷 1 표 참조)
• NULL

date_lang_string_literal : (date lang string literal 표 참조)

• 'en_US'
• 'ko_KR'
```

- *date_time*: 날짜/시간 타입의 연산식을 지정한다. 값이 **NULL**인 경우에는 **NULL**이 반환된다.
- *format*: 리턴 값의 포맷을 지정한다. 아래의 [날짜/시간 포맷 1](#)을 참고한다. 포맷이 지정되지 않으면, 디폴트 포맷이 적용된다. 값이 **NULL**인 경우에는 **NULL**이 반환된다.
- *date_lang_string_literal*: 리턴 값에 적용할 언어를 지정한다([date_lang_string_literal](#) 표 참조). 디폴트 값은 'en_US'이다. **CUBRID_DATE_LANG** 환경 변수를 지정하여 수정할 수 있다.

디폴트 날짜/시간 출력 포맷

날짜/시간 타입	디폴트 출력 포맷
DATE	'MM/DD/YYYY'
TIME	'HH:MI:SS AM'
TIMESTAMP	'HH:MI:SS AM MM/DD/YYYY'

DATETIME		'HH:MI:SS.FF AM MM/DD/YYYY'
날짜/시간 포맷 1		
format 값	의미	
CC	세기	
YYYY, YY	4 자리 연도, 2 자리 연도	
Q	분기(1, 2, 3, 4; 1 월~3 월 = 1)	
MM	월(01-12; 1 월 = 01) 참고: 분(minute)은 MI 이다.	
MONTH	월 이름	
MON	축약된 월 이름	
DD	날(1-31)	
DAY	요일 이름	
DY	축약된 요일 이름	
D 또는 d	요일(1-7)	
AM 또는 PM	오전/오후	
A.M. 또는 P.M.	마침표가 포함된 오전/오후	
HH 또는 HH12	시(1-12)	
HH24	시(0-23)	
MI	분(0-59)	
SS	초(0-59)	
FF	밀리초(0-999)	
- / , . ; : "텍스트" 구두점과 인용구는 그대로 결과에 표현됨		
date_lang_string_literal 예		
포맷 구성 요소	Date_lang_string_literal	
	'en_US'	'ko_KR'
MONTH	JANUARY	1 월
MON	JAN	1
DAY	MONDAY	월요일
DY	MON	월
Month	January	1 월

Mon	Jan	1
Day	Monday	월요일
Dy	Mon	월
month	january	1 월
mon	jan	1
day	monday	월요일
Dy	mon	월
AM	AM	오전
Am	Am	오전
am	am	오전
A.M.	A.M.	오전
A.m.	A.m.	오전
a.m.	a.m.	오전
PM	AM	오전
Pm	Am	오전
pm	am	오전
P.M.	A.M.	오전
P.m.	A.m.	오전
p.m.	a.m	오전

리턴 값 포맷의 자릿수

포맷 구성 요소	자릿수
MONTH(Month, month)	9 (ko_KR 의 경우 4)
MON(Mon, mon)	3 (ko_KR 의 경우 2)
DAY(Day, day)	9 (ko_KR 의 경우 6)
DY(Dy, dy)	3 (ko_KR 의 경우 2)
HH12, HH24	2
"텍스트"	텍스트의 길이
나머지 포맷	포맷의 길이와 같음

예제

```
--creating a table having date/time type columns
```



```

CREATE TABLE datetime_tbl(a TIME, b DATE, c TIMESTAMP, d DATETIME);
INSERT INTO datetime_tbl VALUES(SYSTIME, SYSDATE, SYSTIMESTAMP, SYSDATETIME);

--selecting a VARCHAR type string from the data in the specified format
SELECT TO_CHAR(b, 'DD, DY , MON, YYYY') FROM datetime_tbl;
=====
'04, THU , FEB, 2010'

SELECT TO_CHAR(c, 'HH24:MI, DD, MONTH, YYYY') FROM datetime_tbl;
=====
'16:50, 04, FEBRUARY , 2010'

SELECT TO_CHAR(c, 'HH24:MI:FF, DD, MONTH, YYYY') FROM datetime_tbl;
ERROR: Invalid format.

SELECT TO_CHAR(d, 'HH12:MI:SS:FF pm, YYYY-MM-DD-DAY') FROM datetime_tbl;
=====
'04:50:11:624 pm, 2010-02-04-THURSDAY '

SELECT TO_CHAR(TIMESTAMP'2009-10-04 22:23:00', 'Day Month yyyy');
=====
'Sunday October 2009'

```

TO_CHAR 함수(number)

설명

TO_CHAR 함수는 날짜/시간 데이터 타입 또는 수치형 데이터 타입을 [숫자 포맷](#)에 맞는 문자열로 변환하여 이를 반환하며, 리턴 값의 타입은 **VARCHAR**이다. 만약, 포맷 인자가 지정되지 않으면, 디폴트 포맷에 따라 모든 유효 숫자를 문자열로 변환한다.

구문

```
TO_CHAR(number argument[, format argument ])
```

number argument :

- numeric(decimal)*
- integer*
- smallint*
- bigint*
- float(real)*
- double*
- NULL**

format_argument :

- character strings ([숫자 포맷](#) 표 참조)
- NULL**

- number_argument*: 숫자를 반환하는 수치형 데이터 타입의 연산식을 지정한다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- format_argument*: 리턴 값의 포맷을 지정한다. 포맷이 지정되지 않으면, 디폴트로 모든 유효 숫자를 문자열로 변환한다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.

숫자 포맷

포맷 구성 요소	예제	설명
9	9999	"9"의 개수는 반환될 유효숫자 자릿수를 나타낸다. 숫자 인자에 대해 포맷에서 지정된 유효숫자 자릿수가 부족하면, 소수부에 대해서는 반올림 연산을 수행한다. 숫자 인자의 정

		수부 자릿수보다 유효숫자 자릿수가 부족하면 #을 출력한다.
0	0999	포맷에서 지정된 유효숫자 자릿수가 충분한 경우, 정수부 앞 부분을 공백이 아닌 0 으로 채워 반환한다.
S	S9999	지정된 위치에 양수/음수 부호를 출력한다. 부호는 문자열의 시작부분에만 사용할 수 있다.
C	C9999	지정된 위치에 ISO 통화 기호를 반환한다.
, (쉼표)	9,999	지정된 위치에 쉼표(",")를 반환한다. 포맷 내에는 여러 개의 쉼표가 허용된다.
. (마침표)	9.999	지정된 위치에 숫자의 정수부와 소수부를 구분하는 소수점 (".")을 출력한다. 포맷 내에는 하나의 소수점만 허용된다.
EEEE	9.99EEEE	과학적 기수법(scientific notation)을 반환한다.

예제

```
--selecting a string casted from a number in the specified format
SELECT TO_CHAR(12345,'S9999999'), TO_CHAR(12345,'S0999999');

=====
' +12345'          '+012345'

SELECT TO_CHAR(1234567,'C9,999,999,999');
=====
' $1,234,567'

SELECT TO_CHAR(123.4567,'99'), TO_CHAR(123.4567,'999.99999'),
TO_CHAR(123.4567,'99999.999');
=====
'##'              '123.45670'          ' 123.457'

SELECT TO_CHAR(1.234567,'99.999EEEE'), TO_CHAR(1.234567E-4);
=====
'1.235E+00'       '0.0001234567'
```

TO_DATE 함수

설명

TO_DATE 함수는 인자로 지정된 날짜 포맷을 기준으로 문자열을 해석하여, 이를 **DATE** 타입의 값으로 변환하여 반환한다. 날짜 포맷은 [TO_CHAR 함수\(date_time\)](#)의 설명 부분을 참고하고, 만약 날짜 포맷이 인자로 지정되지 않으면, 디폴트 포맷인 'MM/DD/YYYY'을 적용하여 해석한다.

구문

```
TO_DATE(string argument[,format argument[,date lang string literal]])

string argument :
• character strings
• NULL

format_argument :
```

- *character strings* ([날짜/시간 포맷 1](#) 표 참조)
- **NULL**

date_lang_string_literal : ([date lang string literal](#) 표 참조)

- 'en_US'
- 'ko_KR'

- *string_argument*: 문자열을 반환하는 임의의 연산식이다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *format_argument*: 날짜 타입으로 변환할 값의 포맷을 지정하며, [TO_CHAR 함수\(date_time\)](#)의 날짜/시간 포맷 표를 참고한다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *date_lang_string_literal*: 입력 값에 적용할 언어를 지정한다. **CUBRID_DATE_LANG** 환경 변수를 지정하여 수정할 수 있다.

예제

```
--selecting a date type value casted from a string in the specified format
SELECT TO_DATE('12/25/2008');
=====
12/25/2008

SELECT TO_DATE('25/12/2008', 'DD/MM/YYYY');
=====
12/25/2008

SELECT TO_DATE('081225', 'YMMDD');
=====
12/25/2008

SELECT TO_DATE('2008-12-25', 'YYYY-MM-DD');
=====
12/25/2008
```

TO_DATETIME 함수

설명

TO_DATETIME 함수는 인자로 지정된 **DATETIME** 포맷을 기준으로 문자열을 해석하여, 이를 DATETIME타입의 값으로 변환하여 반환한다. **DATETIME** 포맷은 [TO_CHAR 함수\(date_time\)](#)의 설명 부분을 참고하고, 만약 포맷이 인자로 지정되지 않으면 디폴트 포맷인 'HH:MI:SS.FF [am|pm] MM/DD/YYYY'를 적용하여 문자열을 해석한다.

구문

```
TO_DATETIME(string_argument[,format_argument[,date_lang_string_literal]])

string_argument :
• character strings
• NULL

format_argument :
• character strings (날짜/시간 포맷 1 표 참조)
• NULL

date_lang_string_literal : (date lang string literal 표 참조)
• 'en_US'
• 'ko_KR'
```

- *string_argument*: 문자열을 반환하는 임의의 연산식이다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.

- *format_argument*: **DATETIME** 타입으로 변환할 값의 포맷을 지정하며, [TO_CHAR 함수\(date_time\)](#)의 날짜/시간 포맷 표를 참고한다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *date_lang_string_literal*: 입력 값에 적용할 언어를 지정한다. **CUBRID_DATE_LANG** 환경 변수를 지정하여 수정할 수 있다.

예제

```
--selecting a datetime type value casted from a string in the specified format
SELECT TO_DATETIME('13:10:30 12/25/2008');
=====
01:10:30.000 PM 12/25/2008

SELECT TO_DATETIME('08-Dec-25 13:10:30.999', 'YY-Mon-DD HH24:MI:SS.FF');
=====
01:10:30.999 PM 12/25/2008

SELECT TO_DATETIME('DATE: 12-25-2008 TIME: 13:10:30.999', '"DATE:" MM-DD-YYYY "TIME:"
HH24:MI:SS.FF');
=====
01:10:30.999 PM 12/25/2008
```

TO_NUMBER 함수

설명

TO_NUMBER 함수는 인자로 지정된 숫자 포맷을 기준으로 문자열을 해석하여, 이를 **NUMERIC** 타입으로 변환하여 반환한다. 숫자 포맷이 지정되지 않으면, 디폴트로 문자열에 포함된 모든 유효 숫자를 **NUMERIC** 타입의 숫자로 반환한다.

구문

```
TO_NUMBER(string argument[, format argument ])

string argument :
• character strings
• NULL

format_argument :
• character strings
• NULL
```

- *string_argument*: 문자열을 반환하는 임의의 연산식이다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *format_argument*: 숫자로 반환할 값의 포맷을 지정하며, [TO_CHAR 함수\(number\)](#)의 숫자 포맷 표를 참고한다. 값이 **NULL**이면 에러를 반환한다.

예제

```
--selecting a number casted from a string in the specified format
SELECT TO_NUMBER('-1234');
=====
-1234

SELECT TO_NUMBER('12345', '9999999');
=====
12345
```

```
SELECT TO_NUMBER('$12,345.67','C99,999.999');
=====
12345.670

SELECT TO_NUMBER('12345.67','99999.999');
=====
12345.670
```

TO_TIME 함수

설명

TO_TIME 함수는 인자로 지정된 시간 포맷을 기준으로 문자열을 해석하여, 이를 TIME 타입의 값으로 변환하여 반환한다. 시간 포맷은 [TO_CHAR 함수\(date_time\)](#)의 설명 부분을 참고하고, 만약 시간 포맷이 인자로 지정되지 않으면, 디폴트 포맷인 'HH:MI:SS'를 적용하여 해석한다.

구문

```
TO_TIME(string_argument[,format_argument [,date_lang_string_literal]]):
```

string_argument :

- character strings
- **NULL**

format_argument :

- character strings ([날짜/시간 포맷 1](#) 표 참조)
- **NULL**

date_lang_string_literal : ([date_lang_string_literal](#) 표 참조)

- 'en_US'
- 'ko_KR'

- *string_argument*: 문자열을 반환하는 임의의 연산식이다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *format_argument*: 시간 타입으로 변환할 값의 포맷을 지정하며, [TO_CHAR 함수\(date_time\)](#)의 날짜/시간 포맷 표를 참고한다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *date_lang_string_literal*: 입력 값에 적용할 언어를 지정한다. **CUBRID_DATE_LANG** 환경 변수를 지정하여 수정할 수 있다.

예제

```
--selecting a time type value casted from a string in the specified format

SELECT TO TIME ('13:10:30');
=====
01:10:30 PM

SELECT TO TIME('HOUR: 13 MINUTE: 10 SECOND: 30', '"HOUR:" HH24 "MINUTE:" MI "SECOND:" SS');
=====
01:10:30 PM

SELECT TO TIME ('13:10:30', 'HH24:MI:SS');
=====
01:10:30 PM

SELECT TO TIME ('13:10:30', 'HH12:MI:SS');
ERROR //a string and the format not matched
```

TO_TIMESTAMP 함수

설명

TO_TIMESTAMP 함수는 인자로 지정된 타임스탬프 포맷을 기준으로 문자열을 해석하여, 이를 **TIMESTAMP** 타입의 값으로 변환하여 반환한다. 타임스탬프 포맷은 [TO_CHAR 함수\(date_time\)](#)의 설명 부분을 참고하고, 만약 타임스탬프 포맷이 인자로 지정되지 않으면, 디폴트 포맷인 'HH:MI[:SS] [am|pm] MM/DD/YYYY'를 적용하여 문자열을 해석한다.

구문

```
TO_TIMESTAMP(string argument[, format argument[, date_lang_string_literal]])
```

string argument :
• character strings
• **NULL**

format argument :
• character strings ([날짜/시간 포맷 1](#) 표 참조)
• **NULL**

date_lang_string_literal : ([date_lang_string_literal](#) 표 참조)
• 'en_US'
• 'ko_KR'

- *string_argument*: 문자열을 반환하는 임의의 연산식이다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *format_argument*: 타임스탬프 타입으로 변환할 값의 포맷을 지정하며, [TO_CHAR 함수\(date_time\)](#)의 날짜/시간 포맷 표를 참고한다. 값이 **NULL**이면 결과로 **NULL**이 반환된다.
- *date_lang_string_literal*: 입력 값에 적용할 언어를 지정한다. **CUBRID_DATE_LANG** 환경 변수를 지정하여 수정할 수 있다.

예제

```
--selecting a timestamp type value casted from a string in the specified format

SELECT TO_TIMESTAMP('13:10:30 12/25/2008');
=====
01:10:30 PM 12/25/2008

SELECT TO_TIMESTAMP('08-Dec-25 13:10:30', 'YY-Mon-DD HH24:MI:SS');
=====
01:10:30 PM 12/25/2008

SELECT TO_TIMESTAMP('YEAR: 2008 DATE: 12-25 TIME: 13:10:30', '"YEAR:" YYYY "DATE:" MM-DD "TIME:" HH24:MI:SS');
=====
01:10:30 PM 12/25/2008
```

집계 함수

AVG 함수

설명

AVG 함수는 모든 행에 대한 연산식 값의 산술 평균을 구한다. 하나의 연산식 *expression*만 인자로 지정되며, 연산식 앞에 **DISTINCT** 또는 **UNIQUE** 키워드를 포함시키면 연산식 값 중 중복을 제거한 후 평균을 구하고, 키워드가 생략되거나 **ALL**인 경우에는 모든 값에 대해서 평균을 구한다.

구문

```
AVG ( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

- *expression*: 수치 값을 반환하는 임의의 연산식을 지정한다. 집합 데이터를 반환하는 연산식은 지정될 수 없다.
- **ALL**: 모든 값에 대해 평균을 구하기 위해 사용되며, 디폴트 값이다.
- **DISTINCT** 또는 **UNIQUE**: 중복이 제거된 유일한 값에 대해서만 평균을 구하기 위해 사용된다.

예제

다음은 한국이 획득한 금메달의 평균 수를 반환하는 예제이다. (demodb)

```
SELECT AVG(gold)
FROM participant
WHERE nation_code = 'KOR';
```

결과값: 9

COUNT 함수

설명

COUNT 함수는 질의문이 반환하는 결과 행들의 개수를 반환한다. 별표(*)를 지정하면 조건을 만족하는 모든 행(**NULL** 값을 가지는 행 포함)의 개수를 반환하며, **DISTINCT** 또는 **UNIQUE** 키워드를 연산식 앞에 지정하면 중복을 제거한 후 유일한 값을 가지는 행(**NULL**값을 가지는 행은 포함하지 않음)의 개수만 반환한다. 따라서, 반환되는 값은 항상 정수이며, **NULL**은 반환되지 않는다.

연산식 *expression*은 수치형 또는 스트링 타입은 물론, 컬렉션 타입 컬럼과 오브젝트 도메인(사용자 정의 클래스 또는 멀티미디어 클래스)을 가지는 컬럼도 지정될 수 있다.

구문

```
COUNT ( * | [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

- *expression*: 임의의 연산식이다.
- **ALL**: 주어진 *expression*의 모든 행의 개수를 구하기 위해 사용되며, 디폴트 값이다.
- **DISTINCT** 또는 **UNIQUE**: 중복이 제거된 유일한 값을 가지는 행의 개수를 구하기 위해 사용된다.

예제

다음은 역대 올림픽 중에서 마스코트가 존재했었던 올림픽의 수를 반환하는 예제이다. (demodb)

```
SELECT COUNT(*)
FROM olympic
WHERE mascot IS NOT NULL;
```

결과값: 9

MAX 함수

설명

MAX 함수는 모든 행에 대하여 연산식 값 중 최대 값을 구한다. 하나의 연산식 *expression*만 인자로 지정된다. 문자열을 반환하는 연산식에 대해서는 사전 순서를 기준으로 뒤에 나오는 문자열이 최대 값이 되고, 수치를 반환하는 연산식에 대해서는 크기가 가장 큰 값이 최대 값이다.

구문

```
MAX ( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

- *expression*: 수치 또는 문자열을 반환하는 하나의 연산식을 지정한다. 집합 데이터를 반환하는 연산식은 지정할 수 없다.
- **ALL**: 모든 값에 대해 최대 값을 구하기 위해 사용되며, 디폴트 값이다.
- **DISTINCT** 또는 **UNIQUE**: 중복이 제거된 유일한 값에 대해서 최대 값을 구하기 위해 사용된다.

예제

다음은 올림픽 대회 중 한국이 획득한 최대 금메달의 수를 반환하는 예제이다.

```
SELECT MAX(gold) FROM participant WHERE nation code = 'KOR';

=== <Result of SELECT Command in Line 1> ===

      max(gold)
=====
             12
```

MIN 함수

설명

MIN 함수는 모든 행에 대하여 연산식 값 중 최소 값을 구한다. 하나의 연산식 *expression*만 인자로 지정된다. 문자열을 반환하는 연산식에 대해서는 사전 순서를 기준으로 앞에 나오는 문자열이 최소 값이 되고, 수치를 반환하는 연산식에 대해서는 크기가 가장 작은 값이 최소 값이다.

구문

```
MIN ( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

- *expression*: 수치 또는 문자열을 반환하는 하나의 연산식을 지정한다. 집합 데이터를 반환하는 연산식은 지정할 수 없다.

- **ALL** : 모든 값에 대해 최소 값을 구하기 위해 사용되며, 디폴트 값이다.
- **DISTINCT** 또는 **UNIQUE** : 중복이 제거된 유일한 값에 대해서 최소 값을 구하기 위해 사용된다.

예제

다음은 올림픽 대회 중 한국이 획득한 최소 금메달의 수를 반환하는 예제이다.(demodb)

```
SELECT MIN(gold) FROM participant WHERE nation_code = 'KOR';

=== <Result of SELECT Command in Line 1> ===

      min(gold)
=====
              7
```

STDDEV 함수

설명

STDDEV 함수는 모든 행에 대한 연산식 값들에 대해 표준 편차를 반환한다. 하나의 연산식 *expression*만 인자로 지정되며, 연산식 앞에 **DISTINCT** 또는 **UNIQUE** 키워드를 포함시키면 연산식 값 중 중복을 제거한 후, 표준 편차를 구하고, 키워드가 생략되거나 **ALL**인 경우에는 모든 값에 대해 표준 편차를 구한다.

이때, 리턴 값은 인자로 지정된 연산식의 타입을 따르므로, 실제 계산 값과 다를 수 있다.

구문

```
STDDEV( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL] expression )
```

- *expression* : 수치를 반환하는 하나의 연산식을 지정한다.
- **ALL** : 모든 값에 대해 표준 편차를 구하기 위해 사용되며, 디폴트 값이다.
- **DISTINCT** 또는 **UNIQUE** : 중복이 제거된 유일한 값에 대해서만 표준 편차를 구하기 위해 사용된다.

예제

다음은 한국이 획득한 금메달 수의 표준편차를 반환하는 예제이다.

```
SELECT host_year, gold FROM participant WHERE nation_code = 'KOR';

=== <Result of SELECT Command in Line 1> ===

      host_year      gold
=====
      2004           9
      2000           8
      1996           7
      1992          12
      1988          12

SELECT STDDEV(gold), STDDEV(CAST (gold AS FLOAT)) FROM participant
WHERE nation code = 'KOR';

=== <Result of SELECT Command in Line 1> ===

      stddev(gold)  stddev( cast(gold as float))
=====
              2              2.302172e+000
```

SUM 함수

설명

SUM 함수는 모든 행에 대한 연산식 값들의 합계를 반환한다. 하나의 연산식 *expression*만 인자로 지정되며, 연산식 앞에 **DISTINCT** 또는 **UNIQUE** 키워드를 포함시키면 연산식 값 중 중복을 제거한 후 합계를 구하고, 키워드가 생략되거나 **ALL**인 경우에는 모든 값에 대해 합계를 구한다.

구문

```
SUM ( [ { DISTINCT | DISTINCTROW } | UNIQUE | ALL ] expression )
```

단일 값 수식을 **SUM** 함수의 입력으로 주어질 수 있다.

- *expression* : 수치를 반환하는 하나의 연산식을 지정한다.
- **ALL** : 모든 값에 대해 합계를 구하기 위해 사용되며, 디폴트로 지정된다.
- **DISTINCT** 또는 **UNIQUE** : 중복이 제거된 유일한 값에 대해서만 합계를 구하기 위해 사용된다.

예제

다음은 역대 올림픽에서 획득한 금메달 수의 합계를 기준으로 10위권 국가와 금메달 총 수를 출력하는 예제이다. (demodb)

```
SELECT nation_code, SUM(gold) FROM participant GROUP BY nation_code  
ORDER BY SUM(gold) DESC  
FOR ORDERBY_NUM() BETWEEN 1 AND 10 ;
```

=== <Result of SELECT Command in Line 1> ===

nation code	sum(gold)
'USA'	190
'CHN'	97
'RUS'	85
'GER'	79
'URS'	55
'FRA'	53
'AUS'	52
'ITA'	48
'KOR'	48
'EUN'	45

10 rows selected.

VARIANCE 함수

설명

VARIANCE 함수는 모든 행에 대한 연산식 값들에 대해 분산을 반환한다. 하나의 연산식 *expression*만 인자로 지정되며, 연산식 앞에 **DISTINCT** 또는 **UNIQUE** 키워드를 포함시키면 연산식 값 중 중복을 제거한 후 분산을 구하고, 키워드가 생략되거나 **ALL**인 경우에는 모든 값에 대해 분산을 구한다.

이때, 리턴 값은 인자로 지정된 연산식의 타입을 따르므로, 실제 계산 값과 다를 수 있다.

다음은 함수에 적용된 공식이다.

$$\frac{\left[\text{SUM}(x^2) - \frac{(\text{SUM}(x))^2}{n} \right]}{(n-1)}$$

구문

```
VARIANCE( [DISTINCT | UNIQUE | ALL] expression )
```

- *expression*: 수치 값을 반환하는 연산식을 지정한다.
- **ALL**: 모든 값에 대해 분산을 구하기 위해 사용되며, 디폴트로 지정된다.
- **DISTINCT** 또는 **UNIQUE**: 중복이 제거된 유일한 값에 대해서만 분산을 구하기 위해 사용된다.

예제

다음은 한국이 획득한 금메달 수의 분산을 반환하는 예제이다.

```
SELECT VARIANCE(gold), VARIANCE(CAST (gold AS FLOAT)) FROM participant
WHERE nation_code = 'KOR';
=== <Result Of SELECT Command in Line 1> ===

    variance(gold)    variance( cast(gold as float))
=====
                5                5.299995e+000
```

클릭 카운터 함수

INCR 과 DECR 함수

설명

INCR 함수는 **SELECT** 절에 포함되어 인자로 주어진 컬럼의 값을 1 증가시켜 주는 기능을 한다. **DECR** 함수는 해당 컬럼의 값을 1 감소시킨다.

구문

```
SELECT [ qualifier ] select_expression
[ { TO | INTO } variable [ {, variable }...; ] ]
...;
select_expression :
*
table_name. *
[expression | counter expression] [ {, expression |
counter_expression}...]

counter_expression :
INCR(path_expression)
```

INCR 함수와 **DECR** 함수는 '클릭 카운터' 함수로 불리며, 게시판 유형의 웹 서비스에서 게시물의 조회수를 증가시키는데 유용하게 사용될 수 있다. 게시물의 내용을 **SELECT**하고 곧바로 게시물의 조회수를 **UPDATE**로 1 증가하는 유형의 시나리오에서 하나의 **SELECT** 문에 **INCR** 함수를 사용함으로써 한번에 게시물 내용 조회와 조회수 증가 작업을 수행할 수 있다.

INCR 함수는 인자로 명시된 컬럼 값을 증가시킨다. 단, 인자로는 정수 타입의 숫자형만 올 수 있고, 값이 **NULL**인 경우 **INCR** 함수를 수행하여도 값은 **NULL**을 유지한다. 즉, 값이 설정되어야 **INCR** 함수를 써서 값을 증가시킬 수 있다. **DECR** 함수는 인자로 명시된 컬럼 값을 감소시킨다.

SELECT 절에 **INCR** 함수를 명시한 경우, **COUNTER** 값을 1 증가하고 질의 결과는 증가하기 전의 값으로 출력한다. 그리고, **INCR** 함수는 질의 처리 과정에서 참여한 행(tuple)이 아니라 최종 결과에 참여한 행에 대해서만 값을 증가시킨다.

참고 사항

- **INCR/DECR** 함수는 사용자 정의 트랜잭션과 별도로 시스템 내부에서 사용되는 top operation이 적용되어 트랜잭션의 **COMMIT/ROLLBACK**과 상관없이 데이터베이스에 자동으로 적용된다..
- 하나의 **SELECT** 문에 **INCR/DECR** 함수를 여러 개 사용할 경우, 해당 질의 내의 각각의 **INCR/DECR** 함수 중 하나라도 실패하면 전부 실패된다.
- **INCR/DECR** 함수는 최상위 **SELECT** 문에만 적용된다. **INSERT ...; SELECT ...** 구문과 **UPDATE table SET col = SELECT ...** 등과 같은 **SUB SELECT**문은 지원하지 않는다. 다음은 **INCR** 함수가 허용되지 않는 예이다.

```
SELECT b.content, INCR(b.read_count) FROM (SELECT * FROM board WHERE id = 1) AS b
```

- **INCR/DECR** 함수가 포함된 **SELECT** 문의 경우, 결과 행 의 개수가 둘 이상이면 오류로 처리한다. 최종 결과가 1 건만 있을 경우만 유효하다.
- **INCR/DECR** 함수는 숫자 형식의 도메인에 대해서만 사용할 수 있다. 적용 가능한 도메인은 **SMALLINT**, **INTEGER**와 같은 정수형 데이터 타입으로 제한된다. 기타 도메인에는 사용할 수 없다.
- **INCR** 함수 호출 시 결과 값은 현재 값이며, 저장 값은 현재 값 +1인 값이 저장된다. 결과를 저장값과 같은 값을 조회하고자 할 경우는 다음과 같이 수행한다.

```
SELECT content, INCR(read_count) + 1 FROM board WHERE id = 1;
```

- 정의된 도메인의 최대값을 초과 할 경우 **INCR** 함수는 해당 컬럼을 0으로 초기화 한다. 반대로 최소값에 **DECR** 함수가 적용되어도 0으로 초기화된다.
- **INCR/DECR** 함수는 **UPDATE** 트리거와 무관하게 실행되므로 데이터 일관성이 보장되지 않을 수 있다. 다음은 **INCR** 함수가 **UPDATE** 트리거와 무관하게 실행되기 때문에 데이터베이스의 일관성이 위반되는 예이다.

```
CREATE TRIGGER event_tr BEFORE UPDATE ON event EXECUTE REJECT;
SELECT INCR(players) FROM event WHERE gender='M';
```

예제

먼저, board 테이블에는 아래와 같이 3건의 데이터가 입력되었다고 가정한다.

```
CREATE TABLE board (
id INT, title VARCHAR(100), content VARCHAR(4000), read count INT );
INSERT INTO board VALUES (1, 'aaa', 'text...', 0);
INSERT INTO board VALUES (2, 'bbb', 'text...', 0);
INSERT INTO board VALUES (3, 'ccc', 'text...', 0);
```

다음은 id 값이 1인 데이터의 read_count 컬럼의 값을 **INCR** 함수로 증가시키는 예이다.

```
SELECT content, INCR(read_count) FROM board WHERE id = 1;
```

```
=== <Result of SELECT Command in Line 1> ===
```

```
content          read count
=====
'text...'         0
```

예와 같이 **SELECT** 문에 **INCR** 함수를 사용함으로써 해당 컬럼 값은 read_count + 1이 된다. 결과는 다음과 같은 **SELECT** 문을 통해 확인해 볼 수 있다.

```
SELECT content, read count FROM board WHERE id = 1;
```

```
=== <Result of SELECT Command in Line 1> ===
```

```
content          read count
=====
'text...'         1
```

ROWNUM 함수

ROWNUM/INST_NUM() 함수

설명

ROWNUM 함수는 질의 결과로 생성될 각 레코드에 대한 순서를 나타내는 번호를 반환한다. 첫 번째 결과 레코드는 1, 두 번째 결과 레코드는 2를 가진다.

일반적인 **SELECT** 문에서는 **ROWNUM**, **INST_NUM()**을, **GROUP BY** 절을 포함한 **SELECT** 문에서는 **GROUPBY_NUM()**을 사용할 수 있다. **ROWNUM** 함수를 사용하면 질의의 결과 레코드 수를 다양한 방법으로 제한할 수 있다. 예를 들어, 질의 결과의 처음 10건만 조회한다거나, 짝수 번째 또는 홀수 번째 레코드만 반환하도록 할 수 있다.

ROWNUM 함수는 결과 타입이 정수형이고, **SELECT** 절과 **WHERE** 절과 같이 질의 내에 수식이 위치할 수 있는 모든 곳에 사용할 수 있다. 하지만, **ROWNUM** 함수 결과를 속성 또는 연관된 부질의(correlated subquery)와 비교하는 것은 허용되지 않는다.

구문

```
INST_NUM()  
ROWNUM
```

참고 사항

- **WHERE** 절에 명시된 **ROWNUM** 함수는 **INST_NUM()** 함수와 같은 의미를 가진다. **INST_NUM()** 함수는 스칼라(scalar) 함수이지만, **GROUPBY_NUM()** 함수는 집계 함수의 일종이다. **GROUP BY** 절을 가지는 **SELECT** 문에서는 **INST_NUM()** 대신에 **GROUPBY_NUM()**을 사용해야 한다.
- **ROWNUM** 함수는 각각의 **SELECT** 문장에 종속된다. 즉, **ROWNUM** 함수가 부질의에 쓰인 경우, 부질의를 수행하는 동안에 부질의 결과에 대하여 일련 번호를 반환한다. 내부적으로, **ROWNUM** 함수 결과는 조회된 레코드를 질의 결과 집합에 쓰기 직전에 생성된다. 이 순간에 질의 결과 집합의 레코드에 대한 일련 번호를 생성하는 카운터 값이 증가된다.
- **SELECT** 문에 **ORDER BY** 절이 포함된 경우 **WHERE** 절에 명시된 **ROWNUM** 함수의 값은 **ORDER BY** 절 처리를 위한 정렬 과정 전에 생성된다. **SELECT** 문에 **GROUP BY** 절이 포함된 경우에는 **HAVING** 절에

명시된 **GROUPBY_NUM()** 함수의 값은 질의 결과가 그룹화된 이후에 계산된다. **ORDER BY** 절에 의한 정렬 과정이 완료된 이후에 결과 레코드의 일련 번호를 얻어내기 위해서는 **ORDER BY** 절에 **ORDERBY_NUM()** 함수를 사용해야 한다.

- **ROWNUM** 함수는 **SELECT**문 뿐만 아니라 **INSERT**, **DELETE**, **UPDATE**와 같은 SQL 문에도 쓸 수 있다. 예를 들어, **INSERT INTO table_name SELECT ... FROM ... WHERE ...** 질의와 같이 한 테이블의 행(row) 중 일부를 조회하여 다른 테이블에 삽입하고자 할 때, **WHERE** 절에 **ROWNUM** 함수를 사용할 수 있다.

예제

다음은 1988 올림픽에서 금메달 개수를 기준으로 4위권 국가 이름을 반환하는 예제이다. (demodb)

```
--Limiting 4 rows using ROWNUM in the WHERE condition
SELECT * FROM
(SELECT nation_code FROM participant WHERE host_year = 1988
ORDER BY gold DESC) AS T
WHERE ROWNUM <5;

=== <Result of SELECT Command in Line 4> ===

    nation code
=====
    'URS'
    'GDR'
    'USA'
    'KOR'

4 rows selected.

--Limiting 4 rows using FOR ORDERBY_NUM()
SELECT ROWNUM, nation_code FROM participant WHERE host year = 1988
ORDER BY gold DESC
FOR ORDERBY NUM() < 5;

=== <Result of SELECT Command in Line 3> ===

    rownum    nation code
=====
        156    'URS'
        155    'GDR'
        154    'USA'
        153    'KOR'

4 rows selected.

--Unexpected results : ROWNUM operated before ORDER BY
SELECT ROWNUM, nation_code FROM participant
WHERE host_year = 1988 AND ROWNUM < 5
ORDER BY gold DESC;

=== <Result of SELECT Command in Line 3> ===

    rownum    nation code
=====
         1    'ZIM'
         2    'ZAM'
         3    'ZAI'
         4    'YMD'

4 rows selected.
```

GROUPBY_NUM() 함수

설명

GROUPBY_NUM() 함수는 **ROWNUM** 혹은 **INST_NUM()** 함수와 함께, 결과 행들의 개수를 제한하는 목적으로 사용된다. 단, 차이점은 **GROUP BY ... HAVING** 절 뒤에 결합되어 사용되며, 이미 정렬을 수행한 결과에 대해 순서를 부여한다는 점이다. 또한, **INST_NUM()** 함수는 스칼라(scalar) 함수이지만, **GROUPBY_NUM()** 함수는 집계 함수의 일종이다.

즉, **GROUP BY** 절이 포함된 **SELECT** 문장에서 조건 절에 **ROWNUM**을 이용하여 일부 결과 행들만 조회하는 경우, **ROWNUM**이 먼저 적용된 후 **GROUP BY**에 의한 그룹 정렬이 수행된다. 반면, **GROUPBY_NUM()** 함수를 이용하여 일부 결과 행들만 조회하는 경우, **GROUP BY**에 의한 그룹 정렬이 이루어진 결과에 대해서 **ROWNUM**이 적용된다.

구문

```
GROUPBY_NUM()
```

예제

다음은 history 테이블에서 과거 5개의 올림픽에 대해서 최단 기록을 조회하는 예제이다. (demodb)

```
--Group-ordering first and then limiting rows using GROUPBY NUM()
SELECT host year, MIN(score) FROM history
GROUP BY host year HAVING GROUPBY NUM() BETWEEN 1 AND 5;
```

```
=== <Result of SELECT Command in Line 2> ===
```

host year	min(score)
1968	'8.9'
1980	'01:53.0'
1984	'13:06.0'
1988	'01:58.0'
1992	'02:07.0'

```
5 rows selected.
```

```
--Limiting rows first and then Group-ordering using ROWNUM
SELECT host year, MIN(score) FROM history
WHERE ROWNUM BETWEEN 1 AND 5 GROUP BY host year;
```

```
=== <Result of SELECT Command in Line 2> ===
```

host year	min(score)
2000	'03:41.0'
2004	'01:45.0'

```
2 rows selected.
```

ORDERBY_NUM() 함수

설명

ORDERBY_NUM() 함수는 **ROWNUM** 혹은 **INST_NUM()** 함수와 함께, 결과 행들의 개수를 제한하는 목적으로 사용된다. 단, 차이점은 **ORDER BY** 절 뒤에 결합되어 사용되고, 이미 정렬을 수행한 결과에 대해 순서를 부여한다는 점이다.

즉, **ORDER BY** 절이 포함된 **SELECT** 문장에서 조건절에 **ROWNUM**을 이용하여 일부 결과 행들만 조회하는 경우, **ROWNUM**이 먼저 적용된 후 **ORDER BY**에 의한 정렬이 수행된다. 반면, **ORDERBY_NUM()** 함수를 이용하여 일부 결과 행들만 조회하는 경우, **ORDER BY**에 의한 정렬이 이루어진 결과에 대해서 **ROWNUM**이 적용된다.

구문

```
FOR ORDERBY_NUM()
```

예제

다음은 history 테이블에서 3위에서 5위까지의 선수 이름과 기록을 조회하는 예제이다. .(demodb)

```
--Ordering first and then limiting rows using FOR ORDERBY_NUM()
SELECT athlete, score FROM history
ORDER BY score FOR ORDERBY_NUM() BETWEEN 3 AND 5;
```

=== <Result of SELECT Command in Line 1> ===

athlete	score
'Luo Xuejuan'	'01:07.0'
'Rodal Vebjorn'	'01:43.0'
'Thorpe Ian'	'01:45.0'

3 rows selected.

```
--Limiting rows first and then Ordering using ROWNUM
SELECT athlete, score FROM history
WHERE ROWNUM BETWEEN 3 AND 5 ORDER BY score;
```

=== <Result of SELECT Command in Line 1> ===

athlete	score
'Thorpe Ian'	'01:45.0'
'Thorpe Ian'	'03:41.0'
'Hackett Grant'	'14:43.0'

3 rows selected.

정보 함수

CURRENT_USER, USER, USER 함수, SYSTEM_USER 함수

설명

CURRENT_USER 또는 **USER**는 동일하며, 현재 데이터베이스에 로그인한 사용자의 이름을 문자열로 반환한다.

USER() 또는 **SYSTEM_USER()**는 동일하며, 사용자 이름을 호스트 이름과 함께 반환한다.

구문

```
CURRENT_USER
USER
USER()
SYSTEM_USER()
```

예제

```
--selecting the current user on the session
SELECT USER;

=== <Result of SELECT Command in Line 1> ===

      CURRENT_USER
=====
      'PUBLIC'

SELECT USER(), CURRENT_USER;

=== <Result of SELECT Command in Line 1> ===

      user()          CURRENT_USER
=====
      'PUBLIC@cubs006.cub'  'PUBLIC'

--selecting all users of the current database from the system table
SELECT name, id, password FROM db_user;

=== <Result of SELECT Command in Line 1> ===

      name                      id  password
=====
      'DBA'                     NULL  NULL
      'PUBLIC'                  NULL  NULL
      'SELECT_ONLY_USER'        NULL  db_password
      'ALMOST_DB_USER'          NULL  db_password
      'SELECT_ONLY_USER2'       NULL  NULL
```

DATABASE 함수, SCHEMA 함수

설명

DATABASE 함수와 **SCHEMA** 함수는 동일하며, 현재 연결된 데이터베이스 이름을 **VARCHAR** 타입의 스트링으로 반환한다.

구문

```
DATABASE()
```

```
SCHEMA()
```

예제

```
SELECT DATABASE(), SCHEMA();
;xr

=== <Result of SELECT Command in Line 1> ===

      database()           schema()
=====
'demodb'           'demodb'
```

DEFAULT 함수

설명

DEFAULT 함수는 컬럼에 정의된 디폴트 값을 반환한다. 해당 컬럼에 디폴트 값이 지정되지 않으면 **NULL** 또는 에러를 출력한다. 디폴트 값이 정의되지 않은 컬럼에 어떠한 제약 조건이 정의되어 있지 않거나 **UNIQUE** 제약 조건이 정의된 경우에는 **NULL**을 반환하고, 해당 컬럼에 **NOT NULL** 또는 **PRIMARY KEY** 제약 조건이 정의된 경우에는 에러를 반환한다.

구문

```
DEFAULT(column_name)
```

예제

```
CREATE TABLE info_tbl(id INT DEFAULT 0, name VARCHAR)
INSERT INTO info_tbl VALUES (1, 'a'), (2, 'b'), (NULL, 'c');

3 rows affected.

SELECT id, DEFAULT(id) FROM info_tbl;

=== <Result of SELECT Command in Line 1> ===

      id  default(id)
=====
      1             0
      2             0
  NULL             0
```

LIST_DBS 함수

설명

LIST_DBS()는 CUBRID 데이터베이스 서버에 존재하는 모든 데이터베이스 리스트를 공백 문자로 구분하여 출력한다.

구문

```
LIST_DBS()
```

예제

```
SELECT LIST_DBS();
```

```

=== <Result of SELECT Command in Line 1> ===

    dbs
=====
'testdb demodb'

```

ROW_COUNT 함수

설명

ROW_COUNT 함수는 이전 질의문에 의해 갱신(**UPDATE**, **INSERT**, **DELETE**)된 행의 개수를 정수로 반환한다.

ROW_COUNT 함수의 실행 영역은 해당 SQL이 생성된 클라이언트 세션에 한정되며, **;run** 또는 **;xrun**

명령어로 해당 SQL 문을 실행한 후에 이 함수를 호출하면 -1을 반환한다.

구문

```
ROW_COUNT()
```

예제

```

SELECT * FROM info_tbl;

=== <Result of SELECT Command in Line 1> ===

      id  name
=====
      1  'a'
      2  'b'
     NULL  'c'

INSERT INTO info_tbl VALUES (4,'d'), (5, 'e');
SELECT ROW_COUNT();
;xr

=== <Result of SELECT Command in Line 2> ===

    row_count()
=====
              2

DELETE FROM info_tbl WHERE id IN (4,5);
SELECT ROW_COUNT();
;xr

=== <Result of SELECT Command in Line 2> ===

    row_count()
=====
              2

SELECT ROW_COUNT();

=== <Result of SELECT Command in Line 1> ===

    row_count()
=====
            -1

```

조건 연산식과 함수

CASE

설명

CASE 연산식은 **IF ... THEN ... ELSE** 로직을 SQL 문장으로 표현하며, **WHEN**에 지정된 비교 연산 결과가 참이면 **THEN** 절의 값을 반환하고 거짓이면 **ELSE** 절에 명시된 값을 반환한다. 만약, **ELSE** 절이 없다면 **NULL** 값을 반환한다.

구문

```
CASE control_expression simple_when_list
[ else clause ]
END

CASE searched_when_list
[ else clause ]
END

simple when :
WHEN expression THEN result

searched_when :
WHEN search_condition THEN result

else clause :
ELSE result

result :
expression | NULL
```

CASE 문은 반드시 키워드 **END**로 끝나야 하며, *control_expression*과 데이터 타입과 *simple_when* 절 내의 *expression*은 비교 가능한 데이터 타입이어야 한다. 또한, **THEN**과 **ELSE** 절에 지정된 모든 *result*의 데이터 타입은 서로 같거나, 어느 하나의 공통 데이터 타입으로 변환 가능(convertible)해야 한다.

CASE 수식이 반환하는 값의 데이터 타입은 다음과 같은 규칙에 따라 결정된다.

- **THEN** 절에 명시된 모든 *result*의 데이터 타입이 같으면, 해당 타입이 리턴 값의 데이터 타입이 된다.
- 모든 *result*의 데이터 타입이 같지 않더라도 어느 하나의 공통 데이터 타입으로 변환 가능하면, 해당 타입이 리턴 값의 데이터 타입이 된다.
- *result* 중 어느 하나가 가변 길이 스트링인 경우, 리턴 값의 데이터 타입은 가변 길이 스트링이 된다. 또한, *result*가 모두 고정 길이 스트링인 경우에는 가장 긴 길이를 가지는 문자 스트링 또는 비트 스트링이 결과로 반환된다.
- *result* 중 어느 하나가 근사치로 표현되는 수치형이면, 근사치로 표현되고 이때 소수점 이하 자릿수는 모든 *result*의 유효 숫자를 표현할 수 있도록 결정된다.

예제

```
--creating a table
CREATE TABLE case_tbl ( a INT);
INSERT INTO case_tbl VALUES (1);
INSERT INTO case_tbl VALUES (2);
INSERT INTO case_tbl VALUES (3);
INSERT INTO case_tbl VALUES (NULL);
```

```

--case operation with a search when clause
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
FROM case_tbl;

=====
      1  'one'
      2  'two'
      3  'other'
     NULL 'other'

--case operation with a simple when clause
SELECT a,
       CASE a WHEN 1 THEN 'one'
            WHEN 2 THEN 'two'
            ELSE 'other'
       END
FROM case_tbl;

=====
      1  'one'
      2  'two'
      3  'other'
     NULL 'other'

--result types are converted to a single type containing all of significant figures
SELECT a,
       CASE WHEN a=1 THEN 1
            WHEN a=2 THEN 1.2345
            ELSE 1.234567890
       END
FROM case_tbl;

=====
      1  1.000000000
      2  1.234500000
      3  1.234567890
     NULL 1.234567890

--an error occurs when result types are not convertible
SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 1.2345
       END
FROM case_tbl;
=====
ERROR

```

COALESCE 함수

설명

COALESCE 함수는 하나 이상의 연산식 리스트가 인자로 지정되며, 첫 번째 인자가 **NULL**이 아닌 값이면 해당 값을 결과로 반환하고, **NULL**이면 두 번째 인자를 반환한다. 만약 인자로 지정된 모든 연산식이 **NULL**이면 **NULL**을 결과로 반환한다. 이러한 **COALESCE** 함수는 주로 **NULL** 값을 다른 기본 값으로 대체할 때 사용하며, 인자로 지정된 모든 연산식은 서로 동일하거나 변환 가능한 데이터 타입이어야 한다.

구문

```
COALESCE(expression [, ...])
```

```
result :  
expression | NULL
```

COALESCE(*a, b*)는 다음의 **CASE** 문장과 같은 의미를 가진다.

```
CASE WHEN a IS NOT NULL  
THEN a  
ELSE b  
END
```

예제

```
SELECT * FROM case_tbl;  
  
=== <Result of SELECT Command in Line 1> ===  
  
      a  
=====  
      1  
      2  
      3  
     NULL  
  
--substituting a default value 10.0000 for NULL valuse  
SELECT a, COALESCE(a, 10.0000) FROM case_tbl;  
  
=== <Result of SELECT Command in Line 1> ===  
  
      a  coalesce(a, 10.0000)  
=====
```

1	1.0000
2	2.0000
3	3.0000
NULL	10.0000

DECODE 함수

설명

DECODE 함수는 **CASE** 문과 마찬가지로 **IF ... THEN ... ELSE** 문과 동일한 기능을 수행한다. 인자로 지정된 *expression*과 *search*를 비교하여, 같은 값을 가지는 *search*에 대응하는 *result*를 결과로 반환한다. 만약, 같은 값을 가지는 *search*가 없다면 *default* 값을 반환하고, *default* 값이 생략된 경우에는 **NULL**을 반환한다. 비교 연산의 대상이 되는 *expression*과 *search*는 데이터 타입이 동일하거나 서로 변환 가능해야 하고, 지정된 모든 *result* 값의 유효 숫자를 포함하여 표현할 수 있도록 결과 값의 소수점 아래 자릿수가 결정된다.

구문

```
DECODE( expression, search, result [, search, result]* [, default] )  
  
result :  
result | default | NULL
```

DECODE(*a, b, c, d, e*)는 다음의 **CASE** 문장과 같은 의미를 가진다.

```
CASE WHEN a = b THEN c  
      WHEN a = c THEN d  
      ELSE e  
END
```

예제

```
SELECT * FROM case_tbl;
```

```

=== <Result of SELECT Command in Line 1> ===

      a
=====
      1
      2
      3
      NULL

--Using DECODE function to compare expression and search values one by one
SELECT a, DECODE(a, 1, 'one', 2, 'two', 'other') FROM case_tbl;

=====
      1 'one'
      2 'two'
      3 'other'
      NULL 'other'

--result types are converted to a single type containing all of significant figures
SELECT a, DECODE(a, 1, 1, 2, 1.2345, 1.234567890) FROM case_tbl;

=====
      1 1.000000000
      2 1.234500000
      3 1.234567890
      NULL 1.234567890

--an error occurs when result types are not convertible
SELECT a, DECODE(a, 1, 'one', 2, 'two', 1.2345) FROM case_tbl;

=====
ERROR

```

IF 함수

설명

IF 함수는 첫 번째 인자로 지정된 연산식의 값이 **TRUE**이면 *expression2*를 반환하고, **FALSE**이거나 **NULL**이면 *expression3*를 반환한다. 결과로 반환되는 *expression2*와 *expression3*은 데이터 타입이 동일하거나 공통의 타입으로 변환 가능해야 한다. 둘 중 하나가 명확하게 **NULL**이면, 함수의 결과 타입은 **NULL**이 아닌 인자의 타입을 따른다.

구문

```

IF( expression1, expression2, expression3 )

result :
expression2 | expression3

```

IF(*a*, *b*, *c*)는 다음의 **CASE** 문장과 같은 의미를 가진다.

```

CASE WHEN a IS TRUE THEN b
ELSE c
END

```

예제

```

SELECT * FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

      a
=====

```

```

1
2
3
NULL

--IF function returns the second expression when the first is TRUE
SELECT a, IF(a=1, 'one', 'other') FROM case_tbl;

a    if(a=1, 'one', 'other')
=====
1    'one'
2    'other'
3    'other'
NULL 'other'

--If function in WHERE clause
SELECT * FROM case_tbl WHERE IF(a=1, 1, 2) = 1;

a
=====
1

1 rows selected.

```

IFNULL, NVL 함수

설명

IFNULL 함수와 **NVL** 함수는 유사하게 동작하며, **NVL** 함수는 집합형 데이터 타입을 추가로 지원한다. 두 개의 인자가 지정되며, 첫 번째 인자 *expr1*이 **NULL**이 아니면 *expr1*을 반환하고, **NULL**이면 두 번째 인자인 *expr2*를 반환한다. 각 함수가 결과로 반환하는 데이터 타입은 *expr1*과 *expr2* 타입에 따라 아래의 표와 같다.

expr1 타입	expr2 타입	NVL 변환 값 타입	IFNULL 반환 값 타입
?	?	에러	VARCHAR
?	X	X 타입	VARCHAR
CHAR	VARCHAR	VARCHAR	VARCHAR
CHAR	NCHAR	에러	VARCHAR
VARCHAR	NCHAR	VARCHAR	VARCHAR
스트링 타입	숫자 타입	에러	VARCHAR
스트링 타입	날짜/시간 타입	에러	VARCHAR
숫자 타입	날짜/시간 타입	에러	VARCHAR
날짜/시간 타입	날짜/시간 타입	공통으로 변환 가능한 타입	VARCHAR
숫자 타입	숫자 타입	공통으로 변환 가능한 타입	공통으로 변환 가능한 타입
집합형 타입	집합형 타입	공통으로 변환 가능한 타입	에러
집합형 타입	그 외 타입	에러	에러

구문

```
IFNULL( expr1, expr2 )
NVL( expr1, expr2 )

result :
expr1 | expr2
```

IFNULL(a, b) 또는 **NVL(a, b)**는 다음의 **CASE** 문장과 같은 의미를 가진다.

```
CASE WHEN a IS NULL THEN b
ELSE a
END
```

예제

```
SELECT * FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

      a
=====
      1
      2
      3
      NULL

--returning a specific value when a is NULL
SELECT a, NVL(a, 10.0000) FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

      a  nvl(a, 10.0000)
=====
      1    1.0000
      2    2.0000
      3    3.0000
      NULL 10.0000

--IFNULL can be used instead of NVL and return values are converted to the string type
SELECT a, IFNULL(a, 'UNKNOWN') FROM case_tbl;

      a  ifnull(a, 'UNKNOWN')
=====
      1    '1'
      2    '2'
      3    '3'
      NULL 'UNKNOWN'
```

NULLIF 함수

설명

NULLIF 함수는 인자로 지정된 두 개의 연산식이 동일하면 **NULL**을 반환하고, 다르면 첫 번째 인자 값을 반환한다.

구문

```
NULLIF(expr1, expr2)

result :
expr1 | NULL
```

NULLIF(a, b)는 다음의 **CASE** 문장과 같은 의미를 가진다.

```
CASE
```

```

WHEN a = b THEN NULL
ELSE a
END

```

예제

```

SELECT * FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

      a
=====
      1
      2
      3
     NULL

--returning NULL value when a is 1
SELECT a, NULLIF(a, 1) FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

      a  nullif(a, 1)
=====
      1         NULL
      2          2
      3          3
     NULL        NULL

--returning NULL value when arguments are same
SELECT NULLIF (1, 1.000) FROM db_root;

=== <Result of SELECT Command in Line 1> ===

      nullif(1, 1.000)
=====
      NULL

--returning the first value when arguments are not same
SELECT NULLIF ('A', 'a') FROM db_root;

=== <Result of SELECT Command in Line 1> ===

      nullif('A', 'a')
=====
      'A'

```

NVL2 함수

설명

NVL2 함수는 세 개의 인자가 지정되며, 첫 번째 연산식(*expr1*)이 **NULL**이 아니면 두 번째 연산식(*expr2*)을 반환하고, **NULL**이면 세 번째 연산식(*expr3*)을 반환한다. 단, 결과로 반환되는 데이터타입은 *expr2*의 타입을 따르며, *expr3*의 타입 역시 *expr2*의 타입으로 변환된다.

구문

```

NVL2( expr1, expr2, expr3 )

result :
expr2 | expr3

```

예제

```

SELECT * FROM case_tbl;

```

```

=== <Result of SELECT Command in Line 1> ===

      a
=====
      1
      2
      3
     NULL

--returning a specific value of INT type
SELECT a, NVL2(a, a+1, 10.5678) FROM case_tbl;

=== <Result of SELECT Command in Line 1> ===

      a  nv12(a, a+1, 10.5678)
=====
      1                2
      2                3
      3                4
     NULL             11

```

조건식

단순 비교 조건식

조건식(conditional expression)은 **SELECT**, **UPDATE**, **DELETE** 문의 **WHERE** 절과 **SELECT** 문의 **HAVING** 절에 포함되는 표현식으로서, 결합되는 연산자의 종류에 따라 단순 비교 조건식, **ANY/SOME/ALL** 조건식, **BETWEEN** 조건식, **EXISTS** 조건식, **IN/NOT IN** 조건식, **LIKE** 조건식, **IS NULL** 조건식이 있다.

먼저, 단순 비교 조건식(simple comparison codition)은 두 개의 비교 가능한 데이터 값을 비교한다. 피연산자로 일반 연산식(expression) 또는 부질의(sub-query)가 지정되며, 피연산자 중 어느 하나가 **NULL**이면 항상 **NULL**을 반환한다. 단순 비교 조건식에서 사용할 수 있는 연산자는 아래의 표와 같으며, 보다 자세한 내용은 [비교 연산자](#)를 참고한다.

단순 비교 조건식에서 사용할 수 있는 연산자

비교 연산자	설명	조건식	리턴 값
=	왼쪽 및 오른쪽 피연산자의 값이 같다.	1=2	0
<>, !=	왼쪽 및 오른쪽 피연산자의 값이 다르다.	1<>2	1
>	왼쪽 피연산자는 오른쪽 피연산자보다 값이 크다.	1>2	0
<	왼쪽 피연산자는 오른쪽 피연산자보다 값이 작다.	1<2	1
>=	왼쪽 피연산자는 오른쪽 피연산자보다 값이 크거나 같다.	1>=2	0
<=	왼쪽 피연산자는 오른쪽 피연산자보다 값이 작거나 같다.	1<=2	1

ANY/SOME/ALL 수량어와 그룹 조건식

설명

ANY/SOME/ALL과 같은 수량어를 포함하는 그룹 조건식은 하나의 데이터 값과 리스트에 포함된 값들의 일부 또는 모든 값에 대해서 비교 연산을 수행한다. 즉, **ANY** 또는 **SOME**이 포함된 그룹 조건식은, 왼쪽의 데이터 값이 오른쪽 피연산자로 지정된 리스트 내의 값 중 최소한 하나에 대해 단순 비교 연산자를 만족할 때 **TRUE**를 반환한다. 한편, **ALL**이 포함된 그룹 조건식의 경우, 왼쪽 데이터 값이 오른쪽 리스트 내의 모든 값들에 대해 단순 비교 연산자를 만족할 때 **TRUE**를 반환한다.

만약, **ANY** 또는 **SOME**을 포함하는 그룹 조건식에서 **NULL**을 대상으로 비교 연산을 수행하면 그룹 조건식의 결과로 **UNKNOWN** 또는 **TRUE**를 반환하고, **ALL**을 포함하는 그룹 조건식에서 **NULL**을 대상으로 비교 연산을 수행하면 **UNKNOWN** 또는 **FALSE**를 반환한다.

구문

```
expression comp_op SOME expression
expression comp_op ANY expression
expression comp_op ALL expression
```

- comp_op*: 비교 연산자 >, <, =, >=, <=가 들어갈 수 있다.
- expression* (왼쪽): 단일 값을 가지는 컬럼, 경로 표현식, 상수 값 또는 단일 값을 생성하는 산술 함수가 될 수 있다.
- expression* (오른쪽): 컬럼 이름, 경로 표현식, 상수 값의 리스트(집합), 부질의를 될 수 있다. 리스트는 중괄호({}) 안에 표현된 집합을 의미하며, 부질의가 사용되면 부질의의 수행 결과 전부에 대해서 *expression* (왼쪽)와 비교 연산을 수행한다.

예제

```
--creating a table

CREATE TABLE condition_tbl (id int primary key, name char(10), dept_name VARCHAR, salary
INT);
INSERT INTO condition_tbl VALUES(1, 'Kim', 'devel', 4000000);
INSERT INTO condition_tbl VALUES(2, 'Moy', 'sales', 3000000);
INSERT INTO condition_tbl VALUES(3, 'Jones', 'sales', 5400000);
INSERT INTO condition_tbl VALUES(4, 'Smith', 'devel', 5500000);
INSERT INTO condition_tbl VALUES(5, 'Kim', 'account', 3800000);
INSERT INTO condition_tbl VALUES(6, 'Smith', 'devel', 2400000);
INSERT INTO condition_tbl VALUES(7, 'Brown', 'account', NULL);

--selecting rows where department is sales or devel
SELECT * FROM condition_tbl WHERE dept_name = ANY({'devel','sales'});

=== <Result of SELECT Command in Line 1> ===

      id  name      dept_name      salary
=====
      1  'Kim      '      'devel'      4000000
      2  'Moy      '      'sales'      3000000
      3  'Jones     '      'sales'      5400000
      4  'Smith     '      'devel'      5500000
      6  'Smith     '      'devel'      2400000

--selecting rows comparing NULL value in the ALL group conditions
SELECT * FROM condition_tbl WHERE salary > ALL(3000000, 4000000, NULL);
```

```

=== <Result of SELECT Command in Line 1> ===

There are no results.

--selecting rows comparing NULL value in the ANY group conditions
SELECT * FROM condition_tbl WHERE salary > ANY{3000000, 4000000, NULL};

=== <Result of SELECT Command in Line 1> ===

=====
id  name      dept_name      salary
=====
1   'Kim'      'devel'        4000000
3   'Jones'    'sales'        5400000
4   'Smith'    'devel'        5500000
5   'Kim'      'account'      3800000

--selecting rows where salary*0.9 is less than those salary in devel department
SELECT * FROM condition_tbl WHERE (
(0.9 * salary) < ALL (SELECT salary FROM condition_tbl
WHERE dept_name = 'devel')
);

=== <Result of SELECT Command in Line 1> ===

=====
id  name      dept_name      salary
=====
6   'Smith'    'devel'        2400000

```

BETWEEN 조건식

설명

BETWEEN 조건식은 왼쪽의 데이터 값이 오른쪽에 지정된 두 데이터 값 사이에 존재하는지 비교한다. 이때, 왼쪽의 데이터 값이 비교 대상 범위의 경계값과 동일한 경우에도 **TRUE**를 반환한다. 한편, **BETWEEN** 키워드 앞에 **NOT**이 오면 **BETWEEN** 연산의 결과에 **NOT** 연산을 수행하여 결과를 반환한다.

i **BETWEEN** g **AND** m 은 복합 조건식 $i \geq g$ **AND** $i \leq m$ 과 동일하다.

구문

```
expression [ NOT ] BETWEEN expression AND expression
```

- expression*: 컬럼 이름, 경로 표현식, 상수 값, 산술 표현식, 집계 함수가 될 수 있다. 문자열 표현식인 경우에는 문자의 사전순으로 조건이 평가된다. 표현식 중 하나라도 **NULL**이 지정되면 **BETWEEN** 조건식의 결과는 **FALSE** 또는 **UNKNOWN**을 반환한다.

예제

```

--selecting rows where 3000000 <= salary <= 4000000
SELECT * FROM condition_tbl WHERE salary BETWEEN 3000000 AND 4000000;

SELECT * FROM condition_tbl WHERE (salary >= 3000000) AND (salary <= 4000000);

=== <Result of SELECT Command in Line 1> ===

=====
id  name      dept_name      salary
=====
1   'Kim'      'devel'        4000000
2   'Moy'      'sales'        3000000
5   'Kim'      'account'      3800000

--selecting rows where salary < 3000000 or salary > 4000000
SELECT * FROM condition_tbl WHERE salary NOT BETWEEN 3000000 AND 4000000;

```

```

=== <Result of SELECT Command in Line 1> ===

      id  name      dept name      salary
=====
      3  'Jones    '      'sales'      5400000
      4  'Smith    '      'devel'      5500000
      6  'Smith    '      'devel'      2400000

--selecting rows where name starts from A to E
SELECT * FROM condition_tbl WHERE name BETWEEN 'A' AND 'E';

=== <Result of SELECT Command in Line 1> ===

      id  name      dept name      salary
=====
      7  'Brown    '      'account'     NULL

```

EXISTS 조건식

설명

EXISTS 조건식은 오른쪽에 지정되는 부질의를 실행한 결과가 하나 이상 존재하면 **TRUE**를 반환하고, 연산 실행 결과가 공집합이면 **FALSE**를 반환한다.

구문

EXISTS *expression*

- *expression*: 부질의가 지정되며, 부질의 실행 결과가 존재하는지 비교한다. 만약 부질의가 어떤 결과도 만들지 않는다면 조건식 결과는 **FALSE**이다.

예제

```

--selecting rows using EXISTS and subquery
SELECT 'raise' FROM db root WHERE EXISTS(
SELECT * FROM condition_tbl WHERE salary < 2500000);

=== <Result of SELECT Command in Line 2> ===

      'raise'
=====
      'raise'

--selecting rows using NOT EXISTS and subquery
SELECT 'raise' FROM db root WHERE NOT EXISTS(
SELECT * FROM condition_tbl WHERE salary < 2500000);

=== <Result of SELECT Command in Line 2> ===

There are no results.

0 rows selected.

```

IN 조건식

설명

IN 조건식은 왼쪽의 단일 데이터 값이 오른쪽에 지정된 리스트 내에 포함되어 있는지 비교한다. 즉, 왼쪽의 단일 데이터 값이 오른쪽에 지정된 표현식의 원소이면 **TRUE**를 반환한다. **IN** 키워드 앞에 **NOT**이 있으면 **IN** 연산의 결과에 **NOT** 연산을 수행하여 결과를 반환한다.

구문

```
expression [ NOT ] IN expression
```

- *expression* (left) : 단일 값을 가지는 컬럼, 경로 표현식, 상수 값 또는 단일 값을 생성하는 산술 함수가 될 수 있다.
- *expression* (right) : 컬럼 이름, 경로 표현식, 상수 값의 리스트(집합), 부질의가 될 수 있다. 리스트는 소괄호(()) 또는 중괄호({}) 안에 표현된 집합을 의미하며, 부질의가 사용되면 부질의의 수행 결과 전부에 대해서 *expression* (left)와 비교 연산을 수행한다.

예제

```
--selecting rows where department is sales or devel
SELECT * FROM condition_tbl WHERE dept_name IN {'devel','sales'};

SELECT * FROM condition_tbl WHERE dept_name = ANY({'devel','sales'});

=== <Result of SELECT Command in Line 1> ===

=====
id  name      dept_name      salary
=====
1   'Kim      '      'devel'      4000000
2   'Moy      '      'sales'      3000000
3   'Jones    '      'sales'      5400000
4   'Smith    '      'devel'      5500000
6   'Smith    '      'devel'      2400000

--selecting rows where department is neither sales nor devel
SELECT * FROM condition_tbl WHERE dept_name NOT IN {'devel','sales'};

=== <Result of SELECT Command in Line 1> ===

=====
id  name      dept_name      salary
=====
5   'Kim      '      'account'    3800000
7   'Brown    '      'account'    NULL
```

IS NULL 조건식

설명

IS NULL 조건식은 왼쪽에 지정된 표현식의 결과가 **NULL**인지 비교하여, **NULL**인 경우 **TRUE**를 반환하며, 조건절 내에서 사용할 수 있다. **NULL** 키워드 앞에 **NOT**이 있으면 **IS NULL** 연산의 결과에 **NOT** 연산을 수행하여 결과를 반환한다.

구문

```
expression IS [ NOT ] NULL
```

- *expression* : 단일 값을 가지는 컬럼, 경로 표현식, 상수 값 또는 단일 값을 생성하는 산술 함수가 될 수 있다.

예제

```
--selecting rows where salary is NULL
SELECT * FROM condition_tbl WHERE salary IS NULL;

=== <Result of SELECT Command in Line 1> ===

=====
id  name      dept_name      salary
=====
```

```

=====
          7  'Brown'      'account'      NULL

--selecting rows where salary is NOT NULL
SELECT * FROM condition tbl WHERE salary IS NOT NULL;

=== <Result of SELECT Command in Line 1> ===

      id  name      dept_name      salary
=====
      1  'Kim'      'devel'      4000000
      2  'Moy'      'sales'      3000000
      3  'Jones'    'sales'      5400000
      4  'Smith'    'devel'      5500000
      5  'Kim'      'account'    3800000
      6  'Smith'    'devel'      2400000

--simple comparison operation returns NULL when operand is NULL
SELECT * FROM condition tbl WHERE salary = NULL;

=== <Result of SELECT Command in Line 1> ===

There are no results.

0 rows selected.

```

ISNULL 함수

설명

ISNULL 함수는 조건절 내에서 사용할 수 있으며, 인자로 지정된 표현식의 결과가 **NULL**인지 비교하여 **NULL**이면 1을 반환하고, 아니면 0을 반환한다. 이 함수를 이용하여 어떤 값이 **NULL**인지 아닌지를 테스트할 수 있으며, **IS NULL** 조건식과 유사하게 동작한다.

구문

ISNULL(*expression*)

- expression*: 단일 값을 가지는 컬럼, 경로 표현식, 상수 값 또는 단일 값을 생성하는 산술 함수를 입력한다.

예제

```

--Using ISNULL function to select rows with NULL value
SELECT * FROM condition tbl WHERE ISNULL(salary);

=== <Result of SELECT Command in Line 1> ===

      id  name      dept name      salary
=====
      7  'Brown'    'account'      NULL

```

LIKE 조건식

설명

LIKE 조건식은 문자열 데이터 간의 패턴을 비교하는 연산을 수행하여, 검색어와 일치하는 패턴의 문자열이 검색되면 **TRUE**를 반환한다. 패턴 비교 대상이 되는 도메인은 **CHAR**, **VARCHAR**, **STRING**이며, **NCHAR**

또는 **BIT** 타입에 대해서는 **LIKE** 검색을 수행할 수 없다. **LIKE** 키워드 앞에 **NOT**이 있으면 **LIKE** 연산의 결과에 **NOT** 연산을 수행하여 결과를 반환한다.

LIKE 연산자 오른쪽에 오는 검색어에는 임의의 문자 또는 문자열에 대응되는 와일드 카드(wild card) 문자열을 포함할 수 있으며, %(percent)와 _(underscore)를 사용할 수 있다. %는 길이가 0 이상인 임의의 문자열에 대응되며, _는 1개의 문자에 대응된다. 또한, 이스케이프 문자(escape character)는 와일드 카드 문자 자체에 대한 검색을 수행할 때 사용되는 문자로서, 사용자에게 의해 길이가 1인 다른 문자로 지정될 수 있다. 와일드 카드 문자 또는 이스케이프 문자를 포함하는 문자열을 검색어로 사용하는 예제는 아래를 참고한다.

구문

```
expression [ NOT ] LIKE expression [ ESCAPE char]
```

- *expression* (left) : 문자 스트링 데이터 타입 컬럼이 지정된다. 패턴 비교는 컬럼 값의 첫 번째 문자부터 시작되며, 대소문자를 구분한다.
- *expression* (right) : 검색어를 입력하며, 길이가 0 이상인 문자열이 된다. 이때, 검색어 패턴에는 와일드 카드 문자(% 또는 _)가 포함될 수 있다. 문자열의 길이는 0 이상이다.
- **ESCAPE char** : 만약 검색어의 문자열 패턴이 "_" 또는 "%" 자체를 포함하는 경우 이스케이프 문자가 반드시 지정되어야 한다. 예를 들어, 이스케이프 문자를 백슬래시(\)로 지정한 후 '10%'인 문자열을 검색하고자 한다면, *expression* (right)에 '10\%'을 지정해야 한다. 또한, 'C:\'인 문자열을 검색하고자 한다면, *expression* (right)에 'C:\\'을 지정하면 된다.

참고 사항

UTF-8과 같은 멀티바이트 문자 세트 환경에서 입력된 데이터에 대해서는 **LIKE** 검색 결과가 정상적이지 않을 수 있다. 이는 문자 세트에 따라 문자열 비교 연산을 수행하는 바이트 단위가 다르기 때문이며, 1바이트 단위로 문자열 비교를 수행하도록 하는 파라미터(single_byte_compare=yes)를 **cubrid.conf** 파일에 추가한 후 DB를 재가동하면 정상적인 검색 결과를 얻을 수 있다.

CUBRID가 지원하는 문자 세트에 관한 상세한 설명은 [문자 스트링 데이터 타입](#)을 참고하며, **single_byte_compare** 파라미터에 관한 상세한 설명은 [기타 파라미터](#)를 참고한다.

예제

```
--selection rows where name contains lower case 's', not upper case
SELECT * FROM condition_tbl WHERE name LIKE '%s%';

=== <Result of SELECT Command in Line 1> ===

      id  name      dept_name      salary
=====
      3  'Jones'    'sales'      5400000

--selection rows where second letter is 'O' or 'o'
SELECT * FROM condition_tbl WHERE UPPER(name) LIKE ' O%';

=== <Result of SELECT Command in Line 1> ===

      id  name      dept name      salary
=====
```

```

      2 'Moy      '      'sales'      3000000
      3 'Jones    '      'sales'      5400000

--selection rows where name is 3 characters
SELECT * FROM condition_tbl WHERE name LIKE '___';

=== <Result of SELECT Command in Line 1> ===

      id  name      dept name      salary
=====
      1  'Kim      '      'devel'      4000000
      2  'Moy      '      'sales'      3000000
      5  'Kim      '      'account'    3800000

```

데이터 조회 및 조작

SELECT

개요

설명

SELECT 문은 지정된 테이블에서 원하는 컬럼을 조회한다.

구문

```
SELECT [ <qualifier> ] <select_expressions>
[ { TO | INTO } <variable_comma_list> ]
[ FROM <extended_table_specification_comma_list> ]
[ WHERE <search_condition> ]
[ GROUP BY { col_name | expr } [ ASC | DESC ],...[ WITH ROLLUP ] ]
[ HAVING <search_condition> ]
[ ORDER BY { col_name | expr } [ ASC | DESC ],... [ FOR <orderby_for_condition> ] ]
[ LIMIT [offset,] row count ]
[ USING INDEX { index name [,index_name,...] | NONE }]

<qualifier> ::= ALL | DISTINCT | DISTINCTROW | UNIQUE

<select_expressions> ::= * | <expression_comma_list>

<extended table specification comma list> ::=
<table specification> [ {, <table specification> | <join table specification> }... ]

<table specification> ::=
<single_table_spec> [ <correlation> ] [ WITH (lock_hint) ] |
<metaclass specification> [ <correlation> ] |
<subquery> <correlation> |
TABLE ( <expression> ) <correlation>

<correlation> ::= [ AS ] <identifier> [ ( <identifier_comma_list> ) ]

<single_table_spec> ::= [ ONLY ] <table_name> |
ALL <table name> [ EXCEPT <table name> ]

<metaclass specification> ::= CLASS <class_name>

<join table specification> ::=
[ INNER | [ LEFT | RIGHT [ OUTER ] ] JOIN <table specification> ON <search condition>

lock hint :
READ UNCOMMITTED

<orderby_for_condition> ::=
ORDERBY_NUM() { BETWEEN int AND int } |
{ { = | =< | < | > | >= } int } |
IN ( int, ...)
```

- *qualifier*: 한정어. 생략이 가능하며 지정하지 않을 경우에는 ALL로 지정된다.
 - ALL: 테이블의 모든 레코드를 조회한다.
 - DISTINCT: 중복을 허용하지 않고 유일한 값을 갖는 레코드에 대해서만 조회한다.
DISTINCTROW와 동일하다.
 - UNIQUE: DISTINCT와 동일하게 중복을 허용하지 않고 유일한 값을 갖는 레코드에 대해서만 조회한다.

- *select_expression* :
 - * : **SELECT** * 구문을 사용하면 **FROM** 절에서 명시한 테이블에 대한 모든 컬럼을 조회할 수 있다.
 - *expression_comma_list* : *expression*은 컬럼 이름이나 경로 표현식, 변수, 테이블 이름이 될 수 있으며 산술 연산을 포함하는 일반적인 표현식도 모두 사용될 수 있다. 쉼표(,)는 리스트에서 개별 표현식을 구분하는데 사용된다. 조회하고자 하는 컬럼 또는 연산식에 대해 **AS** 키워드를 사용하여 별칭(alias)를 지정할 수 있으며, 지정된 별칭은 컬럼 이름으로 사용되어 **GROUP BY**, **HAVING**, **ORDER BY**, **FOR** 절 내에서 사용될 수 있다. 컬럼의 위치 인덱스(position)는 컬럼이 명시된 순서대로 부여되며, 시작 값은 1이다.

expression에는 **AVG**, **COUNT**, **MAX**, **MIN**, **SUM**과 같이 조회된 데이터를 조작하는 집계 함수가 사용될 수 있다. 만약 *expression*으로 집계 함수가 사용되는 경우, 집계 함수는 하나의 결과만 반환하기 때문에 **SELECT** 컬럼 리스트에 집계 함수로 그룹되지 않은 일반 컬럼이 명시될 수 없다.
 - *table_name*. * : 테이블 이름을 지정한다. *을 사용하면 명시한 테이블에 대한 모든 컬럼을 지정하는 것과 같다.
 - *variable* : *select_expression*이 조회하는 데이터는 하나 이상의 변수에 저장될 수 있다.
 - *[:]identifier* : **TO**(또는 **INTO**) 다음에 :식별자를 쓰면 조회하는 데이터를 ':identifier'의 변수에 저장할 수 있다.

예제 1

다음은 역대 올림픽이 개최된 국가를 중복 없이 조회한 예제이다. 이 예제는 demodb의 olympic 테이블을 대상으로 수행하였다. **DISTINCT** 또는 **UNIQUE** 키워드는 질의 결과가 유일한 값만을 갖도록 만든다. 예를 들어 host_nation 값이 'Greece'인 olympic 인스턴스가 여러 개일 때 질의 결과에는 하나의 값만 나타나도록 할 경우에 사용된다.

```
SELECT DISTINCT host_nation FROM olympic;

=== <Result of SELECT Command in Line 1> ===
  host_nation
=====
'Australia'
'Belgium'
'Canada'
'Finland'
'France'
...
18 rows selected.
```

예제 2

다음은 조회하고자 하는 컬럼에 컬럼 별칭을 부여하고, **ORDER BY** 절에서 컬럼 별칭을 이용하여 결과 레코드를 정렬하는 예제이다. 이때, **LIMIT** 절 및 **FOR ORDERBY_NUM** 함수를 사용하여 결과 레코드 수를 5개로 제한한다.

```
SELECT host_year as col1, host_nation as col2 FROM olympic ORDER BY col2 LIMIT 5;

=== <Result of SELECT Command in Line 1> ===
      col1  col2
=====
      2000  'Australia'
      1956  'Australia'
```

```

1920 'Belgium'
1976 'Canada'
1948 'England'

5 rows selected.

SELECT CONCAT(host_nation, ', ', host_city) AS host_place FROM olympic
ORDER BY host_place FOR ORDERBY NUM() BETWEEN 1 AND 5;
;xr

=== <Result of SELECT Command in Line 1> ===

host_place
=====
'Australia, Melbourne'
'Australia, Sydney'
'Belgium, Antwerp'
'Canada, Montreal'
'England, London'

5 rows selected.

```

FROM 절

일반

설명

FROM 절은 질의에서 데이터를 조회하고자 하는 테이블을 지정한다. 어떤 테이블도 참조되지 않는 경우에는 **FROM** 절을 생략할 수도 있다. 조회할 수 있는 경로는 다음과 같다.

- 개별 테이블
- 부질의
- 유도 테이블

구문

```

SELECT [ <qualifier> ] <select_expressions>
        [ FROM <table_specification> [ {, <table_specification>
| <join table_specification> }... ] ]

<select_expressions> ::= * | <expression comma list>

<table_specification> ::=
<single_table_spec> [ <correlation> ] [ WITH (lock_hint) ] |
<metaclass_specification> [ <correlation> ] |
<subquery> <correlation> |
TABLE ( <expression> ) <correlation>

<correlation> ::= [ AS ] <identifier> [ ( <identifier_comma_list> ) ]

<single_table_spec> ::= [ ONLY ] <table_name> |
                        ALL <table_name> [ EXCEPT <table_name> ]

<metaclass_specification> ::= CLASS <class_name>

lock_hint ::=
READ UNCOMMITTED

```

- *select_expressions*: 조회하고자 하는 컬럼 또는 연산식을 하나 이상 지정할 수 있으며, 테이블 내 모든 컬럼을 조회할 때에는 *를 지정한다. 조회하고자 하는 컬럼 또는 연산식에 대해 **AS** 키워드를 사용하여

별칭(alias)를 지정할 수 있으며, 지정된 별칭은 컬럼 이름으로 사용되어 **GROUP BY, HAVING, ORDER BY, FOR** 절 내에서 사용될 수 있다. 컬럼의 위치 인덱스(position)는 컬럼이 명시된 순서대로 부여되며, 시작 값은 1이다.

- *table_specification*: **FROM** 절 뒤에 하나 이상의 테이블 이름이 명시되며, 부질의와 유도 테이블도 지정될 수 있다. 부질의 유도 테이블에 대한 설명은 [부질의 유도 테이블](#)을 참고한다.
- *lock_hint*: 해당 테이블에 대한 격리 수준(isolation level)을 **READ UNCOMMITTED** 수준으로 설정할 수 있다. **READ UNCOMMITTED**은 오손 읽기(dirty read)가 발생할 수 있는 격리 수준으로서, CUBRID 트랜잭션의 격리 수준에 관한 자세한 설명은 [트랜잭션 격리 수준](#)을 참고한다.

예제

```
--FROM clause can be omitted in the statement
SELECT 1+1 AS sum value;
;xr

=== <Result of SELECT Command in Line 1> ===

      sum value
=====
              2

1 rows selected.

--db root can be used as a dummy table
SELECT 1+1 AS sum_value;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

      sum_value
=====
              2

1 rows selected.

SELECT CONCAT('CUBRID', '2008' , 'R3.0') AS db version;
;xr

=== <Result of SELECT Command in Line 1> ===

      db version
=====
'CUBRID2008R3.0'
```

유도 테이블

질의문에서 **FROM** 절의 테이블 명세 부분에 부질의가 사용될 수 있다. 이런 형태의 부질의는 부질의 결과가 테이블로 취급되는 유도 테이블(derived table)을 만든다. 유도 테이블을 만드는 부질의가 사용될 때 반드시 상관 명세가 사용되어야 한다.

또한 유도 테이블은 집합 값을 갖는 속성의 개별 원소를 접근하는데 사용된다. 이 경우 집합 값의 한 원소는 유도 테이블에서 하나의 레코드로 생성된다.

부질의 유도 테이블

설명

유도 테이블의 각 레코드는 **FROM** 절에 주어진 부질의 결과로부터 만들어진다. 부질의로부터 생성되는 유도 테이블은 임의의 개수의 컬럼과 레코드를 가질 수 있다.

구문

```
FROM (subquery) [ AS ] derived_table_name [( column_name [ (, column_name )_ ] )]
```

- *column_name* 파라미터의 개수와 *subquery*에서 만들어지는 컬럼의 개수는 일치해야 한다.

예제 1

다음은 한국이 획득한 금메달 개수와 일본이 획득한 은메달 개수를 더한 값을 조회하는 예제이다. 이 예제는 유도 테이블을 이용하여 부질의 중간 결과를 모으고 하나의 결과로 처리하는 방법을 보여준다. 이 질의는 *nation_code* 컬럼이 'KOR'인 gold값과 *nation_code* 컬럼이 'JPN'인 silver 값의 전체 합을 반환한다.

```
SELECT SUM(n) FROM (SELECT gold FROM participant WHERE nation_code='KOR'
UNION ALL SELECT silver FROM participant WHERE nation_code='JPN') AS t(n);
=== <Result of SELECT Command in Line 2> ===
      sum(n)
=====
          82
1 rows selected.
```

예제 2

부질의 유도 테이블은 외부 질의와 연관되어 있을 때 유용하게 사용할 수 있다. 예를 들어 **WHERE** 절에서 사용된 부질의 **FROM** 절에 유도 테이블이 사용될 수 있다.

다음은 은메달 및 동메달을 하나 이상 획득한 경우, 해당 은메달과 동메달의 합의 평균보다 많은 수의 금메달을 획득한 *nation_code*, *host_year*, *gold* 필드를 보여주는 쿼리 예제이다. 이 예제에서는 질의(외부 **SELECT** 절)와 부질의(내부 **SELECT** 절)가 *nation_code* 속성으로 연결되어 있다.

```
SELECT nation_code, host_year, gold
FROM participant p
WHERE gold > ( SELECT AVG(s)
              FROM ( SELECT silver + bronze
                    FROM participant
                    WHERE nation_code = p.nation_code
                    AND silver > 0
                    AND bronze > 0
                  ) AS t(s));
=== <Result of SELECT Command in Line 1> ===
nation_code      host_year      gold
=====
'JPN'              2004              16
'CHN'              2004              32
'DEN'              1996              4
'ESP'              1992              13
4 rows selected.
```

WHERE 절

설명

질의에서 컬럼은 조건에 따라 처리될 수 있다. **WHERE** 절은 조회하려는 데이터의 조건을 명시한다.

구문

```
WHERE search_condition
```

```
search_condition :  
• comparison predicate  
• between predicate  
• exists predicate  
• in predicate  
• null predicate  
• like predicate  
• quantified predicate  
• set predicate
```

WHERE 절은 *search_condition* 또는 질의에서 조회되는 데이터를 결정하는 조건식을 지정한다. 조건식이 참인 데이터만 질의 결과로 조회된다(**NULL** 값은 알 수 없는 값으로서 질의 결과로 조회되지 않는다).

- *search_condition* : 자세한 내용은 다음의 항목을 참고한다.
- [단순 비교 조건식](#)
- [BETWEEN 조건식](#)
- [EXISTS 조건식](#)
- [IN 조건식](#)
- [IS NULL 조건식](#)
- [LIKE 조건식](#)
- [ANY/SOME/ALL 조건식](#)

복수의 조건은 논리연산자 **AND**, **OR**를 사용할 수 있다. **AND**가 지정된 경우 모든 조건이 참이어야 하고, **OR**로 지정된 경우에는 하나의 조건만 참이어도 된다. 만약 키워드 **NOT**이 조건 앞에 붙는다면 조건은 반대의 의미를 갖는다. 논리 연산이 평가 되는 순서는 다음 표와 같다.

우선순위	연산자	기능
------	-----	----

1	()	괄호 내에 포함된 논리 표현식은 첫 번째로 평가된다.
---	-----	-------------------------------

2	NOT	논리 표현식의 결과를 부정한다.
---	------------	-------------------

3	AND	논리 표현식에 포함된 모든 조건이 참이어야 한다.
---	------------	-----------------------------

4	OR	논리 표현식에 포함된 조건 중 하나의 조건은 참이어야 한다.
---	-----------	-----------------------------------

GROUP BY ... HAVING 절

설명

SELECT 문으로 검색한 결과를 특정 컬럼을 기준으로 그룹화하기 위해 **GROUP BY** 절을 사용하며, 그룹별로 정렬을 수행하거나 집계 함수를 사용하여 그룹별 집계를 구할 때 사용한다. 그룹이란 **GROUP BY** 절에 명시된 컬럼에 대해 동일한 컬럼 값을 가지는 레코드들을 의미한다.

GROUP BY 절 뒤에 **HAVING** 절을 결합하여 그룹 선택을 위한 조건식을 설정할 수 있다. 즉, **GROUP BY** 절로 구성되는 모든 그룹 중 **HAVING** 절에 명시된 조건식을 만족하는 그룹만 조회한다.

SQL 표준에서는 **GROUP BY** 절에서 명시되지 않은 컬럼(hidden column)을 **SELECT** 컬럼 리스트에 명시할 수 없지만, CUBRID는 문법을 확장하여 **GROUP BY** 절에서 명시되지 않은 컬럼도 **SELECT** 컬럼 리스트에 명시할 수 있다. CUBRID에서 확장된 문법을 사용하지 않으려면 **only_full_group_by** 파라미터 값을 yes로 설정해야 한다. 이에 대한 자세한 내용은 [구문/타입 관련 파라미터](#)를 참고한다.

구문

```
SELECT ...  
GROUP BY { col_name | expr | position } [ ASC | DESC ],... [ WITH ROLLUP ] [ ORDER BY NULL ]  
[ HAVING <search_condition> ]
```

- *col_name | expr | position* : 하나 이상의 컬럼 이름, 표현식 또는 별칭이 지정될 수 있으며, 각 항목은 쉼표로 구분된다. 이를 기준으로 컬럼들이 정렬된다.
- *[ASC | DESC]* : **GROUP BY** 절 내에 명시된 컬럼 뒤에 **ASC** 또는 **DESC**의 정렬 옵션을 명시할 수 있다. 정렬 옵션이 명시되지 않으면 디폴트는 **ASC**가 된다.
- *search_condition* : **HAVING** 절에 검색 조건식을 명시한다. **HAVING** 절에는 **GROUP BY** 절 내에 명시된 컬럼, 별칭, 집계 함수에서 사용되는 컬럼 또는 **GROUP BY** 절에서 명시되지 않은 컬럼(hidden columns)을 참조할 수 있다.
- **WITH ROLLUP** : **GROUP BY** 절에 **WITH ROLLUP** 수정자를 명시하면, **GROUP BY**된 컬럼 각각에 대한 결과 값이 그룹별로 집계되고, 모든 결과 행에 대한 결과 값이 마지막 행에 추가로 출력된다.
- **ORDER BY NULL** : **GROUP BY** 절에 **ORDER BY NULL** 수정자를 명시하면, **GROUP BY**에 의한 정렬 오버헤드를 회피할 수 있다.

예제

```
--creating a new table  
CREATE TABLE sales tbl  
(dept_no int, name VARCHAR(20) PRIMARY KEY, sales_month int, sales_amount int DEFAULT 100);  
INSERT INTO sales tbl VALUES  
(201, 'George' , 1, 450),  
(201, 'Laura' , 2, 500),  
(301, 'Max' , 4, 300),  
(501, 'Stephan', 4, DEFAULT),  
(501, 'Chang' , 5, 150),  
(501, 'Sue' , 6, 150),  
(NULL, 'Yoka' , 4, NULL);  
  
--selecting rows grouped by dept_no with ORDER BY NULL modifier  
SELECT dept_no, avg(sales amount) FROM sales tbl  
GROUP BY dept_no ORDER BY NULL;  
;xr
```

=== <Result of SELECT Command in Line 1> ===

dept no	avg(sales amount)
NULL	NULL
201	475
301	300
501	133

--conditions in WHERE clause operate first before GROUP BY
SELECT dept_no, avg(sales amount) FROM sales_tbl
WHERE sales_amount > 100 GROUP BY dept_no;
;xr

=== <Result of SELECT Command in Line 1> ===

dept no	avg(sales amount)
201	475
301	300
501	150

--conditions in HAVING clause operate last after GROUP BY
SELECT dept_no, avg(sales amount) FROM sales_tbl
WHERE sales_amount > 100 GROUP BY dept_no HAVING avg(sales_amount) > 200;
;xr

=== <Result of SELECT Command in Line 1> ===

dept_no	avg(sales amount)
201	475
301	300

--selecting and sorting rows with using column alias
SELECT dept_no AS a1, avg(sales amount) AS a2 FROM sales_tbl
WHERE sales_amount > 200 GROUP BY a1 HAVING a2 > 200 ORDER BY a2;
;xr

=== <Result of SELECT Command in Line 1> ===

a1	a2
301	300
201	475

2 rows selected.

--selecting rows grouped by dept no with WITH ROLLUP modifier
SELECT dept_no AS a1, name AS a2, avg(sales amount) AS a3 FROM sales_tbl
WHERE sales_amount > 100 GROUP BY a1,a2 WITH ROLLUP;
;xr

=== <Result of SELECT Command in Line 1> ===

a1	a2	a3
201	'George'	450
201	'Laura'	500
201	NULL	475
301	'Max'	300
301	NULL	300
501	'Chang'	150
501	'Sue'	150
501	NULL	150
NULL	NULL	310

9 rows selected.

ORDER BY 절

설명

ORDER BY 절은 질의 결과를 오름차순 또는 내림차순으로 정렬하며, **ASC** 또는 **DESC**와 같은 정렬 옵션을 명시하지 않으면 디폴트로 오름차순으로 정렬한다. **ORDER BY** 절을 지정하지 않으면, 조회되는 레코드의 순서는 질의에 따라 다르다.

구문

```
SELECT ...  
ORDER BY {col_name | expr | position} [ASC | DESC],...]  
[ FOR <orderby_for_condition> ] ]  
  
<orderby_for_condition> ::=  
ORDERBY_NUM() { BETWEEN int AND int } |  
{ { = | < | <= | > | >= } int } |  
IN ( int, ...)
```

- *col_name | expr | position*: 정렬 기준이 되는 컬럼 이름, 표현식 또는 별칭을 지정한다. 하나 이상의 값을 지정할 수 있으며 각 항목은 쉼표로 구분한다. **SELECT** 컬럼 리스트에 명시되지 않은 컬럼도 지정할 수 있다.
- [**ASC** | **DESC**]: **ASC**은 오름차순, **DESC**은 내림차순으로 정렬하며, 정렬 옵션이 명시되지 않으면 오름차순으로 정렬한다.

예제

```
--selecting rows sorted by ORDER BY clause  
SELECT * FROM sales_tbl ORDER BY dept no DESC, name ASC;  
;xr  
  
=== <Result of SELECT Command in Line 1> ===  
  
      dept_no  name                sales_month  sales_amount  
=====
```

501	'Chang'	5	150
501	'Stephan'	4	100
501	'Sue'	6	150
301	'Max'	4	300
201	'George'	1	450
201	'Laura'	2	500
NULL	'Yoka'	4	NULL

```
7 rows selected.  
  
--sorting reversely and limiting result rows by LIMIT clause  
SELECT dept no AS a1, avg(sales amount) AS a2 FROM sales_tbl  
GROUP BY a1 ORDER BY a2 DESC LIMIT 0,3;  
;xr  
  
=== <Result of SELECT Command in Line 1> ===  
  
      a1      a2  
=====
```

201	475
301	300
501	133

```
3 rows selected.  
  
--sorting reversely and limiting result rows by FOR clause
```

```
SELECT dept no AS a1, avg(sales amount) AS a2 FROM sales tbl
GROUP BY a1 ORDER BY a2 DESC FOR ORDERBY NUM() BETWEEN 1 AND 3;
;xr
```

=== <Result of SELECT Command in Line 1> ===

a1	a2
201	475
301	300
501	133

3 rows selected.

LIMIT 절

설명

LIMIT 절은 출력되는 레코드의 개수를 제한할 때 사용한다. 결과 셋의 특정 행부터 마지막 행까지 출력하기 위해 *row_count*에 매우 큰 정수를 지정할 수 있다. **LIMIT** 절은 prepared statement으로 사용할 수 있으며, 인자 대신에 바인드 파라미터(?)를 사용할 수 있다.

LIMIT 절을 포함하는 질의에서는 **WHERE** 절에 **INST_NUM()**, **ROWNUM**을 포함할 수 없으며, **FOR ORDERBY_NUM()**, **HAVING GROUPBY_NUM()**과 함께 사용할 수 없다.

구문

```
LIMIT [offset,] row_count
```

- *offset*: 출력할 레코드의 시작 행 오프셋 값을 지정한다. 결과 셋의 시작 행 오프셋 값은 0이다. 생략할 수 있으며, 디폴트 값은 0이다.
- *row_count*: 출력하고자 하는 레코드 개수를 명시한다. 0보다 큰 정수를 지정할 수 있다.

예제

```
--LIMIT clause can be used in prepared statement
PREPARE STMT FROM 'SELECT * FROM sales tbl LIMIT ?, ?';
EXECUTE STMT USING 0, 10;
```

```
--selecting rows with LIMIT clause
SELECT * FROM sales tbl WHERE sales amount > 100 LIMIT 5;
;xr
```

=== <Result of SELECT Command in Line 1> ===

dept no	name	sales month	sales amount
201	'George'	1	450
201	'Laura'	2	500
301	'Max'	4	300
501	'Chang'	5	150
501	'Sue'	6	150

5 rows selected.

```
--LIMIT clause can be used in subquery
SELECT t1.* FROM
(SELECT * FROM sales tbl AS t2 WHERE sales amount > 100 LIMIT 5) AS t1 LIMIT 1,3;
;xr
```

=== <Result of SELECT Command in Line 1> ===

dept no	name	sales month	sales amount
201	'Laura'	2	500
301	'Max'	4	300
501	'Chang'	5	150

3 rows selected.

USING INDEX 절

설명

USING INDEX 절은 질의에서 인덱스를 지정할 수 있도록 해서 질의처리가 적절한 인덱스를 선택할 수 있게 한다.

USING INDEX 절은 **SELECT**, **DELETE**, **UPDATE** 문의 **WHERE** 절 다음에 지정되어야 한다.

구문

```
SELECT . . . FROM . . . WHERE . . .
[USING INDEX { NONE | index_spec [ {, index_spec } ... ] } ] [ ; ]
DELETE FROM . . . WHERE . . .
[USING INDEX { NONE | index_spec [ {, index_spec } ... ] } ] [ ; ]
UPDATE . . . SET . . . WHERE . . .
[USING INDEX { NONE | index_spec [ {, index_spec } ... ] } ] [ ; ]
index_spec :
[table_name.]index_name [(+)]
```

- **NONE** : **NONE**을 지정한 경우 순차 스캔이 지정된다.
- (+) : 인덱스 이름 뒤에 (+)가 지정된다면 지정된 인덱스를 사용하는 인덱스 스캔이 지정된다.

USING INDEX 절은 강제로 순차 스캔 또는 인덱스 스캔이 사용되게 하거나, 나쁜 성능을 발생시키지 않는 인덱스가 포함되도록 한다.

만약 **USING INDEX** 절에 인덱스 이름의 리스트가 지정된다면 질의 최적화기는 지정된 인덱스만을 대상으로 질의 실행 비용을 계산하고, 나열된 인덱스의 인덱스 스캔 비용과 순차 스캔 비용을 비교하여 최적의 실행 계획을 만든다(CUBRID는 실행 계획을 선택하는데 비용 기반의 질의 최적화를 수행한다).

USING INDEX는 원하는 **ORDER BY** 없이 원하는 순서로 결과를 얻고자 할 때 유용하게 사용될 수 있다. CUBRID는 인덱스 스캔을 하면 인덱스에 저장된 순서로 결과가 생성되는데, 한 테이블에 여러 인덱스가 있을 경우 특정 인덱스의 순서로 질의 결과를 얻고자 할 때 **USING INDEX**를 사용할 수 있다.

예제

다음은 **athlete** 테이블의 생성문에 맞추어 인덱스를 생성한 예제이다.

```
CREATE TABLE athlete (
  code          SMALLINT      NOT NULL PRIMARY KEY,
  name          VARCHAR(40) NOT NULL,
  gender        CHAR(1)       ,
  nation_code   CHAR(3)       ,
  event         VARCHAR(30)   ,
);
CREATE UNIQUE INDEX athlete_idx ON athlete(code, nation code);
CREATE INDEX char_idx ON athlete(gender, nation_code);
```

다음 질의에 대해서 질의 최적화기는 **athlete_idx** 인덱스를 사용하는 인덱스 스캔을 선택할 수 있다.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA';
```

다음 질의에서와 같이 **USING INDEX char_idx**를 지정하면 질의 최적화기는 **USING INDEX**에서 지정한 인덱스만을 대상으로 인덱스 스캔 비용을 산출한다.

인덱스 스캔 비용이 순차 스캔 비용보다 작을 경우 인덱스 스캔을 하게 된다.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA'
USING INDEX char_idx;
```

만약 **char_idx** 인덱스를 사용하는 인덱스 스캔을 강제로 지정할 경우에는 다음과 같이 인덱스 이름 뒤에 (+)를 지정한다.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA'
USING INDEX char_idx(+);
```

만약 순차 스캔을 선택하도록 할 경우 다음과 같이 **USING INDEX** 절에서 **NONE**을 지정한다.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA'
USING INDEX NONE;
```

만약 다음과 같이 **USING INDEX** 절에서 두 개 이상의 인덱스를 지정한 경우 질의 최적화기는 지정된 인덱스 중 적절한 하나를 선택한다.

```
SELECT * FROM athlete WHERE gender='M' AND nation_code='USA'
USING INDEX char_idx, athlete_idx;
```

조인 질의

설명

조인은 두 개 이상의 테이블 또는 뷰(view)에 대해 행(row)을 결합하는 질의이다. 조인 질의에서 두 개 이상의 테이블에 공통인 컬럼을 비교하는 조건을 조인 조건이라고 하며, 조인된 각 테이블로부터 행을 가져와 지정된 조인 조건을 만족하는 경우에만 결과 행을 결합한다.

조인 질의에서 동등 연산자(=)를 이용한 조인 조건을 포함하는 조인 질의를 동등 조인(equi-join)이라 하고, 조인 조건이 없는 조인 질의를 카티션 곱(cartesian products)이라 한다. 또한, 하나의 테이블을 조인하는 경우를 자체 조인(self join)이라 하는데, 자체 조인에서는 **FROM** 절에 같은 테이블이 두 번 사용되므로 테이블 별칭(alias)을 사용하여 컬럼을 구분한다.

한편, 조인된 테이블에 대해 조인 조건을 만족하는 행만 결과를 출력하는 경우를 내부 조인(inner join) 또는 간단 조인(simple join)이라고 하며, 조인된 테이블에 대해 조인 조건을 만족하는 행은 물론, 조인 조건을 만족하지 못하는 행도 포함하여 출력하는 경우를 외부 조인(outer join)이라 한다. 외부 조인은 왼쪽 테이블의 모든 행이 결과로 출력되는 왼쪽 외부 조인과(left outer join)과 오른쪽 테이블의 모든 행이 결과로 출력되는 오른쪽 외부 조인(right outer join)이 있으며, 양쪽의 행이 모두 출력되는 완전 외부 조인(full outer join)이 있다. 이때, 외부 조인 질의 결과에서 한쪽 테이블에 대해 대응되는 컬럼 값이 없는 경우, 이는 모두 **NULL**로 반환된다.

구문

```
FROM table_specification [{, table_specification | join_table_specification}...]

table_specification :
table_specification [ correlation ]
CLASS table name [ correlation ]
subquery correlation
TABLE (expression) correlation

join_table_specification :
[ INNER | { LEFT | RIGHT } [ OUTER ] ] JOIN table_specification
join_condition

join_condition :
ON search_condition
```

- *oin_table_specification*
 - { LEFT | RIGHT } [OUTER] JOIN : LEFT는 왼쪽 외부 조인을 수행하는 질의를 만드는데 사용되고, RIGHT는 오른쪽 외부 조인을 수행하는 질의를 만드는데 사용된다.

CUBRID는 왼쪽 외부 조인과 오른쪽 외부 조인만 지원하며, 완전 외부 조인(full outer join)을 지원하지 않는다. 또한, 외부 조인에서 조인 조건에 부질의와 하위 컬럼을 포함하는 경로 표현식을 사용할 수 없다.

외부 조인의 경우 조인 조건은 내부 조인의 경우와는 다른 방법으로 지정된다. 내부 조인의 조인 조건은 **WHERE** 절에 표현되지만, 외부 조인의 경우에는 조인 조건이 **FROM** 절에서 **ON** 키워드 뒤에 나타난다. 다른 검색 조건은 **WHERE** 절이나 **ON** 절에서 사용할 수 있지만 검색 조건이 **WHERE** 절에 있을 때와 **ON** 절에 있을 때 질의 결과가 달라질 수 있다.

FROM 절에 명시된 순서대로 테이블 실행 순서가 고정되므로, 외부 조인을 사용하는 경우 테이블 순서에 주의하여 질의문을 작성한다. 외부 조인 연산자 '(+)'를 **WHERE** 절에 명시하여 Oracle 스타일의 조인 질의문도 작성 가능하나, 실행 결과나 실행 계획이 원하지 않는 방향으로 유도될 수 있으므로 { LEFT | RIGHT } [OUTER] JOIN 을 이용한 표준 구문을 사용할 것을 권장한다.

예제 1

다음은 내부 조인을 이용하여 1950년 이후에 열린 올림픽 중에서 신기록이 세워진 올림픽의 개최년도와 개최국가를 조회하는 예제이다. 다음 질의는 history 테이블의 host_year가 1950보다 큰 범위에서 값이 존재하는 레코드를 가져온다.

```
SELECT DISTINCT h.host_year, o.host_nation FROM history h, olympic o
WHERE h.host_year=o.host_year AND o.host_year>1950;
=== <Result of SELECT Command in Line 2> ===
      host_year  host_nation
=====
          1968    'Mexico'
          1980    'U.S.S.R.'
          1984    'United States of America'
          1988    'Korea'
          1992    'Spain'
          1996    'United States of America'
          2000    'Australia'
          2004    'Greece'
8 rows selected.
```

예제 2

다음은 외부 조인을 이용하여 1950년 이후에 열린 올림픽에서 신기록이 세워진 올림픽의 개최국가와 개최년도를 조회하되, 신기록이 세워지지 않은 올림픽에 대한 정보도 포함하는 예제이다. 이 예제는 오른쪽 외부 조인이므로, olympic 테이블의 host_nation의 모든 레코드를 포함하고, 값이 존재하지 않는 history 테이블의 host_year에 대해서는 컬럼 값으로 NULL을 반환한다.

```
SELECT DISTINCT h.host year, o.host nation
FROM history h RIGHT OUTER JOIN olympic o ON h.host year=o.host year WHERE
o.host_year>1950;
=== <Result of SELECT Command in Line 3> ===
      host_year  host_nation
=====
          NULL  'Australia'
          NULL  'Canada'
          NULL  'Finland'
          NULL  'Germany'
          NULL  'Italy'
          NULL  'Japan'
          1968  'Mexico'
          1980  'U.S.S.R.'
          1984  'United States of America'
          1988  'Korea'
          1992  'Spain'
          1996  'United States of America'
          2000  'Australia'
          2004  'Greece'
14 rows selected.
```

예제 3

다음은 왼쪽 외부 조인을 이용하여 예제 2와 동일한 결과를 출력하는 예제이다. **FROM** 절에서 두 테이블의 순서를 바꾸어 명시한 후, 왼쪽 외부 조인을 수행한다.

```
SELECT DISTINCT h.host year, o.host nation
FROM olympic o LEFT OUTER JOIN history h ON h.host_year=o.host_year WHERE o.host_year>1950;
=== <Result of SELECT Command in Line 3> ===
      host_year  host_nation
=====
          NULL  'Australia'
          NULL  'Canada'
          NULL  'Finland'
          NULL  'Germany'
          NULL  'Italy'
          NULL  'Japan'
          1968  'Mexico'
          1980  'U.S.S.R.'
          1984  'United States of America'
          1988  'Korea'
          1992  'Spain'
          1996  'United States of America'
          2000  'Australia'
          2004  'Greece'
14 rows selected.
```

이 예에서 h.host_year=o.host_year는 외부 조인 조건이고 o.host_year > 1950은 검색 조건이다. 만약 검색 조건이 **WHERE** 절이 아닌 **ON** 절에서 조인 조건으로 사용될 경우 질의의 의미와 결과는 달라진다. 다음 질의는 o.host_year가 1950보다 크지 않은 값도 질의 결과에 포함된다.

```
SELECT DISTINCT h.host_year, o.host_nation
FROM olympic o LEFT OUTER JOIN history h ON h.host_year=o.host_year AND
o.host_year>1950;
=== <Result of SELECT Command in Line 3> ===
      host_year  host_nation
```



```
=====
      NULL  'Australia'
      NULL  'Belgium'
      NULL  'Canada'
...
      1996  'United States of America'
      2000  'Australia'
      2004  'Greece'
22 rows selected.
```

예제 4

다음은 **WHERE** 절에서 (+)를 사용해서 외부 조인 질의를 작성한 예이며, 예제 2, 예제 3과 같은 결과를 출력한다. 단, (+) 연산자를 이용한 Oracle 스타일의 외부 조인 질의문은 ISO/ANSI 표준이 아니며 모호한 상황을 만들어 낼 수 있으므로 가능하면 표준 구문인 **LEFT OUTER JOIN**(또는 **RIGHT OUTER JOIN**)을 사용할 것을 권장한다.

```
SELECT DISTINCT h.host_year, o.host_nation FROM history h, olympic o
WHERE o.host_year=h.host_year(+) AND o.host_year>1950;
=== <Result of SELECT Command in Line 2> ===
      host_year  host_nation
=====
      NULL      'Australia'
      NULL      'Canada'
      NULL      'Finland'
      NULL      'Germany'
      NULL      'Italy'
      NULL      'Japan'
      1968      'Mexico'
      1980      'U.S.S.R.'
      1984      'United States of America'
      1988      'Korea'
      1992      'Spain'
      1996      'United States of America'
      2000      'Australia'
      2004      'Greece'
14 rows selected.
```

부질의

부질의는 질의 내에서 **SELECT** 절이나 **WHERE** 절 등 표현식이 가능한 모든 곳에서 사용할 수 있다.

부질의가 표현식으로 사용될 경우에는 반드시 단일 컬럼을 반환해야 하지만, 표현식이 아닌 경우에는 하나 이상의 행이 반환될 수 있다. 부질의가 사용되는 경우에 따라 단일 행 부질의와 다중 행 부질의로 나눌 수 있다.

단일 행 부질의

설명

단일 행 부질의는 하나의 컬럼을 갖는 하나의 행을 만든다. 부질의에 의해 행이 반환되지 않을 경우에 부질의 표현식은 **NULL**을 가진다. 만약 부질의가 두 개 이상의 행을 반환하도록 만들어진 경우에는 에러가 발생된다.

예제

다음은 역대 기록 테이블을 조회하는데, 신기록을 수립한 올림픽이 개최된 국가도 함께 조회하는 예제이다. 이 예제는 표현식으로 사용된 단일 행 부질의 결과를 보여준다. 이 예에서 부질은 **olympic** 테이블에서 **host_year** 컬럼 값이 **history** 테이블의 **host_year** 컬럼 값과 같은 행에 대해 **host_nation** 값을 반환한다. 조건에 일치되는 값이 없을 경우 부질의 결과는 **NULL**이 표시된다.

```
SELECT h.host_year, (SELECT host_nation FROM olympic o WHERE o.host_year=h.host_year),
h.event_code, h.score, h.unit from history h;
=== <Result of SELECT Command in Line 1> ===
      host_year (SELECT host_nation FROM olympic o WHERE
o.host_year=h.host_year)      event_code      score      unit
=====
==
      2004      'Greece'      20283      '07:53.0'      'time'
      2004      'Greece'      20283      '07:53.0'      'time'
      2004      'Greece'      20281      '03:57.0'      'time'
      2004      'Greece'      20281      '03:57.0'      'time'
      2004      'Greece'      20281      '03:57.0'      'time'
      2004      'Greece'      20281      '03:57.0'      'time'
      2004      'Greece'      20326      '210'      'kg'
      2000      'Australia'      20328      '225'      'kg'
      2004      'Greece'      20331      '237.5'      'kg'
...
147 rows selected.
```

다중 행 부질의

설명

다중 행 부질의는 지정된 컬럼을 갖는 하나 이상의 행을 반환한다. 다중 행 부질의 결과는 적절한 키워드(**SET**, **MULTISET**, **LIST**, **SEQUENCE**)를 사용하여 집합, 다중집합, 순차집합을 만드는데 사용될 수 있다.

예제

다음은 국가 테이블에서 국가 이름과 수도 이름을 조회하되, 올림픽을 개최한 국가는 개최도시를 리스트로 묶어 함께 조회하는 예제이다. 이 예제 같은 경우는 부질의 결과를 이용하여 **olympic** 테이블의 **host_city** 컬럼 값으로 리스트를 만든다. 이 질의는 **nation** 테이블에 대해 **name**, **capital** 값과 **host_nation** 값을 포함하는 **olympic** 테이블의 **host_city** 값에 대한 집합을 반환한다. 질의 결과에서 **name** 값이 공집합인 경우는 제외되고, **name**과 같은 값을 갖는 **olympic** 테이블이 존재하지 않는 경우에는 공집합이 반환된다.

```
SELECT name, capital, list(SELECT host_city FROM olympic WHERE host_nation = name) FROM
nation;
=== <Result of SELECT Command in Line 1> ===
      name      capital      sequence((SELECT host_city FROM olympic
WHERE host_nation=name))
=====
'Somalia'      'Mogadishu'      {}
'Sri Lanka'      'Sri Jayewardenepura Kotte'      {}
'Sao Tome & Principe'      'Sao Tome'      {}
...
'U.S.S.R.'      'Moscow'      {'Moscow'}
'Uruguay'      'Montevideo'      {}
'United States of America'      'Washington.D.C'      {'Atlanta ', 'St. Louis', 'Los
Angeles', 'Los Angeles'}
'Uzbekistan'      'Tashkent'      {}
'Vanuatu'      'Port Vila'      {}
```

```
215 rows selected.
```

이런 형태의 다중 행 부질의 표현식은 집합 값을 갖는 표현식이 허용되는 모든 곳에서 사용할 수 있다.

단, 클래스 속성 정의에서 **DEFAULT** 명세 부분과 같이 집합 상수 값이 요구되는 곳에는 사용될 수 없다.

부질의 내에서 **ORDER BY** 절을 명시적으로 사용하지 않는 경우 다중 행 부질의 결과의 순서는 지정되지

않으므로, 순차 집합을 생성하는 다중 행 부질의는 **ORDER BY** 절을 사용하여 결과의 순서를 지정해야 한다.

계층적 질의

START WITH ... CONNECT BY 절

설명

계층적 질의란 테이블에 포함된 행(row)간에 수직적 계층 관계가 성립되는 데이터에 대하여 계층 관계에 따라 각 행을 출력하는 질의이다. **START WITH ... CONNECT BY** 절은 **SELECT** 구문과 결합하여 사용된다.

구문

```
SELECT column_list
FROM table_joins | tables
[WHERE join_conditions and/or filtering_conditions]
[START WITH condition]
CONNECT BY [NOCYCLE] condition
```

START WITH 절

START WITH 절은 계층 관계가 시작되는 루트 행(root row)을 지정하기 위한 것으로, **START WITH** 절

다음에 계층 관계를 검색하기 위한 조건식을 포함한다. 만약, **START WITH** 절에 다음에 위치하는 조건식이 생략되면 대상 테이블 내에 존재하는 모든 행을 루트 행으로 간주하여 계층 관계를 검색할 것이다.

참고 **START WITH** 절이 생략되거나, **START WITH** 조건식을 만족하는 결과 행이 존재하지 않는 경우, 테이블 내의 모든 행을 루트 행으로 간주하여 각 루트 행에 속하는 하위 자식 행들 간 계층 관계를 검색하므로 결과 행들 중 일부는 중복되어 출력될 수 있다.

CONNECT BY [NOCYCLE] PRIOR 절

- **PRIOR** : **CONNECT BY** 조건식은 한 쌍의 행에 대한 상-하 계층 관계(부모-자식 관계)를 정의하기 위한 것으로, 조건식 내에서 하나는 부모(parent)로 지정되고, 다른 하나는 자식(child)으로 지정된다. 이처럼 행 간의 부모-자식 간 계층 관계를 정의하기 위하여 **CONNECT BY** 조건식 내에 **PRIOR** 연산자를 이용하여 부모 행의 컬럼 값을 지정한다. 즉, 부모 행의 컬럼 값과 같은 컬럼 값을 가지는 모든 행은 자식 행이 된다.
- **NOCYCLE** : **CONNECT BY** 절의 조건식에 따른 계층 질의 결과는 루프를 포함할 수 있으며, 이것은 계층 트리를 생성할 때 무한 루프를 발생시키는 원인이 될 수 있다. 따라서, CUBRID는 루프를 발견하면 기본적으로 오류를 반환하고, 특수 연산자인 **NOCYCLE**이 **CONNECT BY** 절에 명시된 경우에는 오류를 발생시키지 않고 해당 루프에 의해 검색된 결과를 출력한다.
만약, **CONNECT BY** 절에서 **NOCYCLE**이 명시되지 않은 계층 질의문을 수행 중에 루프가 감지되는 경우,

CUBRID는 오류를 반환하고 해당 질의문을 취소한다. 반면, **NOCYCLE**이 명시된 계층 질의문에서 루프가 감지되는 경우, CUBRID는 오류를 반환하지는 않지만 루프가 감지된 행에 대해 **CONNECT_BY_ISCYCLE** 값을 1로 설정하고, 더 이상 계층 트리의 검색을 확장하지 않을 것이다.

예제

아래 예제를 참조하여 계층 질의문을 작성할 수 있다. 예제를 실행하기 위해 필요한 데이터베이스 스키마는 다음과 같다.

tree 테이블

ID	MgrID	Name
1	NULL	Kim
2	NULL	Moy
3	1	Jonas
4	1	Smith
5	2	Verma
6	2	Foster
7	6	Brown

tree_cycle 테이블

ID	MgrID	Name
1	NULL	Kim
2	11	Moy
3	1	Jonas
4	1	Smith
5	3	Verma
6	3	Foster
7	4	Brown
8	4	Lin
9	2	Edwin
10	9	Audrey
11	10	Stone

```
-- tree 테이블을 만들고 데이터를 삽입하기
CREATE TABLE tree(ID INT, MgrID INT, Name VARCHAR(32));

INSERT INTO tree VALUES (1,NULL,'Kim');
INSERT INTO tree VALUES (2,NULL,'Moy');
```

```

INSERT INTO tree VALUES (3,1,'Jonas');
INSERT INTO tree VALUES (4,1,'Smith');
INSERT INTO tree VALUES (5,2,'Verma');
INSERT INTO tree VALUES (6,2,'Foster');
INSERT INTO tree VALUES (7,6,'Brown');

-- tree_cycle 테이블을 만들고 데이터를 삽입하기
CREATE TABLE tree_cycle(ID INT, MgrID INT, Name VARCHAR(32));

INSERT INTO tree_cycle VALUES (1,NULL,'Kim');
INSERT INTO tree_cycle VALUES (2,11,'Moy');
INSERT INTO tree_cycle VALUES (3,1,'Jonas');
INSERT INTO tree_cycle VALUES (4,1,'Smith');
INSERT INTO tree_cycle VALUES (5,3,'Verma');
INSERT INTO tree_cycle VALUES (6,3,'Foster');
INSERT INTO tree_cycle VALUES (7,4,'Brown');
INSERT INTO tree_cycle VALUES (8,4,'Lin');
INSERT INTO tree_cycle VALUES (9,2,'Edwin');
INSERT INTO tree_cycle VALUES (10,9,'Audrey');
INSERT INTO tree_cycle VALUES (11,10,'Stone');

-- CONNECT BY 절을 이용하여 계층 질의문 수행하기
SELECT *
  FROM tree
 CONNECT BY PRIOR id=mgrid
 ORDER BY id;

id  mgrid      name
=====
1    null      Kim
2    null      Moy
3    1          Jonas
3    1          Jonas
4    1          Smith
4    1          Smith
5    2          Verma
5    2          Verma
6    2          Foster
6    2          Foster
7    6          Brown
7    6          Brown
7    6          Brown

-- START WITH 절을 이용하여 계층 질의문 수행하기
SELECT *
  FROM tree
 START WITH mgrid IS NULL
 CONNECT BY prior id=mgrid
 ORDER BY id;

id  mgrid      name
=====
1    null      Kim
2    null      Moy
3    1          Jonas
4    1          Smith
5    2          Verma
6    2          Foster
7    6          Brown

```

조인 테이블에 대한 계층 질의

테이블 조인 조건식

SELECT 문에서 대상 테이블이 조인된 경우, **WHERE** 절에는 검색 조건식 외에 테이블 조인 조건을 포함할 수 있다. 이때, CUBRID는 제일 먼저 **WHERE** 절의 조인 조건을 적용하여 테이블 조인 연산을 수행한 후,

CONNECT BY 절의 조건식을 적용하고, 마지막으로 **WHERE** 절 내의 나머지 검색 조건식을 적용하여 연산을 처리한다.

WHERE 절 내에 조인 조건식과 검색 조건식을 함께 명시하는 경우, 내부적으로 조인 조건식이 검색 조건식으로 분류되어 의도하지 않게 연산 순서가 달라질 수 있으므로, **WHERE** 절보다는 **FROM** 절 내에 테이블 조인 조건을 명시하는 것을 권장한다.

계층 질의 결과

조인 테이블에 대한 계층 질의 결과는 **START WITH** 절의 조건식에 따라 루트 행으로부터 출력된다.

만약 **START WITH** 절이 생략되면 조인된 테이블의 모든 행들을 루트 행으로 간주하여 계층 관계를 출력한다. 이를 위해 CUBRID는 하나의 루트 행에 대하여 모든 자식 행을 검색한 후, 각 자식 행 하위에 속하는 모든 자식 행을 재귀적으로 검색한다. 이러한 검색은 더 이상의 자식 행이 발견되지 않을 때까지 반복된다.

또한, 계층 질의문은 **CONNECT BY** 절의 조건식을 먼저 적용하여 결과 행들을 검색한 후, **WHERE** 절에 명시된 검색 조건식을 적용하여 최종 결과 행들을 출력한다.

예제

다음은 두 개의 조인된 테이블에 대하여 계층 질의문을 수행하는 예제이다.

```
-- tree2 테이블을 생성하고 데이터를 삽입하기
CREATE TABLE tree2(id int, treeid int, job varchar(32));

INSERT INTO tree2 VALUES(1,1,'Partner');
INSERT INTO tree2 VALUES(2,2,'Partner');
INSERT INTO tree2 VALUES(3,3,'Developer');
INSERT INTO tree2 VALUES(4,4,'Developer');
INSERT INTO tree2 VALUES(5,5,'Sales Exec. ');
INSERT INTO tree2 VALUES(6,6,'Sales Exec. ');
INSERT INTO tree2 VALUES(7,7,'Assistant');
INSERT INTO tree2 VALUES(8,null,'Secretary');

-- 조인 테이블에 대해 계층 질의문을 수행하기
SELECT t.id,t.name,t2.job,level
FROM tree t
      inner join tree2 t2 on t.id=t2.treeid
START WITH t.mgrid is null
CONNECT BY prior t.id=t.mgrid
ORDER BY t.id;
```

id	name	job	level
1	Kim	Partner	1
2	Moy	Partner	1
3	Jonas	Developer	2
4	Smith	Developer	2
5	Verma	Sales Exec.	2
6	Foster	Sales Exec.	2
7	Brown	Assistant	3

계층 질의문에서 사용 가능한 의사 컬럼

LEVEL

LEVEL은 계층 질의 결과 행의 깊이 레벨(depth)을 나타내는 의사 컬럼(pseudo column)이다. 루트 노드의 **LEVEL**은 1이며, 하위 자식 노드의 **LEVEL**은 2가 된다.

LEVEL 의사 컬럼은 **SELECT** 문 내의 **WHERE** 절, **ORDER BY** 절, **GROUP BY ... HAVING** 절에서 사용 가능하며, 집계 함수를 이용하는 구문에서도 사용 가능하다.

다음은 노드의 레벨을 확인하기 위하여 **LEVEL** 값을 조회하는 예제이다.

```
-- LEVEL의 값을 확인하기
SELECT id, mgrid, name, LEVEL
FROM tree
WHERE LEVEL=2
START WITH mgrid IS NULL
CONNECT BY PRIOR id=mgrid
ORDER BY id;
```

id	mgrid	name	level
3	1	Jonas	2
4	1	Smith	2
5	2	Verma	2
6	2	Foster	2

CONNECT_BY_ISLEAF

CONNECT_BY_ISLEAF는 계층 질의 결과 행이 리프 노드(leaf node : 하위에 자식 노드를 가지지 않는 단말 노드)인지 가리키는 의사 컬럼이다. 계층 구조 하에서 현재 행이 리프 노드이면 1을 반환하고, 그렇지 않으면 0을 반환한다.

다음은 리프 노드를 확인하기 위하여 **CONNECT_BY_ISLEAF** 값을 조회하는 예제이다.

```
-- CONNECT_BY_ISLEAF의 값을 확인하기
SELECT id, mgrid, name, CONNECT_BY_ISLEAF
FROM tree
START WITH mgrid IS NULL
CONNECT BY PRIOR id=mgrid
ORDER BY id;
```

id	mgrid	name	connect_by_isleaf
1	null	Kim	0
2	null	Moy	0
3	1	Jonas	1
4	1	Smith	1
5	2	Verma	1
6	2	Foster	0
7	6	Brown	1

CONNECT_BY_ISCYCLE

CONNECT_BY_ISCYCLE은 계층 질의 결과 행이 루프를 발생시키는 행인지를 가리키는 의사 컬럼이다. 즉, 현재 행의 자식이 동시에 조상이 되어 루프를 발생시키는 경우 1을 반환하고, 그렇지 않으면 0을 반환한다.

CONNECT_BY_ISCYCLE 의사 컬럼은 **SELECT** 문 내의 **WHERE** 절, **ORDER BY** 절, **GROUP BY ... HAVING** 절에서 사용할 수 있으며, 집계 함수를 이용하는 구문에서도 사용 가능하다.

참고 **CONNECT_BY_ISCYCLE**은 **CONNECT BY** 절에 **NOCYCLE** 키워드가 명시되는 경우에만 사용할 수 있다.

다음은 루프를 발생시키는 행을 확인하기 위해 **CONNECT_BY_ISCYCE** 값을 조회하는 예제이다.

```
-- CONNECT_BY_ISCYCLE의 값을 확인하기
SELECT id, mgrid, name, CONNECT BY ISCYCLE
FROM tree_cycle
START WITH name in ('Kim', 'Moy')
CONNECT BY NOCYCLE PRIOR id=mgrid
ORDER BY id;
```

id	mgrid	name	connect by iscycle
1	null	Kim	0
2	11	Moy	0
3	1	Jonas	0
4	1	Smith	0
5	3	Verma	0
6	3	Foster	0
7	4	Brown	0
8	4	Lin	0
9	2	Edwin	0
10	9	Audrey	0
11	10	Stone	1

계층 질의문에서 사용 가능한 연산자

CONNECT_BY_ROOT 연산자

CONNECT_BY_ROOT은 컬럼 값으로 루트 행의 값을 반환한다.

이 연산자는 **SELECT** 문 내의 **WHERE** 절 및 **ORDER BY** 절에서 사용할 수 있다.

다음은 계층 질의 결과 행에 대하여 루트 행의 id 값을 조회하는 예제이다.

```
-- 각 행마다 루트 행의 id 값을 확인하기
SELECT id, mgrid, name, CONNECT_BY_ROOT id
FROM tree
START WITH mgrid IS NULL
CONNECT BY PRIOR id=mgrid
ORDER BY id;
```

id	mgrid	name	connect_by_root id
1	null	Kim	1
2	null	Moy	2
3	1	Jonas	1
4	1	Smith	1
5	2	Verma	2
6	2	Foster	2
7	6	Brown	2

PRIOR 연산자

PRIOR 연산자는 컬럼 값으로 부모 행의 값을 반환하고, 루트 행에 대해서는 **NULL**을 반환한다.

이 연산자는 **SELECT** 문 내의 **WHERE** 절, **ORDER BY** 절 및 **CONNECT BY** 절에서 사용할 수 있다.

다음은 계층 질의 결과 행에 대하여 부모 행의 id 값을 조회하는 예제이다.

```
-- 각 행마다 부모 행의 id 값을 확인하기
SELECT id, mgrid, name, PRIOR id as "prior_id"
FROM tree
```



```

START WITH mgrid IS NULL
CONNECT BY PRIOR id=mgrid
ORDER BY id;

id mgrid      name      prior id
=====
1  null      Kim      null
2  null      Moy      null
3  1         Jonas     1
4  1         Smith     1
5  2         Verma     2
6  2         Foster    2
7  6         Brown     6

```

계층 질의문에서 사용 가능한 함수

설명

SYS_CONNECT_BY_PATH 함수는 루트 행으로부터 해당 행까지의 상-하 관계의 path를 문자열로 반환하는 함수이다. 이때, 함수의 인자로 지정되는 컬럼과 구분자는 문자형 타입이어야 하며, 각 path는 지정된 구분자에 의해 구분되어 연속적으로 출력된다.

이 함수는 **SELECT** 문 내의 **WHERE** 절과 **ORDER BY** 절에서 사용할 수 있다.

구문

```
SYS_CONNECT_BY_PATH (column_name, separator_char)
```

예제

다음은 루트 행으로부터 해당 행의 path를 확인하는 예제이다.

```

-- 구분자를 이용하여 루트 행으로부터 해당 행까지 path를 확인하기
SELECT id, mgrid, name, SYS_CONNECT_BY_PATH(name, '/') as [hierarchy]
FROM tree
START WITH mgrid IS NULL
CONNECT BY PRIOR id=mgrid
ORDER BY id;

id mgrid      name      hierarchy
=====
1  null      Kim      /Kim
2  null      Moy      /Moy
3  1         Jonas     /Kim/Jonas
4  1         Smith     /Kim/Smith
5  2         Verma     /Moy/Verma
6  2         Foster    /Moy/Foster
7  6         Brown     /Moy/Foster/Brown

```

계층 질의문에서의 데이터 정렬

설명

ORDER SIBLINGS BY 절은 계층 질의 결과 값들의 계층 정보를 유지하면서 특정 컬럼을 기준으로 오름차순 또는 내림차순으로 데이터를 정렬하며, 동일한 부모를 가진 자식 행들을 정렬할 수 있다.

계층적 질의문에서 데이터의 계층적 순서를 파악하기 위해 사용한다.

구문

```
ORDER SIBLINGS BY col_1 [ASC|DESC] [, col_2 [ASC|DESC] [...[, col_n [ASC|DESC]]...]]
```

예제 1

R이라는 부모 행에 C1과 C2라는 자식 행들이 있다. C1은 C1_1, C1_2 및 C1_3이라는 자식 행을 가지며, C2는 C2_1, C2_2 및 C2_3의 자식 행을 가진다.

ORDER SIBLINGS BY 절을 명시하지 않고 데이터를 조회할 경우, 계층 질의에서는 아래와 같이 기본적으로 깊이 우선(depth-first) 순서에 따라 결과 데이터를 출력하고, 정렬을 수행하지 않는다.

ROWS	COLUMN VALUES	LEVEL
R(parent node)	A	Level 1
C1	B	Level 2
C1_1	G	Level 3
C1_2	F	Level 3
C1_3	H	Level 3
C2	D	Level 2
C2_1	C	Level 3
C2_2	E	Level 3
C2_3	I	Level 3

ORDER SIBLINGS BY 절을 이용하면 아래와 같이, 명시된 컬럼 리스트에 따라 데이터가 정렬되지만, 계층 관계를 파악하기 어렵다.

ROWS	COLUMN VALUES	LEVEL
R	A	Level 1
C1	B	Level 2
C2_1	C	Level 3
C2	D	Level 2
C2_2	E	Level 3
C1_2	F	Level 3
C1_1	G	Level 3
C1_3	H	Level 3
C2_3	I	Level 3

계층 질의 결과는 기본적으로 **ORDER SIBLINGS BY** 절에 명시된 컬럼 리스트를 따라 정렬된 부모와 그 부모의 자식 노드들이 연속적으로 출력되며, 동일 부모를 가지는 형제는 명시된 정렬 순서에 따라 정렬되어 출력된다.

예제 2

다음은 동일한 부모를 가지는 행들을 정렬시키는 예제이다.

```
-- 동일한 부모를 가지는 자식 노드들을 연속 출력하기
SELECT id, mgrid, name, LEVEL
FROM tree
START WITH mgrid IS NULL
CONNECT BY PRIOR id=mgrid
ORDER SIBLINGS BY id;

id  mgrid      name      level
=====
1    null      Kim       1
3    1         Jonas     2
4    1         Smith     2
2    null      Moy       1
5    2         Verma     2
6    2         Foster    2
7    6         Brown     3
```

계층 질의문 사용 예

SELECT 문에 **CONNECT BY** 절을 명시하여 계층 질의문을 작성하는 예이다.

재귀적 참조 관계를 가지는 테이블을 생성했으며, 이 테이블은 ID와 ParentID라는 두 개의 컬럼으로 구성되고, ID와 ParentID는 각각 기본 키와 외래 키로 정의된다고 가정한다. 이때, 루트 노드의 ParentID 값은 **NULL**이 된다.

테이블이 생성되었다면, 아래와 같이 **UNION ALL**을 이용하여 계층 구조를 가지는 전체 데이터와 **LEVEL** 값을 조회할 수 있다.

```
SELECT L1.ID, L1.ParentID, ..., 1 AS [Level]
FROM tree table AS L1
WHERE L1.ParentID IS NULL
UNION ALL
SELECT L2.ID, L2.ParentID, ..., 2 AS [Level]
FROM tree table AS L1
INNER JOIN tree table AS L2 ON L1.ID=L2.ParentID
WHERE L1.ParentID IS NULL
UNION ALL
SELECT L3.ID, L3.ParentID, ..., 3 AS [Level]
FROM tree_table AS L1
INNER JOIN tree table AS L2 ON L1.ID=L2.ParentID
INNER JOIN tree_table AS L3 ON L2.ID=L3.ParentID
WHERE L1.ParentID IS NULL
UNION ALL ...
```

계층 관계를 가지는 데이터의 레벨이 얼마나 될지 예측할 수 없으므로, 위 질의문은 새로운 행이 검색되지 않을 때까지 루프를 도는 저장 프로시저(STORED PROCEDURE) 문으로 재작성할 수 있다.

그러나 루프를 도는 동안 각 단계마다 계층 트리를 확인해야 하므로, 아래와 같이 **SELECT** 문에 **CONNECT BY** 절을 명시하여 계층 질의문을 재작성할 수 있다. 다음의 질의문을 실행하면, 계층 관계를 가지는 데이터 전체와 각 행의 레벨이 출력된다.

```
SELECT ID, ParentID, ..., Level
FROM tree table
START WITH ParentID IS NULL
CONNECT BY ParentID=PRIOR ID
```

루프로 인한 오류를 발생시키지 않으려면 다음과 같이 **NOCYCLE**을 명시할 수 있다.

```
SELECT ID, ParentID, ..., Level
FROM tree table
START WITH ParentID IS NULL
CONNECT BY NOCYCLE ParentID=PRIOR ID
```

계층 질의문의 성능

CONNECT BY 절을 이용한 계층 질의문이 짧고 간편하지만 질의 처리 속도 측면에서는 한계를 가지고 있으므로 주의해야 한다.

질의문 수행 결과가 대상 테이블의 모든 행을 출력하는 경우라면, **CONNECT BY** 절을 이용한 계층 질의문은 루프 감지, 의사 컬럼의 예약 등의 내부적인 처리로 인해 오히려 일반적인 질의문보다 성능이 낮을 수 있다. 반대로 대상 테이블에 대해 일부 행만 출력하는 경우라면 **CONNECT BY** 절을 이용한 계층 질의문의 성능이 높다.

예를 들어, 2만 개의 레코드를 가지는 테이블에 대하여 약 1000개의 레코드를 포함하는 서브 트리를 검색하는 경우라면, **CONNECT BY** 절을 포함한 **SELECT** 문은 **UNION ALL**을 결합한 **SELECT** 문보다 약 30%의 성능 향상을 기대할 수 있다.

INSERT

개요

설명

INSERT 문을 사용하여 데이터베이스에 존재하는 테이블에 새로운 레코드를 삽입할 수 있다. CUBRID는 **INSERT ... VALUES** 문, **INSERT ... SET** 문, **INSERT ... SELECT** 문을 지원한다.

INSERT ... VALUES 문과 **INSERT ... SET** 문은 명시적으로 지정된 값을 기반으로 새로운 레코드를 삽입하며, **INSERT ... SELECT** 문은 다른 테이블에서 조회한 결과 레코드를 삽입할 수 있다. 단일 **INSERT** 문을 이용하여 여러 행을 삽입하기 위해서는 **INSERT ... VALUES** 문 또는 **INSERT ... SELECT** 문을 사용한다.

구문

```
<INSERT ... VALUES statement>
INSERT [INTO] table name [(column name, ...)]
    {VALUES | VALUE}({expr | DEFAULT}, ...)[, ({expr | DEFAULT}, ...),...]
    [ON DUPLICATE KEY UPDATE column name = expr, ... ]
INSERT [INTO] table_name DEFAULT [ VALUES ]
INSERT [INTO] table name VALUES()

<INSERT ... SET statement>
INSERT [INTO] table name
    SET column name = {expr | DEFAULT}[, column name = {expr | DEFAULT},...]
    [ON DUPLICATE KEY UPDATE column_name = expr, ... ]

<INSERT ... SELECT statement>
```

```
INSERT [INTO] table_name [(column_name, ...)]
SELECT ...
[ON DUPLICATE KEY UPDATE column_name = expr, ... ]
```

- *table_name*: 새로운 레코드를 삽입할 대상 테이블 이름을 지정한다.
- *column_name*: 값을 삽입할 컬럼 이름을 지정한다. 이 값을 생략하면, 테이블에 정의된 모든 컬럼이 명시된 것으로 간주되므로 모든 컬럼에 대한 값을 **VALUES** 뒤에 명시해야 한다. 테이블에 정의된 컬럼 중 일부 컬럼만 명시하면 나머지 컬럼에는 **DEFAULT**로 정의된 값이 할당되며, 정의된 디폴트 값이 없는 경우 **NULL** 값이 할당된다.
- *expr* | **DEFAULT**: **VALUES** 뒤에는 컬럼에 대응하는 컬럼 값을 명시하며, 표현식 또는 **DEFAULT** 키워드를 값으로 지정할 수 있다. 명시된 컬럼 리스트의 순서와 개수는 컬럼 값 리스트와 대응되어야 하며, 하나의 레코드에 대해 컬럼 값 리스트는 괄호로 처리된다.
- **DEFAULT**: 디폴트 값을 컬럼 값으로 명시하기 위하여 **DEFAULT** 키워드를 사용할 수 있다. **VALUES** 키워드 뒤의 컬럼 값 리스트 내에 **DEFAULT**를 명시하면 해당 컬럼에 디폴트 값을 저장하고, **VALUES** 키워드 앞에 **DEFAULT**를 명시하면 테이블 내 모든 컬럼에 대해 디폴트 값을 저장한다. 디폴트 값이 정의되지 않은 컬럼에 대해서는 **NULL**을 저장한다.
- **ON DUPLICATE KEY UPDATE**: **PRIMARY KEY** 또는 **UNIQUE** 속성이 정의된 컬럼에 중복 값이 삽입되어 제약 조건 위반이 발생하면, **ON DUPLICATE KEY UPDATE** 절에 명시된 액션을 수행하면서 제약 조건 위반을 발생시킨 값을 특정 값으로 변경한다.

예제

```
CREATE TABLE a_tbl1(
id INT UNIQUE,
name VARCHAR,
phone VARCHAR DEFAULT '000-0000');

--insert default values with DEFAULT keyword before VALUES
INSERT INTO a_tbl1 DEFAULT VALUES;

--insert multiple rows
INSERT INTO a_tbl1 VALUES (1,'aaa', DEFAULT),(2,'bbb', DEFAULT);

--insert a single row specifying column values for all
INSERT INTO a_tbl1 VALUES (3,'ccc', '333-3333');

--insert two rows specifying column values for only
INSERT INTO a_tbl1(id) VALUES (4), (5);

--insert a single row with SET clauses
INSERT INTO a_tbl1 SET id=6, name='eee';
INSERT INTO a_tbl1 SET id=7, phone='777-7777';

SELECT * FROM a_tbl1;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  name                phone
=====
      NULL NULL                '000-0000'
      1  'aaa'                '000-0000'
      2  'bbb'                '000-0000'
      3  'ccc'                '333-3333'
      4  NULL                '000-0000'
      5  NULL                '000-0000'
      6  'eee'                '000-0000'
      7  NULL                '777-7777'

8 rows selected.
```

INSERT ... SELECT 문

설명

INSERT 문에 **SELECT** 절의를 사용하면 하나 이상의 다른 테이블로부터 특정 검색 조건을 만족하는 데이터를 추출하여 대상 테이블에 삽입할 수 있다.

SELECT 문은 **VALUES** 키워드 대신 사용하거나 **VALUES** 뒤의 컬럼 값 리스트 내에 부질의로서 포함될 수 있다. **VALUES** 키워드를 대신하여 **SELECT** 문을 명시하면, 질의 결과로 얻은 다수의 레코드를 한 번에 대상 테이블 컬럼에 삽입할 수 있다. **SELECT** 문을 컬럼 값 리스트 내에 부질의로 사용하려면 질의 결과 레코드가 하나여야 한다.

구문

```
INSERT [INTO] table_name [(column_name, ...)]
    SELECT...
    [ON DUPLICATE KEY UPDATE column_name = expr, ... ]
```

예제

```
--creating an empty table which schema replicated from a tbl1
CREATE TABLE a_tbl2 LIKE a_tbl1;

--inserting multiple rows from SELECT query results
INSERT INTO a_tbl2 SELECT * FROM a_tbl1 WHERE id IS NOT NULL;

7 rows affected.

--inserting column value with SELECT subquery specified in the value list
INSERT INTO a_tbl2 VALUES(8, SELECT name FROM a_tbl1 WHERE name <'bbb', DEFAULT);

SELECT * FROM a_tbl2;
csql> ;xr

=== <Result of SELECT Command in Line 1> ===

      id  name                phone
=====
      1  'aaa'                '000-0000'
      2  'bbb'                '000-0000'
      3  'ccc'                '333-3333'
      4  NULL                 '000-0000'
      5  NULL                 '000-0000'
      6  'eee'                '000-0000'
      7  NULL                 '777-7777'
      8  'aaa'                '000-0000'

8 rows selected.
```

ON DUPLICATE KEY UPDATE 절

설명

INSERT 문에 **ON DUPLICATE KEY UPDATE** 절을 명시하여 **UNIQUE** 인덱스 또는 **PRIMARY KEY** 제약 조건이 설정된 컬럼에 중복된 값이 삽입되는 상황에서 에러를 출력하지 않고 새로운 값으로 갱신할 수 있다.

ON DUPLICATE KEY UPDATE 절은 **INSERT** 또는 **UPDATE**에 대한 트리거가 활성화된 테이블에 대해서는 사용할 수 없으며, 중첩된 **INSERT** 문에서도 사용할 수 없다.

구문

```
<INSERT ... VALUES statement>
<INSERT ... SET statement>
<INSERT ... SELECT statement>
INSERT ...
[ON DUPLICATE KEY UPDATE column_name = expr, ... ]
```

- column_name = expr*: **ON DUPLICATE KEY UPDATE** 뒤에 컬럼 값을 변경하고자 하는 컬럼 이름을 명시하고, 등호 부호를 이용하여 새로운 컬럼 값을 명시한다.

예제

```
--creating a new table having the same schema as atheltel
CREATE TABLE a_tbl3 LIKE a_tbl1;
INSERT INTO a_tbl3 SELECT * FROM a_tbl1 WHERE id IS NOT NULL and name IS NOT NULL;
SELECT * FROM a_tbl3;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  name                phone
=====
      1  'aaa'                '000-0000'
      2  'bbb'                '000-0000'
      3  'ccc'                '333-3333'
      6  'eee'                '000-0000'

--insert duplicated value violating UNIQUE constraint
INSERT INTO a_tbl3 VALUES(2, 'bbb', '222-2222');
;xr

In the command from line 1,

ERROR: Operation would have caused one or more unique constraint violations.

--insert duplicated value with specifying ON DUPLICATED KEY UPDATE clause
INSERT INTO a_tbl3 VALUES(2, 'bbb', '222-2222')
ON DUPLICATE KEY UPDATE phone = '222-2222';

SELECT * FROM a_tbl3 WHERE id=2;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  name                phone
=====
      2  'bbb'                '222-2222'

1 rows selected.
```

UPDATE

설명

UPDATE 문을 사용하면 대상 테이블에 저장된 레코드의 컬럼 값을 새로운 값으로 업데이트할 수 있다. **SET** 절에는 업데이트할 컬럼 이름과 새로운 값을 명시하며, [WHERE 절](#)에는 업데이트할 레코드를 추출하기 위한 조건을 명시한다. [LIMIT 절](#)에 업데이트할 레코드 수를 명시할 수 있다.

구문

```
UPDATE table_name SET column_name = {expr | DEFAULT} [, column_name = {expr | DEFAULT}]...  
[WHERE search_condition]  
[LIMIT row_count]
```

- *table_name*: 대상 테이블의 이름을 지정한다.
column_name: 업데이트할 컬럼 이름을 지정한다.
- *expr* | **DEFAULT**: 해당 컬럼의 새로운 값을 지정하며, 표현식 또는 **DEFAULT** 키워드를 값으로 지정할 수 있다. 단일 결과 레코드를 반환하는 **SELECT** 질의를 지정할 수도 있다.
- *search_condition*: [WHERE 절](#)에 조건식을 명시하면, 조건식을 만족하는 레코드에 대해서만 컬럼 값을 업데이트할 수 있다.
- *row_count*: [LIMIT 절](#)에 업데이트할 레코드 수를 명시하며, 0보다 큰 정수를 지정할 수 있다.

참고 사항

같은 **UPDATE** 문에서 한 컬럼은 한 번만 갱신 가능하다.

예제

```
--creating a new table having all records copied from atheltel  
CREATE TABLE a_tbl5 AS SELECT * FROM a_tbl1;  
SELECT * FROM a_tbl5 WHERE name IS NULL;  
;xr  
  
=== <Result of SELECT Command in Line 1> ===  


| id   | name | phone      |
|------|------|------------|
| NULL | NULL | '000-0000' |
| 4    | NULL | '000-0000' |
| 5    | NULL | '000-0000' |
| 7    | NULL | '777-7777' |

  
4 rows selected.  
  
UPDATE a_tbl5 SET name='yyy', phone='999-9999' WHERE name IS NULL LIMIT 3;  
SELECT * FROM a_tbl5;  
;xr  
  
=== <Result of SELECT Command in Line 1> ===  


| id   | name  | phone      |
|------|-------|------------|
| NULL | 'yyy' | '999-9999' |
| 1    | 'aaa' | '000-0000' |
| 2    | 'bbb' | '000-0000' |
| 3    | 'ccc' | '333-3333' |
| 4    | 'yyy' | '999-9999' |
| 5    | 'yyy' | '999-9999' |
| 6    | 'eee' | '000-0000' |
| 7    | NULL  | '777-7777' |

  
8 rows selected.
```


REPLACE

설명

REPLACE 문은 [INSERT 문](#)과 유사하지만, **PRIMARY KEY**와 **UNIQUE** 제약 조건이 정의된 컬럼에 중복된 값을 삽입하면 기존 레코드를 삭제한 후 새로운 레코드를 삽입한다(**INSERT** 문은 에러를 출력한다). **REPLACE** 문은 삽입 또는 삭제 후 삽입을 수행하므로, **REPLACE** 문을 사용하기 위해서는 테이블에 대한 **INSERT**와 **DELETE** 권한을 동시에 가지고 있어야 한다.

REPLACE 문은 새로운 레코드에 의한 **PRIMARY KEY** 또는 **UNIQUE** 인덱스 컬럼 값 중복을 판단하므로, **PRIMARY KEY** 또는 **UNIQUE** 인덱스가 정의되지 않은 테이블에 대해서는 **INSERT** 문을 사용하는 것이 성능상 유리하다. **REPLACE** 문은 SQL 표준을 확장한 질의문이며, 다음을 참고하여 사용한다.

- **REPLACE** 문은 부질의를 포함할 수 없다.
- **REPLACE** 문은 **INSERT** 또는 **DELETE** 트리거가 설정된 테이블에 대해서는 사용할 수 없다.
- **SET col_name = col_name + 1**와 같은 할당문은 유효하지 않다. **SET col_name = DEFAULT(col_name) + 1**으로 수정하여 사용할 수 있다. 이때, *col_name* 컬럼에는 **NULL**이 아닌 디폴트 값이 설정되어 있어야 한다.

구문

```
<REPLACE ... VALUES statement>
REPLACE [INTO] table_name [(column_name, ...)]
    {VALUES | VALUE} ({expr | DEFAULT}, ...)[, ({expr | DEFAULT}, ...),...]

<REPLACE ... SET statement>
REPLACE [INTO] table_name
    SET column_name = {expr | DEFAULT}[, column_name = {expr | DEFAULT},...]

<REPLACE ... SELECT statement>
REPLACE [INTO] table_name [(column_name, ...)]
    SELECT...
```

- *table_name*: 새로운 레코드를 삽입할 대상 테이블 이름을 지정한다.
- *column_name*: 값을 삽입할 컬럼 이름을 지정한다. 이 값을 생략하면, 테이블에 정의된 모든 컬럼이 명시된 것으로 간주되므로 모든 컬럼에 대한 값을 **VALUES** 뒤에 명시해야 한다. 테이블에 정의된 컬럼 중 일부 컬럼만 명시하면 나머지 컬럼에는 **DEFAULT**로 정의된 값이 할당되며, 정의된 디폴트 값이 없는 경우 **NULL** 값이 할당된다.
- *expr | DEFAULT*: **VALUES** 뒤에는 컬럼에 대응하는 컬럼 값을 명시하며, 표현식 또는 **DEFAULT** 키워드를 값으로 지정할 수 있다. 명시된 컬럼 리스트의 순서와 개수는 컬럼 값 리스트와 대응되어야 하며, 하나의 레코드에 대해 컬럼 값 리스트는 괄호로 처리된다.

예제

```
--creating a new table having the same schema as atheltel
CREATE TABLE a_tbl4 LIKE a_tbl1;
INSERT INTO a_tbl4 SELECT * FROM a_tbl1 WHERE id IS NOT NULL and name IS NOT NULL;
SELECT * FROM a_tbl4;
;xr

=== <Result of SELECT Command in Line 1> ===

      id      name      phone
```

```
=====
      1 'aaa'          '000-0000'
      2 'bbb'          '000-0000'
      3 'ccc'          '333-3333'
      6 'eee'          '000-0000'

--insert duplicated value violating UNIQUE constraint
REPLACE INTO a_tbl4 VALUES(1, 'aaa', '111-1111'), (2, 'bbb', '222-2222');
REPLACE INTO a_tbl4 SET id=6, name='fff', phone=DEFAULT;

SELECT * FROM a_tbl4;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  name          phone
=====
      3  'ccc'          '333-3333'
      1  'aaa'          '111-1111'
      2  'bbb'          '222-2222'
      6  'fff'          '000-0000'

4 rows selected.
```

DELETE

설명

DELETE 문을 사용하여 테이블 내에 레코드를 삭제할 수 있으며, [WHERE 절](#)과 결합하여 삭제 조건을 명시할 수 있다. 삭제하고자 하는 레코드의 개수를 한정하려면 [LIMIT 절](#)에 삭제하고자 하는 레코드 개수를 지정할 수 있다. [WHERE 절](#)을 만족하는 레코드 개수가 *row_count*를 초과하면 *row_count* 개의 레코드만 삭제된다.

구문

```
DELETE FROM <table specification> [ WHERE <search condition> ] [LIMIT row count]

<table_specification> ::= <table_hierarchy> | ( <table_hierarchy_comma_list > )

<table_hierarchy> ::= [ ONLY ] <table_name> |
                      ALL <table_name> [ EXCEPT <table_specification> ]
```

- *table_name*: 삭제할 데이터가 포함되어 있는 테이블의 이름을 지정한다.
- *search_condition*: [WHERE 절](#)을 이용하여 *search_condition*을 만족하는 데이터만 삭제한다. 생략할 경우 지정된 테이블의 모든 데이터를 삭제한다.
- *row_count*: [LIMIT 절](#)에 삭제할 레코드 수를 명시하며, 0보다 큰 정수를 지정할 수 있다.

예제

```
CREATE TABLE a_tbl(
id INT NOT NULL,
phone VARCHAR(10));
INSERT INTO a_tbl VALUES(1,'111-1111'), (2,'222-2222'), (3, '333-3333'), (4, NULL), (5,
NULL);
;xr

5 rows affected.

DELETE FROM a_tbl WHERE phone IS NULL LIMIT 1;
;xr

1 rows affected.
```

```
--delete one record only from a tbl
SELECT * FROM a_tbl;
;xr

=== <Result of SELECT Command in Line 1> ===

      id  phone
=====
      1  '111-1111'
      2  '222-2222'
      3  '333-3333'
      5  NULL

4 rows selected.

--delete all records from a_tbl
DELETE FROM a_tbl;
;xr

4 rows affected.
```

TRUNCATE

설명

TRUNCATE 문은 명시된 테이블의 모든 레코드들을 삭제한다.

내부적으로 테이블에 정의된 모든 인덱스와 제약 조건을 먼저 삭제한 후 레코드를 삭제하기 때문에,

WHERE 조건이 없는 **DELETE FROM** *table_name* 문을 사용하는 것보다 빠르다. **TRUNCATE** 문을 사용해서 삭제하면 **ON DELETE** 트리거가 활성화되지 않는다.

대상 테이블에 **PRIMARY KEY** 제약 조건이 정의되어 있고, 이 **PRIMARY KEY**를 하나 이상의 **FOREIGN KEY**가 참조하고 있는 경우에는 **FOREIGN KEY ACTION**을 따른다. **FOREIGN KEY**의 **ON DELETE** 액션이 **RESTRICT**나 **NO ACTION**이면 **TRUNCATE** 문은 에러를 반환하고, **CASCADE**이면 **FOREIGN KEY**도 함께 삭제한다. **TRUNCATE** 문은 해당 테이블의 **AUTO INCREMENT** 컬럼을 초기화하여, 다시 데이터가 입력되면 **AUTO INCREMENT** 컬럼의 초기값부터 생성된다.

구문

```
TRUNCATE [ TABLE ] <table_name>
```

- *table_name*: 삭제할 데이터가 포함되어 있는 테이블의 이름을 지정한다.

예제

```
CREATE TABLE a_tbl(A INT AUTO INCREMENT(3,10) PRIMARY KEY);
INSERT INTO a_tbl VALUES (NULL), (NULL), (NULL);
SELECT * FROM a_tbl;

=====
      3
     13
     23

--AUTO INCREMENT column value increases from the initial value after truncating the table
TRUNCATE TABLE a_tbl;
INSERT INTO a_tbl VALUES (NULL);
SELECT * FROM a_tbl;
```

DO

설명

DO 문은 지정된 연산식을 실행하지만 결과 값을 리턴하지 않는다. **DO** 문은 데이터베이스 서버에서 연산 결과 또는 에러를 반환하지 않기 때문에, 일반적으로 **SELECT** 문보다 수행 속도가 빠르다.

구문

```
DO expression
```

- expression*: 임의의 연산식을 지정한다.

PREPARED STATEMENT

개요

CUBRID 2008 R3.0 버전부터는 서버 측에서 prepared statement 기능을 제공하므로, SQL 레벨에서 직접 이를 수행할 수 있다. SQL 레벨의 prepared statement의 영역은 해당 SQL이 생성된 클라이언트 세션에 한정되며, **;run** 또는 **;xrun** 명령어로 해당 SQL 문을 실행한 후에는 이전의 prepared statement를 실행할 수 없다.

prepared statement 사용을 위해 다음의 SQL 문을 제공한다.

- 실행하고자 하는 SQL 문을 준비한다.

```
PREPARE stmt_name FROM preparable_stmt
```

- prepared statement를 실행한다.

```
EXECUTE stmt_name [USING value [, value] ...]
```

- prepared statement를 해제한다.

```
{DEALLOCATE | DROP} PREPARE stmt_name
```

PREPARE 문

설명

PREPARE 문은 **FROM** 절의 *preparable_stmt*에 지정된 질의문을 준비하고, 이후에 해당 SQL 문을 참조할 때 사용될 이름을 *stmt_name*에 할당한다.

구문

```
PREPARE stmt_name FROM preparable_stmt
```

- stmt_name*: prepared statement의 이름을 할당한다. 해당 클라이언트 세션에 이미 동일한 *stmt_name*을 가지는 SQL 문이 존재하면, 기존 prepared statement을 해제한 후 새로운 SQL 문을 준비한다. 주어진 SQL 문의 오류로 인해 **PREPARE** 문이 정상 수행되지 않는 경우, 해당 SQL 문에 할당된 *stmt_name*은 존재하지 않는 것으로 처리된다.

- *preparable_stmt*: 반드시 단일 SQL 문이어야 하며, 여러 개의 SQL 문을 지정할 수 없다.
preparable_stmt 인자에 바인드 파라미터(?)를 사용할 수 있으며, 이를 따옴표로 감싸지 않아야 한다.

예제

```
--prepare and execute a statement without any parameter marker
PREPARE stmt1 FROM 'SELECT CURRENT_TIMESTAMP ()';
EXECUTE stmt1;
;ru

Current transaction has been committed.

=== <Result of SELECT Command in Line 2> ===

      SYS_TIMESTAMP
=====
03:49:28 PM 04/02/2010

--prepare and execute a statement with a parameter marker
PREPARE stmt1 FROM 'SELECT POWER(?,2)*PI()';
EXECUTE stmt1 USING 2;
;ru

=== <Result of SELECT Command in Line 2> ===

      power( ?;0 , 2)* pi()
=====
1.256637061435917e+01
```

EXECUTE 문

설명

EXECUTE 문은 prepared statement을 실행하며, prepared statement에 바인드 파라미터(?)를 포함하면 **USING** 절 뒤에 데이터 값을 바인딩할 수 있다. 단, **USING** 절에서는 속성과 같은 변수(variable)를 지정할 수 없으며, 반드시 리터럴, 입력 파라미터와 같은 값(value)만 지정할 수 있다.

구문

```
EXECUTE stmt_name [USING value [, value] ...]
```

- *stmt_name*: 실행하고자 하는 prepared statement에 부여된 이름을 지정한다. *stmt_name*이 유효하지 않거나 prepared statement가 존재하지 않는 경우 에러가 출력된다.
- *value*: 바인드 파라미터가 prepared statement에 있는 경우 바인딩할 데이터를 입력한다. 바인드 파라미터와 데이터의 개수 및 순서가 대응되어야 한다. 그렇지 않으면 에러가 출력된다.

예제

```
;clear
EXECUTE stmt1
;ru

ERROR: A prepared statement with the name stmt1 does not exist.

--prepare and execute a statement
PREPARE stmt2 FROM 'SELECT MID(?,?,?)';
EXECUTE stmt2 USING '12345abcdeabcde',6,5;
EXECUTE stmt2 USING '12345abcdeabcde',6,10;
;ru

Current transaction has been committed.
```

```

=== <Result of SELECT Command in Line 1> ===
      mid( ?:0 , ?:1 , ?:2 )
=====
      'abcde'

=== <Result of SELECT Command in Line 2> ===
      mid( ?:0 , ?:1 , ?:2 )
=====
      'abcdeabcde'

```

DEALLOCATE PREPARE 문, DROP PREPARE 문

설명

DEALLOCATE PREPARE 문과 **DROP PREPARE** 문은 동일하며, prepared statement를 해제한다.

DEALLOCATE PREPARE 문 또는 **DROP PREPARE** 문을 수행하지 않더라도 클라이언트 세션이 종료되면, 서버에 의해 모든 prepared statement가 자동 해제된다.

구문

```
{DEALLOCATE | DROP} PREPARE stmt_name
```

- *stmt_name* : 해제하고자 하는 prepared statement에 부여된 이름을 지정한다. *stmt_name*이 유효하지 않거나 prepared statement가 존재하지 않으면 에러가 출력된다.

예제

```

DEALLOCATE PREPARE stmt1;
DEALLOCATE PREPARE stmt2;
;xr

2 command(s) successfully processed.

```

트랜잭션과 잠금

개요

이 장에서는 트랜잭션을 커밋하거나 롤백하는 방법뿐만 아니라 동시성과 복구 이슈에 대하여 논의한다.

다중 사용자 환경에서 데이터베이스의 무결성을 보호하고 사용자의 트랜잭션이 항상 정확하고 일관된 데이터를 확보할 수 있도록 보장하기 위해서는 접근과 갱신의 조정이 필수적이다. 충분한 조정이 없다면 데이터는 타당하지 않거나 순서에 어긋나게 갱신될 수 있다.

동일한 데이터에 대한 병렬 연산을 통제하는 방법은 트랜잭션이 진행되는 동안 데이터를 잠그고 트랜잭션이 끝나기 전까지 다른 트랜잭션에서 용납되지 않은 데이터 접근을 허용하지 않는 것이다. 게다가 특정 테이블에 대한 갱신이 있을 때 이것을 커밋하기 전까지 다른 이들이 볼 수 없도록 한다. 만약 갱신을 커밋하지 않기로 결정했다면 마지막으로 갱신을 커밋하거나 롤백한 이후부터 입력된 모든 질의문을 무효화할 수 있다.

여기에서 소개하는 모든 예제는 `csql`로 실행한 것이며, 예제에서 출력 부분은 이탤릭체로 표기하였다.

데이터베이스 트랜잭션

개요

데이터베이스 트랜잭션은 CUBRID 질의문을 일관성(다중 사용자 환경에서 유효한 결과를 만들어내는 것)과 복구(시스템 실패와 같은 어떤 장애에도 커밋된 트랜잭션의 결과를 유지하는 것과, 어떤 고장에도 불구하고 중단된 트랜잭션은 데이터베이스로부터 무효화되는 것을 보장하는 것)의 단위로 그룹화한다. 하나의 트랜잭션은 데이터베이스에 접근하고 갱신하는 하나의 질의문 또는 여러 질의문으로 구성된다.

CUBRID는 많은 사용자가 동시에 데이터베이스에 접근하도록 하고 데이터베이스의 불일치를 방지하기 위하여 사용자 간 접근과 갱신을 관리한다. 예를 들어 데이터가 한 사용자에게 의해 갱신되었을 때 그 트랜잭션에 관련된 데이터의 변화는 갱신이 커밋될 때까지 다른 사용자나 데이터베이스에서 일어나는 다른 트랜잭션에 보이지 않는다. 트랜잭션이 커밋되지 않고 롤백될 수 있기 때문에 이 원칙은 중요하다.

트랜잭션 처리 결과를 확신할 때까지, 데이터베이스에 영구적으로 갱신하는 것을 연기할 수 있다. 또한 트랜잭션 처리 과정에서 응용 프로그램이나 컴퓨터 시스템에서 만족할 수 없는 결과나 실패가 발생하면 데이터베이스의 모든 갱신을 제거(**ROLLBACK**)할 수 있다. 트랜잭션의 끝은 **COMMIT WORK** 또는 **ROLLBACK WORK** 문으로 결정된다. **COMMIT WORK** 문은 데이터베이스의 모든 갱신을 영구적으로 만드는 반면에 **ROLLBACK WORK** 문은 트랜잭션에서 입력된 모든 갱신을 무효화시킨다. 보다 자세한 설명은 각각 [트랜잭션 커밋](#)과 [트랜잭션 롤백](#)을 참조한다.

트랜잭션 커밋

설명

데이터베이스에서 일어난 갱신들은 **COMMIT WORK** 문이 주어지기 전까지 영구히 저장되지 않는다.

"영구히(permanently)" 저장된다는 것은 디스크에 저장이 완료되는 것을 의미한다. 키워드 **WORK**는 생략이 가능하다. 추가로 데이터베이스의 다른 사용자는 변경이 영구히 반영되기 전까지는 변경 사항을 볼 수 없다.

예를 들어 테이블에 새로운 행을 삽입했을 때 데이터베이스 트랜잭션이 커밋되기 전까지 그 행에 접근할 수 있는 것은 그 행을 삽입한 사용자뿐이다(**UNCOMMITTED INSTANCES** 격리 수준을 사용하면 다른 사용자가 일관성이 없는 커밋되지 않은 갱신을 볼 수도 있다).

트랜잭션이 커밋된 후에는 트랜잭션에서 획득한 모든 잠금이 해제된다.

구문

```
[;]COMMIT [ WORK ]
```

구문 앞에 ;을 붙일 경우 세션 명령어로 간주하여 바로 실행되며 생략할 경우 쿼리문의 하나로 간주하여 ;x[run]을 수행하기 전까지 실행을 보류한다.

예제

아래 예제의 데이터베이스 트랜잭션은 3개의 **UPDATE** 문으로 구성되는데, 3개의 stadium의 seats 컬럼 값을 변경한다. 결과를 검토하기 위해 갱신이 일어나기 전에 현재의 값과 이름을 검색한다. 기본적으로 `csql`은 자동으로 `autocommit`으로 작동되므로, 예제에서는 `autocommit` 모드를 `off`로 설정한 후 동작을 시험한다.

```
;autocommit off
AUTOCOMMIT IS OFF
select name, seats
from stadium where code in (30138, 30139, 30140);
;xrun
=== <Result of SELECT Command in Line 1>===
      name                seats
=====
'Athens Olympic Tennis Centre'      3200
'Goudi Olympic Hall'              5000
'Vouliagmeni Olympic Centre'      3400
3 rows selected.
```

3개의 **UPDATE** 문은 각각의 stadium의 현재 seats을 가지고 있도록 한다. 명령이 수행되면 정확하게 입력이 되었는지 확인하기 위해 seats 테이블의 관련된 컬럼을 검색할 수 있다.

```
update stadium
set seats = seats + 1000
where code in (30138, 30139, 30140);
;xrun
3 rows affected.
select name, seats from stadium where code in (30138, 30139, 30140);
;xrun
=== <Result of SELECT Command in Line 1>===
      name                seats
=====
'Athens Olympic Tennis Centre'      4200
'Goudi Olympic Hall'              6000
'Vouliagmeni Olympic Centre'      4400
```



```
3 rows selected.
```

만약 갱신이 제대로 이루어 졌다면 변경을 영구적으로 만들 수 있다. 이때 아래처럼 **COMMIT WORK** 문을 사용한다.

```
;commit work
```

참고 CUBRID에서는 트랜잭션 관리를 위하여 자동 커밋 모드를 설정할 수 있다. 자동 커밋 모드는 모든 SQL 문을 자동으로 커밋 또는 롤백하는 모드로서, 해당 SQL 문이 정상 수행되면 해당 트랜잭션을 자동 커밋하고, 오류가 발생하면 트랜잭션을 롤백한다. 이러한 자동 커밋 모드는 CUBRID JDBC, ODBC, OLEDB 및 CSQL인터프리터에서 지원되며, CUBRID CCI 및 CUBRID PHP에서는 브로커 파라미터 설정을 통해 **SELECT** 문에 대해서만 자동 커밋 모드를 적용할 수 있다. 이에 관해서는 [브로커별 파라미터](#)에서 **SELECT_AUTO_COMMIT**의 설명을 참고한다. CSQL 인터프리터에서 자동 커밋 모드를 설정하는 세션 명령어(**Autocommit**)에 대해서는 [세션 명령어](#)를 참조한다.

트랜잭션 롤백

설명

ROLLBACK WORK 문은 마지막 트랜잭션 이후의 모든 데이터베이스의 갱신을 제거한다. **WORK** 키워드는 생략 가능하다. 이것은 데이터베이스에 영구적으로 입력하기 전에 부정확하고 불필요한 갱신을 무효화할 수 있다. 트랜잭션 동안 획득한 모든 잠금은 해제된다.

구문

```
[;] ROLLBACK [ WORK ]
```

구문 앞에 ;을 붙일 경우 세션 명령어로 간주하여 바로 실행되며 생략할 경우 쿼리문의 하나로 간주하여 ;x[run]을 수행하기 전까지 실행을 보류한다.

예제

다음 예제는 동일한 테이블의 정의와 행을 수정하는 두 개의 명령을 보여주고 있다.

```
alter class code
drop s name;
insert into code (s_name, f_name) values ('D','Diamond');
;xrun

In line 3, column 21,
ERROR: s_name is not defined.
```

code 테이블의 정의에서 s_name 컬럼이 이전에 제거되었기 때문에 **INSERT** 문의 실행은 실패한다. code 테이블에 입력하려고 했던 데이터는 틀리지 않으나 테이블에서 컬럼이 잘못 제거되었다. 이 시점에서 code 테이블의 원래 정의를 복원하기 위해서 **ROLLBACK WORK** 문을 사용할 수 있다.

```
;rollback work
```

이후에 **ALTER CLASS** 명령을 다시 입력하여 s_name 컬럼을 제거하며, **INSERT** 문을 수정한다. 트랜잭션이 중단되었기 때문에 **INSERT** 명령은 다시 입력되어야 한다. 데이터베이스 갱신이 의도한 대로 이루어졌으면 변경을 영구화하기 위해 트랜잭션을 커밋한다.

```
alter class code
drop s name;

insert into code (f name)
values ('Diamond');

;commit work
```

세이프포인트와 부분 롤백

설명

세이프포인트(savepoint)는 트랜잭션이 진행되는 중에 수립되는데, 트랜잭션에 의해 수행되는 데이터베이스 갱신을 세이프포인트 지점까지만 롤백할 수 있도록 하기 위해서이다. 이 연산을 부분 롤백(partial rollback)이라고 부른다. 부분 롤백에서는 세이프포인트 이후의 데이터베이스 연산(삽입, 삭제, 갱신 등)은 하지 않은 것으로 되고 세이프포인트 지점을 포함하여 이전에 진행된 트랜잭션의 연산은 그대로 유지된다. 부분 롤백이 실행된 후에 트랜잭션은 다른 연산을 계속 진행할 수 있다. 또는 **COMMIT WORK** 문이나 **ROLLBACK WORK** 문으로 트랜잭션을 끝낼 수도 있다. 세이프포인트는 트랜잭션에서 수행된 갱신을 커밋하는 것이 아님을 명심해야 한다.

세이프포인트는 트랜잭션의 어느 시점에서도 만들 수 있고 몇 개의 세이프포인트라도 어떤 주어진 시점에 사용될 수 있다. 특정 세이프포인트보다 앞선 세이프포인트로 부분 롤백이 수행되거나 **COMMIT WORK** 또는 **ROLLBACK WORK** 문으로 트랜잭션이 끝나면 특정 세이프포인트는 제거된다. 특정 세이프포인트 이후에 대한 부분 롤백은 여러 번 수행될 수 있다.

세이프포인트는 길고 복잡한 프로그램을 통제할 수 있도록 중간 단계를 만들고 이름을 붙일 수 있기 때문에 유용하다. 예를 들어, 많은 갱신 연산 수행 시 세이프포인트를 사용하면 실수를 했을 때 모든 문장을 다시 수행할 필요가 없다.

구문

```
SAVEPOINT mark;
mark:
- a SQL identifier
- a host variable (starting with :)
```

같은 트랜잭션 내에 여러 개의 세이프포인트를 지정할 때 *mark*를 같은 값으로 하면 마지막 세이프포인트만 부분 롤백에 나타난다. 그리고 이전의 세이프포인트는 제일 마지막 세이프포인트로 부분 롤백할 때까지 감춰졌다가 제일 마지막 세이프포인트가 사용된 후 없어지면 나타난다.

구문

```
ROLLBACK [ WORK ] [ TO [ SAVEPOINT ] mark ] [ ; ]
mark:
- a SQL identifier
- a host variable (starting with :)
```

앞에서는 **ROLLBACK WORK** 문이 마지막 트랜잭션 이후로 입력된 모든 데이터베이스의 갱신을 제거하였다. **ROLLBACK WORK** 문은 특정 세이프포인트 이후로 트랜잭션의 갱신을 되돌리는 부분 롤백에도 사용된다.

*mark*의 값이 주어지지 않으면 트랜잭션은 모든 갱신을 취소하면서 종료한다. 여기에는 트랜잭션에 만들어진 모든 세이브포인트도 포함한다. *mark*가 주어지면 지정한 세이브포인트 이후의 것은 취소되고, 세이브포인트를 포함한 이전의 것은 갱신 사항이 남는다.

예제

다음 예제는 트랜잭션의 일부를 롤백하는 방법을 보여준다.

먼저 savepoint SP1, SP2를 설정한다.

```
create class athlete2 (name varchar(40), gender char(1), nation_code char(3), event
varchar(30));
insert into athlete2(name, gender, nation_code, event)
values ('Lim Kye-Sook', 'W', 'KOR', 'Hockey');
savepoint SP1;

select * from athlete2;
insert into athlete2(name, gender, nation_code, event)
values ('Lim Jin-Suk', 'M', 'KOR', 'Handball');

select * from athlete2;
savepoint SP2;

rename class athlete2 as sportsman;
select * from sportsman;
rollback work to SP2;
;xrun
```

위에서 athlete2 테이블의 이름 변경은 위의 부분 롤백에 의해서 롤백된다. 다음의 문장은 원래의 이름으로 질의를 수행하여 이것을 검증하고 있다.

```
select * from athlete2;
delete from athlete2 where name = 'Lim Jin-Suk';
select * from athlete2;
rollback work to SP2;
;xrun
```

위에서 'Lim Jin-Suk' 을 삭제한 것은 이후에 진행되는 rollback work to SP2 명령문에 의해서 취소되었다.

다음은 SP1으로 롤백하는 경우이다.

```
select * from athlete2;
rollback work to SP1;
select * from athlete2;
commit work;
;xrun
```

데이터베이스 동시성

다수의 사용자들이 데이터베이스에서 읽고 쓰는 권한을 가질 때, 한 명 이상의 사용자가 동시에 같은 데이터에 접근할 가능성이 있다. 데이터베이스의 무결성을 보호하고, 사용자와 트랜잭션이 항상 정확하고 일관된 데이터를 지니기 위해서는 다중 사용자 환경에서의 접근과 갱신에 대한 통제가 필수적이다. 적절한 통제가 없으면 데이터는 어긋난 순서로 부정확하게 갱신될 수 있다.

대부분의 상용 데이터베이스 시스템과 마찬가지로 CUBRID도 데이터베이스 내의 동시성(concurrency)을 위한 기본 요소인 직렬성(serializability)을 수용한다. 직렬성이란 여러 트랜잭션이 동시에 수행될 때, 마치 각각의 트랜잭션이 순차적으로 수행되는 것처럼 트랜잭션 간 간섭이 없다는 것을 의미하며, 트랜잭션의

격리 수준(isolation level)이 높을수록 보장된다. 이러한 원칙은 원자성(atomic, 트랜잭션의 모든 영향들은 커밋되거나 롤백되어야 함)을 갖는 트랜잭션이 각각 수행된다면, 데이터베이스의 동시성이 보장된다는 가정에 기초하고 있다. CUBRID에서 직렬성은 잘 알려진 2단계 잠금 기법을 통해 관리된다. 이것은 [잠금 프로토콜](#)에서 설명한다.

커밋하고자 하는 트랜잭션은 데이터베이스의 동시성을 보장하고, 적합한 결과를 보장해야 한다. 여러 트랜잭션이 동시에 수행 중일 때, 트랜잭션 T1 내의 이벤트는 트랜잭션 T2에 영향을 끼치지 않아야 하며, 이를 격리성(isolation)이라 한다. 즉, 트랜잭션의 격리 수준(isolation level)은 동시에 수행되는 다른 트랜잭션으로부터 간섭받는 것을 허용하는 정도의 단위이다. 격리 수준이 높을수록 트랜잭션 간 간섭이 적으며 직렬적이고, 격리 수준이 낮을수록 트랜잭션 간 간섭이 많고 병렬적이며 동시성이 높아진다. 이러한 트랜잭션의 격리 수준에 따라 데이터베이스는 테이블과 레코드에 대해 어떤 잠금을 획득할지 결정한다. 따라서, 적용하고자 하는 서비스의 특성에 따라 격리 수준을 적절히 설정함으로써 데이터베이스의 일관성(consistency)과 동시성(concurrency)을 조정할 수 있다.

사용자는 [SET TRANSACTION ISOLATION LEVEL](#) 문을 사용하거나 CUBRID가 지원하는 동시성/잠금 파라미터를 이용하여 격리 수준을 설정할 수 있다. 이에 관한 설명은 [동시성/잠금 파라미터](#)를 참조한다.

트랜잭션 격리 수준 설정을 통해 트랜잭션 간 간섭을 허용할 수 있는 읽기 연산의 종류는 다음과 같다.

- **더티 읽기(dirty read):** 트랜잭션 T1가 데이터 D를 D'로 갱신한 후 커밋을 수행하기 전에 트랜잭션 T2가 D'를 읽을 수 있다.
- **반복할 수 없는 읽기(non-repeatable read, unrepeatable read):** 트랜잭션 T1이 데이터를 여러 번 조회하는 중에 다른 트랜잭션 T2가 데이터를 수정하는 경우, 트랜잭션 T1은 다른 값을 읽을 수 있다.
- **유령 읽기(phantom read):** 트랜잭션 T1에서 데이터를 여러 번 조회하는 중에 다른 트랜잭션 T2가 새로운 레코드 E를 삽입한 경우, 트랜잭션 T1은 E를 읽을 수 있다.

CUBRID는 행(row) 단위는 물론 인덱스, 테이블, 데이터베이스 단위로 잠금/해제를 수행할 수 있으며, 총 6개의 트랜잭션 격리 수준을 제공하므로 타 DBMS에 비해 보다 세밀하게 동시성 레벨을 조정할 수 있다.

CUBRID가 제공하는 격리 수준

CUBRID 격리 수준 (isolation_level)	타 DBMS 격리 수준 (isolation_level)	더티 반복할 수 유령 비교 읽기 없는 읽기 읽기			
SERIALIZABLE (6)	SERIALIZABLE (4)	N	N	N	데이터 일관성 보장. 잠금으로 인한 오버헤드 높음.
REPEATABLE READ CLASS with REPEATABLE READ INSTANCES (3) (5)	REPEATABLE READ	N	N	Y	조회 중인 테이블에 대한 스키마 갱신을 허용하지 않음.
REPEATABLE READ CLASS with	READ COMMITTED	N	Y	Y	조회 중인 테이블

READ COMMITTED INSTANCES (2)					에 대한 스키마 갱신을 허용하지 않음.
(4)					
REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES (3)	READ UNCOMMITTED (1)	Y	Y	Y	디폴트 설정. 조회 중인 테이블에 대한 스키마 갱신을 허용하지 않음.
READ COMMITTED CLASS with READ COMMITTED INSTANCES (2)		N	Y	Y	조회 중인 테이블에 대한 스키마 갱신을 허용.
READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES (1)		Y	Y	Y	조회 중인 테이블에 대한 스키마 갱신을 허용.

잠금 프로토콜

개요

CUBRID는 동시성 제어를 위해 2단계 잠금 프로토콜(2-phase locking protocol, 2PL)을 사용하여 트랜잭션 스케줄을 관리한다. 이는 트랜잭션이 사용하는 자원, 즉 객체에 대해 상호 배제 기능을 제공하는 기법이다. 확장 단계(growing phase)에서는 트랜잭션들이 잠금 연산만 수행할 수 있고 잠금 해제(unlock) 연산은 수행할 수 없다. 축소 단계(shirinking phase)에서는 트랜잭션들이 잠금 해제(unlock) 연산만 수행할 수 있고 잠금 연산은 수행할 수 없다. 즉, 트랜잭션 T1이 특정 객체에 대해 읽기 또는 갱신 연산을 수행하기 전에 반드시 잠금 연산을 먼저 수행하고, T1을 종료하기 전에 잠금 해제 연산을 수행해야 한다.

잠금의 단위

CUBRID는 잠금의 개수를 줄이기 위해서 단위 잠금(granularity locking) 프로토콜을 사용한다. 단위 잠금 프로토콜에서는 잠금 단위의 크기에 따라 계층으로 모델화되며, 행 잠금(row lock), 테이블 잠금(table lock), 데이터베이스 잠금(database lock)이 있다. 이때, 단위가 큰 잠금은 작은 단위의 잠금을 내포한다.

잠금을 설정하고 해제하는 과정에서 발생하는 성능 손실을 잠금 비용(overhead)이라고 하는데, 큰 단위보다 작은 단위의 잠금을 수행할 때 이러한 잠금 비용이 높아지고 대신 트랜잭션 동시성은 향상된다. 따라서, CUBRID는 잠금 비용과 트랜잭션 동시성을 고려하여 잠금 단위를 결정한다. 예를 들어, 한 트랜잭션이 테이블의 모든 행들을 조회하는 경우 행 단위로 잠금을 설정/해제하는 비용이 너무 높으므로 차라리 해당 테이블에 잠금을 설정한다. 이처럼 테이블에 잠금이 설정되면 트랜잭션 동시성이 저하되므로, 동시성을 보장하려면 풀 스캔(full scan)이 발생하지 않도록 적절한 인덱스를 사용해야 할 것이다.

이와 같은 잠금 관리를 위해 CUBRID는 잠금 에스컬레이션(lock escalation) 기법을 사용하여 설정 가능한 단위 잠금의 수를 제한한다. 예를 들어, 한 트랜잭션이 행 단위에서 특정 개수 이상의 잠금을 가지고 있으면 시스템은 계층적으로 상위 단위인 테이블에 대해 잠금을 요청하기 시작한다. 단, 상위 단위로 잠금 에스컬레이션을 수행하기 위해서는 어떤 트랜잭션도 상위 단위 객체에 대한 잠금을 가지고 있지 않아야 한다. 그래야만 잠금 변환에 따른 교착 상태(deadlock)를 예방할 수 있다. 이때, 소정의 단위에서 허용하는 잠금 개수는 시스템 파라미터 **lock_escalation**을 통해 설정할 수 있다.

잠금 모드의 종류와 호환성

CUBRID는 트랜잭션이 수행하고자 하는 연산의 종류에 따라 획득하고자 하는 잠금 모드를 결정하며, 다른 트랜잭션에 의해 이미 선점된 잠금 모드의 종류에 따라 잠금 공유 여부를 결정한다. 이와 같은 잠금에 대한 결정은 시스템이 자동으로 수행하며, 사용자에 의한 수동 지정은 허용되지 않는다. CUBRID의 잠금 정보를 확인하기 위해서는 **cubrid lockdb db_name** 명령어를 사용하며, 자세한 내용은 [잠금\(lock\) 상태 확인](#)을 참고한다.

- **공유 잠금(shared lock, S_LOCK)** : 객체에 대해 읽기 연산을 수행하기 전에 획득하며, 여러 트랜잭션이 동일 객체에 대해 획득할 수 있는 잠금이다.

트랜잭션 T1이 특정 객체 X에 대해 읽기 연산을 수행하기 전에 공유 잠금을 먼저 획득하고, 트랜잭션 T1이 커밋되기 전이라도 읽기 연산을 완료하면 즉시 획득한 공유 잠금을 해제한다. 이때, 트랜잭션 T2, T3은 동시에 X에 대해 읽기 연산은 수행할 수 있으나 갱신 연산은 수행할 수 없다.

- **배타 잠금(exclusive lock, X_LOCK)** : 객체에 대해 갱신 연산을 수행하기 전에 획득하며, 하나의 트랜잭션만 획득할 수 있는 잠금이다.

트랜잭션 T1이 특정 객체 X에 대해 갱신 연산을 수행하기 전에 배타 잠금을 먼저 획득하고, 갱신 연산을 완료하더라도 트랜잭션 T1이 커밋될 때까지 배타 잠금을 해제하지 않는다. 따라서, 트랜잭션 T2, T3은 트랜잭션 T1이 배타 잠금을 해제하기 전까지는 X에 대한 읽기 연산도 수행할 수 없다.

- **갱신 잠금(update lock, U_LOCK)** : 갱신 연산을 수행하기 전, 조건절에서 읽기 연산을 수행할 때 획득하는 잠금이다.

예를 들어 **WHERE** 절과 결합된 **UPDATE** 문을 수행하는 경우, **WHERE** 절에서 인덱스 검색을 하거나 풀 스캔 검색을 수행할 때 행 단위로 갱신 잠금을 획득하고, 조건을 만족하는 결과 행들에 대해서만 배타 잠금을 획득하여 갱신 연산을 수행한다. 이처럼 갱신 잠금은 실제 갱신 연산을 수행할 때 배타 잠금으로 변환되며, 이는 다른 트랜잭션이 동일한 객체에 대해 읽기 연산을 수행하지 못하도록 하므로 준 배타 잠금이라고 할 수 있다.

- **의도 잠금(내재된 잠금, intention lock)** : 특정 단위의 객체 X에 걸리는 잠금을 보호하기 위하여 X보다 상위 단위의 객체에 내재적으로 설정하는 잠금을 의미한다.

예를 들어, 특정 행에 공유 잠금이 요청되면 이보다 계층적으로 상위에 있는 테이블에도 의도 공유 잠금을 함께 설정하여 다른 트랜잭션에 의해 테이블이 잠금되는 것을 예방한다. 따라서, 의도 잠금은 계층적으로 가장 낮은 단위인 행에 대해서는 설정되지 않으며, 이보다 높은 단위의 객체에 대해서만 설정된다. 의도 잠금의 종류는 다음과 같다.

- **의도 공유 잠금(intention shared lock, IS_LOCK)** : 특정 행에 공유 잠금이 설정됨에 따라 상위 객체인 테이블에 의도 공유 잠금이 설정되면, 다른 트랜잭션은 컬럼을 추가하거나 테이블 이름을 변경하는 등의 테이블 스키마를 변경할 수 없고, 모든 행을 갱신하는 작업을 수행할 수 없다. 그러나 일부 행을 갱신하는 작업이나, 모든 행을 조회하는 작업은 허용된다.
- **의도 배타 잠금(intention exclusive lock, IX_LOCK)** : 특정 행에 배타 잠금이 설정됨에 따라 상위 객체인 테이블에 의도 배타 잠금이 설정되면, 다른 트랜잭션은 테이블 스키마를 변경할 수 없고, 모든 행을 갱신하는 작업은 물론, 모든 행을 조회하는 작업은 수행할 수 없다. 그러나, 일부 행을 갱신하는 작업은 허용된다.
- **공유 의도 배타 잠금(shared with intent exclusive, SIX_LOCK)** : 계층적으로 더 낮은 모든 객체에 설정된 공유 잠금을 보호하고, 계층적으로 더 낮은 일부 객체에 대한 의도 배타 잠금을 보호하기 위하여 상위 객체에 내재적으로 설정되는 잠금이다.

테이블에 공유 의도 배타 잠금이 설정되면, 다른 트랜잭션은 테이블 스키마를 변경할 수 없고, 모든 행/일부 행을 갱신할 수 없으며, 모든 행을 조회할 수 없다. 그러나, 일부 행을 조회하는 작업은 허용된다.

위에서 설명한 잠금들의 호환 관계(lock compatibility)를 정리하면 아래의 표와 같다. 호환된다는 것은 잠금 유지자(lock holder)가 객체 X에 대해 획득한 잠금과 중복하여 잠금 요청자(lock requester)가 잠금을 획득할 수 있다는 의미다. 한편, N/A는 해당 사항이 없음을 의미한다.

잠금 호환성

		잠금 유지자(lock holder)						
		NULL_LOCK	IS_LOCK	S_LOCK	IX_LOCK	SIX_LOCK	U_LOCK	X_LOCK
잠금 요청자 (lock requester)	NULL_LOCK	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
	IS_LOCK	TRUE	TRUE	TRUE	TRUE	TRUE	N/A	FALSE
	S_LOCK	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
	IX_LOCK	TRUE	TRUE	FALSE	TRUE	FALSE	N/A	FALSE
	SIX_LOCK	TRUE	TRUE	FALSE	FALSE	FALSE	N/A	FALSE
	U_LOCK	TRUE	N/A	TRUE	N/A	N/A	FALSE	FALSE
	X_LOCK	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

- **NULL_LOCK** : 아무 잠금도 없는 상태

예제

session 1	session 2
<pre> ;autocommit off AUTOCOMMIT IS OFF set transaction isolation level 4 ;xr Isolation level set to: REPEATABLE READ SCHEMA, READ COMMITTED INSTANCES. </pre>	<pre> ;autocommit off AUTOCOMMIT IS OFF set transaction isolation level 4 ;xr Isolation level set to: REPEATABLE READ SCHEMA, READ COMMITTED INSTANCES. </pre>

1 command(s) successfully processed.	1 command(s) successfully processed. /* C:\CUBRID>cubrid lockdb demodb *** Lock Table Dump *** ... Object Lock Table: Current number of objects which are locked = 0 Maximum number of objects which can be locked = 10000 ... */
select nation code, gold from participant where nation_code='USA'; ;xr === <Result of SELECT Command> === nation_code gold ===== 'USA' 36 'USA' 37 'USA' 44 'USA' 37 'USA' 36 5 rows selected. 1 command(s) successfully processed. /* C:\CUBRID>cubrid lockdb demodb *** Lock Table Dump *** ... Object type: Root class. LOCK HOLDERS: Tran index = 2, Granted mode = IS_LOCK, Count = 1, Nsubgranules = 1 Object type: Class = participant. LOCK HOLDERS: Tran index = 2, Granted mode = IS_LOCK, Count = 2, Nsubgranules = 0 */	update participant set gold = 11 where nation_code = 'USA' ; ;xr 5 rows affected.
select nation code, gold from participant where nation_code='USA'; ;xr /* no results until transaction 2 releases a	


```

lock

C:\CUBRID>cupbrid lockdb demodb
*** Lock Table Dump ***
...

Object type: Instance of class
( 0| 551| 7) = participant.
LOCK HOLDERS:
    Tran index = 3, Granted mode
= X_LOCK, Count = 2
...

Object type: Root class.
LOCK HOLDERS:
    Tran index = 3, Granted mode
= IX_LOCK, Count = 1, Nsubgranules = 3
NON_2PL_RELEASED:
    Tran index = 2, Non 2 phase lock
= IS_LOCK
...

Object type: Class = participant.
LOCK HOLDERS:
    Tran index = 3, Granted mode
= IX_LOCK, Count = 3, Nsubgranules = 5
    Tran index = 2, Granted mode
= IS_LOCK, Count = 2, Nsubgranules = 0
*/

;commit work

Current transaction has been
committed.

=== <Result of SELECT Command> ===
nation_code      gold
=====
'USA'            11
'USA'            11
'USA'            11
'USA'            11
'USA'            11

5 rows selected.

1 command(s) successfully processed.

/*
C:\CUBRID>cupbrid lockdb demodb
...

Object type: Root class.
LOCK HOLDERS:
    Tran index = 2, Granted mode
= IS_LOCK, Count = 1, Nsubgranules = 1

Object type: Class = participant.
LOCK HOLDERS:
    Tran index = 2, Granted mode
= IS_LOCK, Count = 3, Nsubgranules = 0

```

```

...
*/
;commit work

Current transaction has been committed.

/*
C:\CUBRID>cubrid lockdb demodb
...

Object Lock Table:
      Current number of objects which are
locked      = 0
      Maximum number of objects which can
be locked = 10000
*/

```

트랜잭션 교착 상태(deadlock)

교착 상태(deadlock)은 둘 이상의 트랜잭션이 서로 맞물려 상대방의 잠금이 해제되기를 기다리는 상태이다. 이러한 교착 상태에서는 서로가 상대방의 작업 수행을 차단하기 때문에 CUBRID는 트랜잭션 중 하나를 롤백시켜 교착 상태를 해결한다. 롤백되는 트랜잭션은 일반적으로 가장 적은 갱신을 수행한 것인데 보통 가장 최근에 시작된 트랜잭션이다. 시스템에 의해 트랜잭션이 롤백되자마자 그 트랜잭션이 가지고 있던 잠금이 해제되고 교착 상태에 있던 다른 트랜잭션이 진행되도록 허가된다.

이러한 교착 상태가 발생하는 상황은 예측 불가능하지만, 가급적 발생 상황을 줄이기 위해서 인덱스를 설정하여 잠금이 설정되는 범위를 줄이거나, 트랜잭션을 짧게 만들거나, 트랜잭션 격리 수준(isolation level)을 낮게 설정하는 것이 좋다.

예제

session 1	session 2
<pre> ;autocommit off AUTOCOMMIT IS OFF set transaction isolation level 6 ;xr Isolation level set to: SERIALIZABLE 1 command(s) successfully processed. </pre>	<pre> ;autocommit off AUTOCOMMIT IS OFF set transaction isolation level 6 ;xr Isolation level set to: SERIALIZABLE 1 command(s) successfully processed. </pre>
<pre> CREATE TABLE lock_tbl(host year integer, nation_code char(3)); INSERT INTO lock_tbl VALUES (2004, 'KOR'); INSERT INTO lock_tbl VALUES (2004, 'USA'); INSERT INTO lock_tbl VALUES (2004, 'GER'); INSERT INTO lock_tbl VALUES (2008, 'GER'); COMMIT; ;xr 6 command(s) successfully processed. </pre>	

```

SELECT * FROM lock_tbl;
;xr

=== <Result of SELECT Command> ===

  host_year  nation_code
=====
      2004   'KOR'
      2004   'USA'
      2004   'GER'
      2008   'GER'

4 rows selected.

```

```

SELECT * FROM lock_tbl;
;xr

=== <Result of SELECT Command> ===

  host_year  nation_code
=====
      2004   'KOR'
      2004   'USA'
      2004   'GER'
      2008   'GER'

4 rows selected.

```

```

DELETE FROM lock_tbl WHERE
host_year=2008;
;xr

/* no result until transaction 2
releases a lock

C:\CUBRID> cubrid lockdb demodb
*** Lock Table Dump ***
...

Object type: Class = lock_tbl.
LOCK HOLDERS:
  Tran index = 2, Granted mode
= S LOCK, Count = 2, Nsubgranules
= 0

BLOCKED LOCK HOLDERS:
  Tran index = 1, Granted mode
= S LOCK, Count = 3, Nsubgranules
= 0
  Blocked_mode = SIX_LOCK
  Start waiting at = Fri Feb 12
14:22:58 2010
  Wait_for_nsecs = -1

*/

```

```

INSERT INTO lock_tbl VALUES (2004,
'AUS');
;xr

1 rows affected.

```

In the command from line 1,

```

ERROR: Your transaction (index 1,
dba@ 090205|4760) has been
unilaterally aborted by the system.

0 command(s) successfully processed.

/*
System rolled back the transaction 1
to resolve a deadlock

C:\CUBRID> cubrid lockdb demodb
*** Lock Table Dump ***

Object type: Class = lock_tbl.

LOCK HOLDERS:
    Tran index =    2, Granted mode =
SIX LOCK, Count =    3, Nsubgranules
=    0
*/

```

트랜잭션 잠금 타임아웃

CUBRID는 트랜잭션 잠금 설정이 허용될 때까지 잠금을 대기하는 시간을 설정하는 잠금 타임아웃(lock timeout) 기능을 제공한다.

만약 설정된 잠금 타임아웃 시간 이내에 잠금이 허용되지 않으면, 잠금 타임아웃 시간이 경과된 시점에 해당 트랜잭션을 롤백시키고 에러를 출력한다. 또한, 잠금 타임아웃 시간 이내에 트랜잭션 교착상태가 발생하면, CUBRID는 교착 상태에 있는 여러 트랜잭션 중 대기시간이 타임아웃 시간에 가까운 트랜잭션을 롤백시킨다.

잠금 타임아웃 값 설정

설명

\$CUBRID/conf/cubrid.conf 파일 내의 시스템 파라미터 **lock_timeout_in_secs** 또는 **SET TRANSACTION** 구문을 통해 응용 클라이언트가 잠금을 대기하는 타임아웃 시간(초 단위)을 설정하며, 설정된 시간이 경과된 이후에는 해당 트랜잭션을 롤백시키고 에러를 출력한다. **lock_timeout_in_secs** 파라미터의 디폴트 값은 -1이며, 이는 트랜잭션 잠금이 허용되는 시점까지 무한정 대기한다는 의미이다. 따라서, 사용자는 응용 클라이언트의 트랜잭션 패턴에 맞게 이 값을 변경할 수 있다. 만약, 잠금 타임아웃 값이 0으로 설정되면 잠금이 발생하는 즉시 에러 메시지가 출력될 것이다.

구문

```

SET TRANSACTION LOCK TIMEOUT timeout_spec [ ; ]
timeout_spec:
- INFINITE
- OFF
- unsigned integer
- variable

```

- **INFINITE** : 트랜잭션 잠금이 허용될 때까지 무한정 대기한다. 시스템 파라미터 **lock_timeout_in_secs**를 -1로 설정한 것과 같다.
- **OFF** : 잠금을 대기하지 않고, 해당 트랜잭션을 롤백시킨 후 에러 메시지를 출력한다. 시스템 파라미터 **lock_timeout_in_secs**를 0으로 설정한 것과 같다.
- *unsigned_integer* : 초 단위로 설정되며, 설정된 시간만큼 트랜잭션 잠금을 대기한다.
- *variable* : 변수를 지정할 수 있으며, 변수에 저장된 값만큼 트랜잭션 잠금을 대기한다.

예제 1

```
vi $CUBRID/conf/cubrid.conf
...
lock_timeout_in_secs = 10
...
```

예제 2

```
csql> SET TRANSACTION LOCK TIMEOUT 10;
csql> ;xr
```

잠금 타임아웃 값 확인

설명

GET TRANSACTION 문을 이용하여 현재 응용 클라이언트에 설정된 잠금 타임아웃 값을 확인할 수 있고, 이 값을 변수에 저장할 수도 있다.

구문

```
GET TRANSACTION LOCK TIMEOUT [ { INTO | TO } variable ] [ ; ]
```

예제

```
csql> GET TRANSACTION LOCK TIMEOUT;
csql> ;xr

=== <Result of GET LOCK TIMEOUT Command in Line 1> ===

      Result
=====
1.000000e+001
```

잠금 타임아웃 에러 메시지 확인과 조치 방법

다른 트랜잭션의 잠금이 해제되기를 대기하던 트랜잭션에 대해 잠금 타임아웃이 발생하면, 아래와 같은 에러 메시지를 출력한다. 잠금 타임아웃 에러 메시지에서 출력하는 정보의 수준을 보다 상세하게 설정하려면, [동시성/잠금 파라미터](#)에서 **lock_timeout_message_type**의 설명을 참고한다.

```
ERROR: Your transaction (index 3, cub user@cdb006.cub|15668) timed out waiting
on X_LOCK lock on instance 0|636|34 of class participant. You are waiting for
user(s) to finish.
```

- Your transaction(index 3 ...): 잠금을 대기하다가 타임아웃으로 롤백된 트랜잭션의 인덱스가 3이라는 의미이다. 트랜잭션 인덱스는 클라이언트가 데이터베이스 서버에 접속하였을 때 순차적으로 할당되는 번호이다. 이는 **cubrid lockdb** 유틸리티 실행을 통해서도 확인할 수 있다.

- (...cub_user@cubs006.cub|15668) : cub_user는 클라이언트의 로그인 아이디이고, @의 뒷 부분은 클라이언트가 수행된 호스트 이름이다. 또한 | 의 뒷 부분은 클라이언트의 프로세스 ID(PID)이다.
- X_LOCK : 데이터 갱신을 수행하기 위해 객체에 설정하는 배타 잠금을 의미한다. 이에 관한 상세한 설명은 [잠금 모드의 종류와 호환성](#)을 참고한다.
- Instance 0|636|34 of class participant : participant 라는 테이블 내의 특정 행에 대해 **X_LOCK**이 설정되어 있으며, 특정 행의 OID(해당 객체에 할당된 고유한 ID)가 0|636|34 라는 의미이다.

즉, 위의 잠금 예러 메시지는 "다른 클라이언트가 participant 테이블의 특정 행에 **X_LOCK**을 점유하고 있으므로, cubs006.cub 호스트에서 수행된 트랜잭션 3은 잠금을 대기하다가 타임아웃 시간이 경과되어 롤백되었다."로 해석할 수 있다.

만약, 예러 메시지에 명시된 트랜잭션의 잠금 정보를 확인하고자 한다면, **cubrid lockdb** 유틸리티를 통해 현재 **X_LOCK**이 설정된 특정 행의 OID 값(예제: 0|636|34)과 동일한 객체를 검색하여 현재 잠금을 점유 중인 클라이언트의 트랜잭션 ID 값, 클라이언트 프로그램 이름, 프로세스 ID(PID)를 확인할 수 있다. 이에 관한 상세한 설명은 [잠금\(Lock\) 상태 확인](#)을 참고한다. 또한, CUBRID 매니저에서도 트랜잭션의 잠금 정보를 확인할 수 있으며, 이에 관한 상세한 설명은 [데이터베이스 잠금 정보](#)를 참고한다.

이처럼 트랜잭션의 잠금 정보를 확인한 후에는 SQL 로그를 통해 커밋되지 않은 질의문을 확인하여 트랜잭션을 정리할 수 있다. SQL 로그를 확인하는 방법은 [브로커 로그](#) 또는 [SQL 로그](#)를 참고한다.

또한, **cubrid killtran** 유틸리티를 통해 문제가 되는 트랜잭션을 강제 종료할 수 있으며, 이에 관한 상세한 설명은 [트랜잭션 제거](#)를 참고한다.

트랜잭션 격리 수준

개요

트랜잭션의 격리 수준은 트랜잭션이 동시에 진행 중인 다른 트랜잭션에 의해 간섭받는 정도를 의미하며, 트랜잭션 격리 수준이 높을수록 트랜잭션 간 간섭이 적으며 직렬적이고, 트랜잭션 격리 수준이 낮을수록 트랜잭션 간 간섭은 많으나 높은 동시성을 보장한다. 사용자는 적용하고자 하는 서비스의 특성에 따라 격리 수준을 적절히 설정함으로써 데이터베이스의 일관성(consistency)과 동시성(concurrency)을 조정할 수 있다.

참고 지원되는 모든 격리 수준에서 트랜잭션은 복구 가능하다. 이는 트랜잭션이 끝나기 전에는 갱신을 커밋하지 않기 때문이다.

격리 수준 설정

설명

\$CUBRID/conf/cubrid.conf 파일 내의 시스템 파라미터 **isolation_level**과 **SET TRANSACTION** 문을 사용하면, 응용 클라이언트에서 수행되는 트랜잭션 격리 수준을 설정할 수 있다. 디폴트로 설정된 격리

수준은 **REPEATABLE READ CLASS, READ UNCOMMITTED INSTANCES**이며, CUBRID가 제공하는 1부터 6까지의 격리 수준 중에 3에 해당한다. 이에 관한 상세한 설명은 [데이터베이스 동시성](#)을 참고한다.

구문

```
SET TRANSACTION ISOLATION LEVEL isolation_level_spec [ ; ]
isolation_level_spec:
- SERIALIZABLE
- CURSOR STABILITY
- isolation_level [ { CLASS | SCHEMA } [ , isolation_level INSTANCES ] ]
- isolation_level [ INSTANCES [ , isolation_level { CLASS | SCHEMA } ] ]
- variable
isolation_level:
- REPEATABLE READ
- READ COMMITTED
- READ UNCOMMITTED
```

예제 1

```
vi $CUBRID/conf/cubrid.conf
```

```
...
isolation_level = 1
...
```

또는

```
isolation_level = "TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE"
```

예제 2

```
csql> SET TRANSACTION ISOLATION LEVEL 1;
csql> ;xr
```

또는

```
csql> SET TRANSACTION ISOLATION LEVEL READ COMMITTED CLASS,READ UNCOMMITTED INSTANCES;
csql> ;xr
```

아래의 표는 CUBRID에서 지원하는 1에서 6까지의 격리 수준에 관한 설명이다. 이는 테이블 스키마와 행(row)에 대한 격리 수준 조합으로 구성되며, CUBRID에서 허용되지 않는 격리 수준의 조합은 [지원하지 않는 격리 수준 조합](#)을 참고한다.

CUBRID가 지원하는 격리 수준

격리 수준 이름	설명
SERIALIZABLE (6)	모든 동시성 관련한 모든 문제들(더티 읽기, 반복 불가능한 읽기, 유령 읽기)이 발생하지 않는다
REPEATABLE READ CLASS with	트랜잭션 T1 이 테이블 A 를 조회하는 중에 다른 트랜잭션 T2 가 테이블 A 의 스키마를 갱신할 수 없다.
REPEATABLE READ INSTANCES (5)	트랜잭션 T1 이 특정 레코드를 여러 번 조회하는 중에, 다른 트랜잭션 T2 가 삽입한 레코드 R 에 대한 유령 읽기를 경험할 수 있다.
REPEATABLE READ CLASS with	트랜잭션 T1 이 테이블 A 를 조회하는 중에 다른 트랜잭션 T2 가 테이블 A 의 스키마를 갱신할 수 없다.
READ COMMITTED INSTANCES	트랜잭션 T1 이 레코드 R 을 여러 번 조회하는 중에, 다른 트랜잭션 T2 가 갱신하고 커밋한 R' 읽기(반복 불가능한 읽기)를 경험할 수

(or **CURSOR STABILITY**) 있다.

(4)

REPEATABLE READ	격리 수준 디폴트 값.
CLASS with	트랜잭션 T1 이 테이블 A 를 조회하는 중에, 다른 트랜잭션 T2 가
READ UNCOMMITTED	테이블 A 의 스키마를 갱신할 수 없다.
INSTANCES (3)	트랜잭션 T1 이 다른 트랜잭션 T2 가 갱신한 후 커밋하지 않은 레 코드 R' 읽기(더티 읽기)를 경험할 수 있다.
READ COMMITTED	트랜잭션 T1 이 테이블 A 를 여러 번 조회하는 중에, 다른 트랜잭션
CLASS with	T2 가 스키마를 갱신하고 커밋한 테이블 A' 읽기(반복 불가능한 읽 기)를 경험할 수도 있다.
READ COMMITTED	
INSTANCES (2)	트랜잭션 T1 이 레코드 R 을 여러 번 조회하는 중에, 다른 트랜잭션 T2 가 갱신하고 커밋한 R' 읽기(반복 불가능한 읽기)를 경험할 수 있다.
READ COMMITTED	트랜잭션 T1 이 테이블 A 를 여러 번 조회하는 중에, 다른 트랜잭션
CLASS with	T2 가 스키마를 갱신하고 커밋한 테이블 A' 읽기(반복 불가능한 읽 기)를 경험할 수도 있다.
READ UNCOMMITTED	
INSTANCES (1)	트랜잭션 T1 이 다른 트랜잭션 T2 가 갱신한 후 커밋하지 않은 레 코드 R' 읽기(더티 읽기)를 경험할 수 있다.

응용 클라이언트에서 트랜잭션 수행 중에 격리 수준이 변경되면, 수행 중인 트랜잭션의 남은 부분부터 변경된 격리 수준이 적용된다. 따라서, 트랜잭션 수행 중 객체에 대해 이미 획득한 일부 잠금이 새로운 격리 수준이 적용되는 동안 해제될 수도 있다. 이처럼 설정된 격리 수준이 하나의 트랜잭션 전체에 적용되는 것이 아니라 트랜잭션 중간에 변경되어 적용될 수 있기 때문에, 트랜잭션 격리 수준은 트랜잭션 시작 시점(커밋, 롤백, 또는 시스템 재시작 이후)에 변경하는 것이 하는 것이 바람직하다.

격리 수준 값 확인

설명

GET TRANSACTION 문을 이용하여 현재 클라이언트에 설정된 격리 수준 값을 출력하거나 *variable*에 할당할 수 있다. 아래는 격리 수준을 확인하기 위한 구문이다.

구문

```
GET TRANSACTION ISOLATION LEVEL [ { INTO | TO } variable ] [ ; ]
```

예제

```
csql> GET TRANSACTION ISOLATION LEVEL;  
csql> ;xr  
  
Result  
=====  
READ COMMITTED SCHEMA, READ UNCOMMITTED INSTANCES
```


SERIALIZABLE

가장 높은 격리 수준(6)으로서, 더티 읽기, 반복 불가능한 읽기, 유령 읽기 등의 동시성 관련 문제가 발생하지 않는다.

다음과 같은 규칙이 적용된다.

- 트랜잭션 T1은 다른 트랜잭션 T2에서 갱신 중인 레코드를 읽을 수 없고, 수정할 수 없다.
- 트랜잭션 T1은 트랜잭션 T2에서 조회 중인 레코드를 읽을 수 없고, 수정할 수 없다.
- 트랜잭션 T1이 테이블 A의 레코드를 조회하는 중에 다른 트랜잭션 T2가 테이블 A로 새로운 레코드를 삽입할 수 없다.

이 격리 수준은 공유 잠금 및 배타 잠금 모두 2단계 잠금 프로토콜을 따르므로, 연산이 수행되더라도 해당 트랜잭션이 종료될 때까지 잠금이 유지된다.

예제

다음은 동시에 수행되는 트랜잭션의 격리 수준이 **SERIALIZABLE**인 경우 한 트랜잭션에서 객체 읽기 또는 객체 갱신을 수행하는 동안 다른 트랜잭션이 테이블 또는 레코드에 접근할 수 없음을 보여주는 예제이다.

session 1	session 2
<pre>;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 6 ;xr Isolation level set to: SERIALIZABLE</pre>	<pre>;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 6 ;xr Isolation level set to: SERIALIZABLE</pre>
<pre>--creating a table CREATE TABLE isol6 tbl(host year integer, nation_code char(3)); INSERT INTO isol6_tbl VALUES (2008, 'AUS'); COMMIT; ;xr</pre>	<pre>--selecting records from the table SELECT * FROM isol6 tbl WHERE nation_code = 'AUS'; ;xr === <Result of SELECT Command> === host_year nation_code ===== 2008 'AUS' 1 rows selected.</pre>
<pre>INSERT INTO isol6 tbl VALUES (2004, 'AUS'); ;xr</pre>	

/* unable to insert a row until the tran 2 committed */	
1 rows affected.	COMMIT; ;xr
	SELECT * FROM isol6 tbl WHERE nation_code = 'AUS'; ;xr /* unable to select rows until tran 1 committed */
COMMIT; ;xr	=== <Result of SELECT Command> === host_year nation_code =====
	2008 'AUS' 2004 'AUS'
	2 rows selected.
DELETE FROM isol6_tbl WHERE nation_code = 'AUS' and host_year=2008; ;xr /* unable to delete rows until tran 2 committed */	
1 rows affected.	COMMIT; ;xr
1 command(s) successfully processed.	
	SELECT * FROM isol6 tbl WHERE nation_code = 'AUS'; ;xr /* unable to select rows until tran 1 committed */
COMMIT; ;xr	=== <Result of SELECT Command> === host_year nation_code =====
	2004 'AUS'
	1 rows selected.
ALTER TABLE isol6_tbl ADD COLUMN gold INT; ;xr /* unable to alter the table schema until tran 2 committed */	/* repeatable read is ensured while tran_1 is altering table schema */ SELECT * FROM isol6 tbl WHERE nation_code = 'AUS'; ;xr === <Result of SELECT Command> === host_year nation_code =====
	2004 'AUS'
	1 rows selected.
1 command(s) successfully processed.	COMMIT; ;xr
	SELECT * FROM isol6 tbl WHERE nation_code = 'AUS'; ;xr

	/* unable to access the table until tran_1 committed */
COMMIT;	=== <Result of SELECT Command > ===
;xr	host_year nation_code gold
	=====
	2004 'AUS' NULL
	1 rows selected.

REPEATABLE READ CLASS with REPEATABLE READ INSTANCES

비교적 높은 격리 수준(5)으로서, 더티 읽기, 반복 불가능한 읽기가 발생하지 않지만, 유령 읽기는 발생할 수 있다.

다음과 같은 규칙이 적용된다.

- 트랜잭션 T1은 다른 트랜잭션 T2에서 갱신 중인 레코드를 읽을 수 없고, 수정할 수 없다.
- 트랜잭션 T1은 다른 트랜잭션 T2에서 조회 중인 레코드를 읽을 수 없고, 수정할 수 없다.
- 트랜잭션 T1이 테이블 A의 레코드를 조회하는 중에 다른 트랜잭션 T2가 테이블 A로 새로운 레코드를 삽입할 수 있다. 단, 트랜잭션 T1과 T2가 동일한 레코드에 대해 잠금을 설정할 수 없다.

이 격리 수준은 2단계 잠금 프로토콜을 따른다.

예제

다음은 동시에 수행되는 트랜잭션의 격리 수준이 **REPEATABLE READ CLASS with REPEATABLE READ INSTANCES**인 경우 한 트랜잭션에서 객체 읽기를 수행하는 동안 다른 트랜잭션이 새로운 레코드를 추가할 수 있으므로 유령 읽기가 발생할 수 있음을 보여주는 예제이다.

session 1	session 2
<pre> ;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 5 ;xr Isolation level set to: REPEATABLE READ SCHEMA, REPEATABLE READ INSTANCES. --creating a table CREATE TABLE isol5 tbl(host year integer, nation_code char(3)); CREATE UNIQUE INDEX on isol5_tbl(nation_code, host_year); INSERT INTO isol5 tbl VALUES (2008, 'AUS'); INSERT INTO isol5 tbl VALUES (2004, 'AUS'); COMMIT; ;xr </pre>	<pre> ;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 5 ;xr Isolation level set to: REPEATABLE READ SCHEMA, REPEATABLE READ INSTANCES. </pre>

	<pre>--selecting records from the table SELECT * FROM isol5_tbl WHERE nation_code='AUS'; ;xr === <Result of SELECT Command> === host_year nation_code ===== 2004 'AUS' 2008 'AUS' 2 rows selected.</pre>
<pre>INSERT INTO isol5_tbl VALUES (2004, 'KOR'); INSERT INTO isol5_tbl VALUES (2000, 'AUS'); ;xr 2 rows affected. /* able to insert new rows only when locks are not conflicted */</pre>	
	<pre>SELECT * FROM isol5_tbl WHERE nation_code='AUS'; ;xr /* phantom read may occur when tran 1 committed */ === <Result of SELECT Command> === host_year nation_code ===== 2000 'AUS' 2004 'AUS' 2008 'AUS' 3 rows selected.</pre>
<pre>COMMIT; ;xr</pre>	
<pre>DELETE FROM isol5_tbl WHERE nation_code = 'AUS' and host_year=2008; ;xr /* unable to delete rows until tran 2 committed */</pre>	
<pre>1 rows affected. 1 command(s) successfully processed.</pre>	<pre>COMMIT; ;xr</pre>
	<pre>SELECT * FROM isol5_tbl WHERE nation_code = 'AUS'; ;xr /* unable to select rows until tran 1 committed */ === <Result of SELECT Command> === host_year nation_code ===== 2000 'AUS' 2004 'AUS' 2 rows selected.</pre>
<pre>COMMIT; ;xr</pre>	

ALTER TABLE isol5_tbl ADD COLUMN gold INT; ;xr	
/* unable to alter the table schema until tran 2 committed */	/* repeatable read is ensured while tran_1 is altering table schema */ SELECT * FROM isol5_tbl WHERE nation_code = 'AUS'; ;xr === <Result of SELECT Command> === host_year nation_code =====
	2000 'AUS' 2004 'AUS' 2 rows selected.
1 command(s) successfully processed.	COMMIT; ;xr
	SELECT * FROM isol5_tbl WHERE nation_code = 'AUS'; ;xr /* unable to access the table until tran_1 committed */
COMMIT; ;xr	=== <Result of SELECT Command > === host_year nation_code gold =====
	2000 'AUS' NULL 2004 'AUS' NULL 2 rows selected.

REPEATABLE READ CLASS with READ COMMITTED INSTANCES

비교적 낮은 격리 수준(4)으로서 더티 읽기는 발생하지 않지만, 반복 불가능한 읽기와 유령 읽기는 발생할 수 있다. 즉, 트랜잭션 T1이 하나의 객체를 반복하여 조회하는 동안 다른 트랜잭션 T2에서의 삽입 또는 갱신이 허용되어, 트랜잭션 T1이 다른 값을 읽을 수 있다는 의미이다.

다음과 같은 규칙이 적용된다.

- 트랜잭션 T1은 다른 트랜잭션 T2에서 갱신 중인 레코드를 읽을 수 없다.
- 트랜잭션 T1은 다른 트랜잭션 T2에서 조회 중인 테이블에 대해 레코드를 갱신/삽입할 수 있다.
- 트랜잭션 T1은 다른 트랜잭션 T2에서 조회 중인 테이블의 스키마를 변경할 수 없다.

이 격리 수준은 배타 잠금에 대해서는 2단계 잠금을 따른다. 하지만 행에 대한 공유 잠금은 행이 조회된 직후 바로 해제되지만, 테이블에 대한 의도 잠금은 스키마에 대한 반복 가능한 읽기를 보장하기 위하여 트랜잭션이 종료될 때 해제된다.

SET TRANSACTION 문을 수행할 때 격리 수준의 다른 명칭으로 **CURSOR STABILITY** 키워드가 사용될 수 있다.

예제

다음은 동시에 수행되는 트랜잭션의 격리 수준이 **REPEATABLE READ CLASS** with **READ COMMITTED INSTANCES**인 경우 한 트랜잭션에서 객체 읽기를 수행하는 동안 다른 트랜잭션이 새로운 레코드를 추가 또는 갱신할 수 있으므로 유령 읽기 및 반복 불가능한 읽기가 발생할 수 있으나, 테이블 스키마 갱신에 대해서는 반복 가능한 읽기를 보장함을 보여주는 예제이다.

session 1	session 2
<pre> ;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 4 ;xr Isolation level set to: REPEATABLE READ SCHEMA, READ COMMITTED INSTANCES. </pre>	<pre> ;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 4 ;xr Isolation level set to: REPEATABLE READ SCHEMA, READ COMMITTED INSTANCES. </pre>
<pre> --creating a table CREATE TABLE isol4 tbl(host year integer, nation_code char(3)); INSERT INTO isol4 tbl VALUES (2008, 'AUS'); COMMIT; ;xr </pre>	<pre> --selecting records from the table SELECT * FROM isol4_tbl; ;xr === <Result of SELECT Command> === host_year nation_code ===== 2008 'AUS' 1 rows selected. </pre>
<pre> INSERT INTO isol4 tbl VALUES (2004, 'AUS'); INSERT INTO isol4 tbl VALUES (2000, 'NED'); ;xr 2 rows affected. /* able to insert new rows even if tran 2 uncommitted */ </pre>	<pre> SELECT * FROM isol4_tbl; ;xr /* phantom read may occur when tran 1 committed */ </pre>
<pre> COMMIT; ;xr </pre>	<pre> === <Result of SELECT Command> === host_year nation_code ===== 2008 'AUS' </pre>

	<pre> 2004 'AUS' 2000 'NED' 3 rows selected. </pre>
<pre> INSERT INTO isol4_tbl VALUES (1994, 'FRA'); ;xr 1 rows affected. </pre>	
	<pre> SELECT * FROM isol4_tbl; ;xr /* unrepeatable read may occur when tran 1 committed */ </pre>
<pre> DELETE FROM isol4_tbl WHERE nation_code = 'AUS' and host_year=2008; ;xr 1 rows affected. /* able to delete rows while tran 2 is selecting rows*/ </pre>	
<pre> COMMIT; ;xr </pre>	<pre> === <Result of SELECT Command> === host_year nation_code ===== 2004 'AUS' 2000 'NED' 1994 'FRA' 3 rows selected. </pre>
<pre> ALTER TABLE isol4_tbl ADD COLUMN gold INT; ;xr /* unable to alter the table schema until tran 2 committed */ </pre>	<pre> /* repeatable read is ensured while tran_1 is altering table schema */ SELECT * FROM isol4_tbl; ;xr === <Result of SELECT Command> === host_year nation_code ===== 2004 'AUS' 2000 'NED' 1994 'FRA' 3 rows selected. </pre>
<pre> 1 command(s) successfully processed. </pre>	<pre> COMMIT; ;xr </pre>
	<pre> SELECT * FROM isol4_tbl; ;xr /* unable to access the table until tran_1 committed */ </pre>
<pre> COMMIT; </pre>	<pre> === <Result of SELECT Command > === </pre>

<code>;xr</code>	host_year	nation_code	gold
	2004	'AUS'	NULL
	2000	'NED'	NULL
	1994	'FRA'	NULL
	3 rows selected.		

REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES

CUBRID에서 디폴트 설정된 격리 수준(3)으로서 동시성이 높다. 행에 대해서는 더티 읽기, 반복 불가능한 읽기, 유령 읽기가 발생할 수 있지만, 테이블에 대해서는 반복 가능한 읽기를 보장한다. 즉, 트랜잭션 T1이 하나의 객체를 갱신하는 동안 다른 트랜잭션 T2에서 객체를 읽을 수 있다는 의미이다.

다음과 같은 규칙이 적용된다.

- 트랜잭션 T1은 다른 트랜잭션 T2에서 갱신 중인 레코드를 읽을 수 있다.
- 트랜잭션 T1은 다른 트랜잭션 T2에서 조회 중인 테이블에 대해 레코드를 갱신/삽입할 수 있다.
- 트랜잭션 T1은 다른 트랜잭션 T2에서 조회 중인 테이블의 스키마를 변경할 수 없다.

이 격리 수준은 배타 잠금에 대해서는 2단계 잠금을 따른다. 하지만 행에 대한 공유 잠금은 행이 조회된 직후 바로 해제되고, 행에 대한 갱신 잠금도 갱신을 수행한 직후 바로 해제된다. 한편, 테이블에 대한 의도 잠금은 반복 가능한 읽기를 보장하기 위하여 트랜잭션이 종료될 때 해제된다.

예제

다음은 동시에 수행되는 트랜잭션의 격리 수준이 **REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES**인 경우 한 트랜잭션에서 커밋하지 않은 더티 데이터를 다른 트랜잭션에서 읽을 수 있는 한편, 테이블 스키마 갱신에 대해서는 반복 가능한 읽기를 보장함을 보여주는 예제이다.

session 1	session 2
<pre> ;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 3 ;xr Isolation level set to: REPEATABLE READ SCHEMA, READ UNCOMMITTED INSTANCES. </pre>	<pre> ;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 3 ;xr Isolation level set to: REPEATABLE READ SCHEMA, READ UNCOMMITTED INSTANCES. </pre>
<pre> --creating a table CREATE TABLE isol3 tbl(host year integer, nation_code char(3)); CREATE UNIQUE INDEX on isol3_tbl(nation_code, host_year); INSERT INTO isol3 tbl VALUES (2008, 'AUS'); COMMIT; ;xr </pre>	

	<pre>--selecting records from the table SELECT * FROM isol3_tbl; ;xr === <Result of SELECT Command> === host_year nation_code ===== 2008 'AUS' 1 rows selected.</pre>
<pre>INSERT INTO isol3_tbl VALUES (2004, 'AUS'); INSERT INTO isol3_tbl VALUES (2000, 'NED'); ;xr 2 rows affected. /* able to insert new rows even if tran 2 uncommitted */</pre>	
	<pre>SELECT * FROM isol3_tbl; ;xr === <Result of SELECT Command> === host_year nation_code ===== 2008 'AUS' 2004 'AUS' 2000 'NED' 3 rows selected. /* dirty read may occur so that tran_2 can select new rows uncommitted by tran_1 */</pre>
<pre>ROLLBACK; ;xr</pre>	
	<pre>SELECT * FROM isol3_tbl; ;xr === <Result of SELECT Command> === host_year nation_code ===== 2008 'AUS' 1 rows selected. /* unrepeatable read may occur so that selected results are different */</pre>
<pre>INSERT INTO isol3_tbl VALUES (1994, 'FRA'); ;xr DELETE FROM isol3_tbl WHERE nation_code = 'AUS' and host_year=2008; ;xr</pre>	

1 rows affected. 1 rows affected. /* able to delete rows even if tran 2 uncommitted */	
	<pre>SELECT * FROM isol3_tbl; ;xr === <Result of SELECT Command> === host_year nation_code ===== 1994 'FRA' 1 rows selected.</pre>
<pre>ALTER TABLE isol3_tbl ADD COLUMN gold INT; ;xr /* unable to alter the table schema until tran 2 committed */</pre>	
	<pre>/* repeatable read is ensured while tran_1 is altering table schema */ SELECT * FROM isol3_tbl; ;xr === <Result of SELECT Command> === host_year nation_code ===== 1994 'FRA' 1 rows selected.</pre>
1 command(s) successfully processed.	<pre>COMMIT; ;xr</pre>
	<pre>SELECT * FROM isol3_tbl; ;xr</pre>
<pre>COMMIT; ;xr</pre>	<pre>=== <Result of SELECT Command > === host_year nation_code gold ===== 1994 'FRA' NULL 1 rows selected.</pre>

참고 CUBRID는 다양한 상황에서 워크 스페이스에 있는 더티 데이터를 데이터베이스로 내려쓰기(flush)한다. 이에 관한 설명은 [CUBRID에서 더티 인스턴스\(dirty instance\)를 다루는 방법](#)을 참고한다.

READ COMMITTED CLASS with READ COMMITTED INSTANCES

비교적 낮은 격리 수준(2)으로서 더티 읽기는 발생하지 않지만, 반복 불가능한 읽기와 유령 읽기는 발생할 수 있다. 즉, 위에 설명한 **REPEATABLE READ CLASS with READ COMMITTED INSTANCES**(수준 4)와 유사하지만, 테이블 스키마에 대해서는 다르게 동작한다. 트랜잭션 T1이 조회 중인 테이블에 대해 다른

트랜잭션 T2가 스키마를 변경할 수 있으므로, 테이블 스키마 갱신에 의한 반복 불가능한 읽기가 발생할 수 있다.

다음과 같은 규칙이 적용된다.

- 트랜잭션 T1은 다른 트랜잭션 T2에서 갱신 중인 레코드를 읽을 수 없다.
- 트랜잭션 T1은 다른 트랜잭션 T2에서 조회 중인 테이블로 레코드를 갱신/삽입할 수 있다.
- 트랜잭션 T1은 다른 트랜잭션 T2에서 조회 중인 테이블의 스키마를 변경할 수 있다.

이 격리 수준은 배타 잠금에 대해서는 2단계 잠금을 따른다. 하지만 행에 대한 공유 잠금은 행이 조회된 직후 바로 해제되고, 테이블에 대한 의도 잠금도 바로 해제되므로 반복 불가능한 읽기가 발생할 수 있다.

예제

다음은 동시에 수행되는 트랜잭션의 격리 수준이 **READ COMMITTED CLASS** with **READ COMMITTED INSTANCES**인 경우 한 트랜잭션에서 객체 읽기를 수행하는 동안 다른 트랜잭션이 새로운 레코드를 추가 또는 갱신할 수 있으므로 유령 읽기 및 반복 불가능한 읽기가 발생할 수 있고, 테이블 스키마에 대해서도 반복 불가능한 읽기가 발생할 수 있음을 보여주는 예제이다.

session 1	session 2
<pre> ;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 2 ;xr Isolation level set to: READ COMMITTED SCHEMA, READ COMMITTED INSTANCES. </pre>	<pre> ;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 2 ;xr Isolation level set to: READ COMMITTED SCHEMA, READ COMMITTED INSTANCES. </pre>
<pre> --creating a table CREATE TABLE isol2_tbl(host_year integer, nation_code char(3)); CREATE UNIQUE INDEX on isol2_tbl(nation_code, host_year); INSERT INTO isol2 tbl VALUES (2008, 'AUS'); COMMIT; ;xr </pre>	<pre> --selecting records from the table SELECT * FROM isol2_tbl; ;xr === <Result of SELECT Command> === host_year nation_code ===== 2008 'AUS' 1 rows selected. </pre>
<pre> INSERT INTO isol2 tbl VALUES (2004, 'AUS'); </pre>	

<pre> INSERT INTO isol2_tbl VALUES (2000, 'NED'); ;xr 2 rows affected. /* able to insert new rows even if tran 2 uncommitted */ </pre>	
	<pre> SELECT * FROM isol2_tbl; ;xr /* phantom read may occur when tran 1 committed */ </pre>
<pre> COMMIT; ;xr </pre>	<pre> === <Result of SELECT Command> === host_year nation_code ===== 2008 'AUS' 2004 'AUS' 2000 'NED' 3 rows selected. </pre>
<pre> INSERT INTO isol2_tbl VALUES (1994, 'FRA'); ;xr 1 rows affected. </pre>	
	<pre> SELECT * FROM isol2_tbl; ;xr /* unrepeatable read may occur when tran 1 committed */ </pre>
<pre> DELETE FROM isol2_tbl WHERE nation_code = 'AUS' and host_year=2008; ;xr 1 rows affected. /* able to delete rows even if tran 2 uncommitted */ </pre>	
<pre> COMMIT; ;xr </pre>	<pre> === <Result of SELECT Command> === host_year nation_code ===== 2004 'AUS' 2000 'NED' 1994 'FRA' 3 rows selected. </pre>
<pre> ALTER TABLE isol2_tbl ADD COLUMN gold INT; ;xr 1 command(s) successfully processed. /* able to alter the table schema even if tran 2 is uncommitted yet*/ </pre>	
	<pre> /* unrepeatable read may occur so that result shows different schema */ </pre>

	SELECT * FROM isol2_tbl; ;xr
COMMIT; ;xr	=== <Result of SELECT Command > === host_year nation_code gold =====
	2004 'AUS' NULL
	2000 'NED' NULL
	1994 'FRA' NULL
	3 rows selected.

READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES

가장 낮은 격리 수준(1)으로서 동시성이 가장 높다. 행에 대해서는 더티 읽기, 반복 불가능한 읽기, 유령 읽기가 발생할 수 있고, 테이블에 대해서도 반복 불가능한 읽기가 발생할 수 있다. 위에서 설명한 **REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES**(수준 3)와 유사하지만, 테이블 스키마에 대해서는 다르게 동작한다. 즉, 트랜잭션 T1이 조회 중인 테이블에 대해 다른 트랜잭션 T2가 스키마를 변경할 수 있으므로, 테이블 스키마 갱신에 의한 반복 불가능한 읽기가 발생할 수 있다.

다음과 같은 규칙이 적용된다.

- 트랜잭션 T1은 다른 트랜잭션 T2에서 갱신 중인 레코드를 읽을 수 있다.
- 트랜잭션 T1은 다른 트랜잭션 T2에서 조회 중인 테이블에 대해 레코드를 갱신/삽입할 수 있다.
- 트랜잭션 T1은 다른 트랜잭션 T2에서 조회 중인 테이블의 스키마를 변경할 수 있다.

이 격리 수준은 배타 잠금에 대해서는 2단계 잠금을 따른다. 하지만 행에 대한 공유 잠금은 행이 조회된 직후 바로 해제되고, 행에 대한 갱신 잠금도 갱신을 수행한 직후 바로 해제된다. 또한, 테이블에 대한 의도 잠금도 조회 직후 바로 해제된다.

예제

session 1	session 2
<pre> ;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 1 ;xr Isolation level set to: READ COMMITTED SCHEMA, READ UNCOMMITTED INSTANCES. </pre>	<pre> ;autocommit off AUTOCOMMIT IS OFF SET TRANSACTION ISOLATION LEVEL 1 ;xr Isolation level set to: READ COMMITTED SCHEMA, READ UNCOMMITTED INSTANCES. </pre>
<pre> --creating a table CREATE TABLE isol1_tbl(host_year integer, nation_code char(3)); CREATE UNIQUE INDEX on isol1_tbl(nation_code, host_year); INSERT INTO isol1 tbl VALUES (2008, 'AUS'); COMMIT; </pre>	

;xr	<pre>--selecting records from the table SELECT * FROM isoll_tbl; ;xr === <Result of SELECT Command> === host_year nation_code ===== 2008 'AUS' 1 rows selected.</pre>
<pre>INSERT INTO isoll_tbl VALUES (2004, 'AUS'); INSERT INTO isoll_tbl VALUES (2000, 'NED'); ;xr 2 rows affected. /* able to insert new rows even if tran 2 uncommitted */</pre>	<pre>SELECT * FROM isoll_tbl; ;xr === <Result of SELECT Command> === host_year nation_code ===== 2008 'AUS' 2004 'AUS' 2000 'NED' 3 rows selected. /* dirty read may occur so that tran 2 can select new rows uncommitted by tran_1 */</pre>
<pre>ROLLBACK; ;xr</pre>	<pre>SELECT * FROM isoll_tbl; ;xr === <Result of SELECT Command> === host_year nation_code ===== 2008 'AUS' 1 rows selected. /* unrepeatable read may occur so that selected results are different */</pre>
<pre>INSERT INTO isoll_tbl VALUES (1994, 'FRA'); ;xr DELETE FROM isoll_tbl WHERE nation_code = 'AUS' and host_year=2008;</pre>	

<pre> ;xr 1 rows affected. 1 rows affected. /* able to delete rows while tran 2 is selecting rows*/ </pre>	
	<pre> SELECT * FROM isoll_tbl; ;xr === <Result of SELECT Command> === host_year nation_code ===== 1994 'FRA' ===== 1 rows selected. </pre>
<pre> ALTER TABLE isoll_tbl ADD COLUMN gold INT; ;xr 1 command(s) successfully processed. /* able to alter the table schema even if tran 2 is uncommitted yet*/ </pre>	
	<pre> /* unrepeatable read may occur so that result shows different schema */ SELECT * FROM isoll_tbl; ;xr === <Result of SELECT Command > === host_year nation_code gold ===== 1994 'FRA' NULL ===== 1 rows selected. </pre>
<pre> COMMIT; ;xr </pre>	

UPDATE INCONSISTENCY

이 격리 수준에서는 커밋되지 않은 갱신을 잃어버릴 수 있다. 이것은 트랜잭션이 끝나기 전에 데이터를 커밋하기 때문에 트랜잭션을 복구 불가능하게(트랜잭션을 롤백할 수 없을 수 있음) 만든다. CUBRID는 이 격리 수준을 지원하지 않는데, 그 이유는 사용자의 갱신을 잃어버릴 수 있기 때문이다. 하지만 명시되면 CUBRID는 사용자의 응용에 적당한 격리 수준을 제공한다.

다음은 이 격리 수준의 규칙이다.

- 트랜잭션은 다른 트랜잭션에서 수정 중인 객체를 덮어쓰지 않는다.

참고 지원되는 모든 격리 수준에 대해 트랜잭션은 복구가 가능한데, 그 이유는 트랜잭션이 끝날 때까지 갱신을 커밋하지 않기 때문이다.

지원하지 않는 격리 수준 조합

SET TRANSACTION ISOLATION LEVEL 구문을 이용하여 위에서 설명한 격리 수준 이외의 다른 조합을 정의할 수는 있으나, 아래의 조합은 허용하지 않는다. 이 경우 시스템은 경고 메시지를 출력하고, 명시된 것에 가까운 격리 수준을 선택한다.

아래는 지원하지 않는 격리 수준이다. 한 트랜잭션이 데이터 조회하는 중에 테이블 스키마 변경이 발생하면, 행에 대해서는 반복 불가능한 읽기가 발생하게 되므로 아래와 같은 조합은 허용되지 않는다.

- **READ COMMITTED CLASS with REPEATABLE READ INSTANCES**
- **READ UNCOMMITTED CLASS with REPEATABLE READ INSTANCES**

또한, 아래와 같은 격리 수준도 지원하지 않는다. 한 트랜잭션에서 테이블 스키마를 변경하는 중에 다른 트랜잭션에서 행을 갱신하는 것은 허용되지 않는다.

- **READ UNCOMMITTED CLASS with READ COMMITTED INSTANCES**
- **READ UNCOMMITTED CLASS with READ UNCOMMITTED INSTANCES**

CUBRID 에서 더티 인스턴스(dirty instances)를 다루는 방법

CUBRID는 다양한 상황에서 클라이언트의 버퍼에 존재하는 더티 데이터(또는 더티 인스턴스)를 데이터베이스 서버로 내려쓰기(flush)한다. 아래에 명시된 상황이 아닌 경우에도 내려쓰기가 발생할 수 있다.

- 트랜잭션 커밋이 수행될 때 더티 데이터는 서버로 내려쓰기된다.
- 클라이언트의 버퍼에 적재된 데이터가 많은 경우, 일부 더티 데이터는 서버로 내려쓰기된다.
- 테이블 A의 더티 데이터는 테이블 A의 스키마가 갱신될 때 서버로 내려쓰기된다.
- 테이블 A의 더티 데이터는 테이블 A가 조회(SELECT)될 때 서버로 내려쓰기된다.
- 더티 데이터의 일부는 서버 함수가 호출될 때 내려쓰기될 수 있다.

트랜잭션 종료와 복구

개요

CUBRID에서 복구 프로세스를 사용하면 소프트웨어 또는 하드웨어에 오류가 발생하더라도 데이터베이스에는 영향을 미치지 않도록 할 수 있다. CUBRID에서 모든 읽기와 갱신 명령문은 원자성(atomic)을 보장한다. 이것은 명령문들이 커밋되어 데이터베이스가 갱신되거나, 커밋되지 않아 갱신이 무효화되어야 함을 의미한다. 원자성의 개념은 트랜잭션을 구성하는 연산의 집합으로 확장된다. 트랜잭션은 커밋을 성공하여 모든 영향이 데이터베이스에 영구화 되거나 아니면 롤백되어 트랜잭션의 모든 영향이 제거되어야 한다. 트랜잭션의 원자성을 보장하기 위해서 CUBRID는 모든 트랜잭션의 갱신이 디스크에 쓰여지지 않은 채 오류가 발생할 때마다 커밋된 트랜잭션의 영향을 다시 적용시킨다. 또한 CUBRID는 사이트가 실패(몇몇 트랜잭션이 커밋되지 못했거나 응용 프로그램이 트랜잭션 취소를 요청했을 수 있다)할 때마다 데이터베이스에서 부분적으로 커밋된 트랜잭션의 영향을 제거한다. 이러한 복구 기능은 응용

프로그램이 시스템 오류에 따라 어떻게 데이터베이스를 일관성 있는 상태로 되돌릴 지에 대한 부담을 덜어준다. CUBRID에서 사용되는 복구 기능은 언두/리두 로깅 기법을 기반으로 한다.

CUBRID는 하드웨어와 소프트웨어 오류가 발생하는 동안 트랜잭션의 원자성을 유지하기 위해서 자동 복구 기법을 제공한다. CUBRID의 복구 기능은 응용 프로그램 또는 컴퓨터 시스템의 오류가 발생하더라도 데이터베이스를 항상 일관된 상태로 되돌려놓기 때문에 사용자는 복구에 대한 책임을 가질 필요가 없다. 이것을 위해 CUBRID는 시스템이나 응용 프로그램의 실패 또는 사용자의 명시적 요청에 따라 커밋된 트랜잭션의 일부를 자동적으로 롤백한다. 예를 들어 **COMMIT WORK** 문이 수행되는 동안 발생한 시스템 오류는 트랜잭션이 아직 커밋되지 않았다면(사용자의 연산이 커밋되었다는 확인을 받지 못한다) 중단해야 할 것이다. 자동 중단은 커밋되지 않은 갱신을 취소함으로써 데이터베이스에 원하지 않은 변경을 야기하는 오류를 방지한다.

데이터베이스 재구동

CUBRID는 시스템과 매체(디스크)에 오류가 발생했을 때 커밋되었거나 커밋되지 않은 트랜잭션을 복구하기 위해 로그 볼륨/파일과 데이터베이스 백업을 이용한다. 로그는 사용자가 지정한 롤백을 지원하는데도 사용된다. 로그는 CUBRID가 생성한 순차적인 파일의 모음으로 구성된다. 가장 최근의 로그를 활성 로그(active log)라고 부르며, 나머지 로그를 보관 로그(archive log)라고 부른다. 로그 파일은 활성 로그와 보관 로그 전체를 가리키는데 사용된다.

데이터베이스에 대한 모든 갱신은 로그에 기록된다. 실제로 갱신에 대한 2개의 복사본이 기록되는데, 첫 번째 복사본은 before 이미지라고 불리며 사용자가 명시한 **ROLLBACK WORK** 문이 수행되는 동안이나 매체 또는 시스템에 오류가 발생했을 때 데이터를 복원하는데 사용된다. 두 번째 복사본은 after 이미지인데 매체 또는 시스템에 오류가 발생했을 때 갱신을 다시 적용시키는데 사용된다.

CUBRID는 활성 로그가 꽉 차면 보관 로그로 복사하여 디스크에 보존한다. 보관 로그는 시스템 장애가 발생했을 때 데이터베이스 복구를 위해 필요하다. 시스템 장애에 대한 복구가 필요하지 않으면 보관 로그를 유지할 필요는 없다. 이것은 **media_failure_support** 시스템 매개 변수로 지정할 수 있다. 이 매개 변수에 대한 설명은 [로깅 관련 파라미터](#)를 참조한다.

정상적인 종료 또는 오류

데이터베이스가 정상적인 종료나 장비의 오류로 다시 시작되면 CUBRID는 자동적으로 데이터베이스를 복구한다. 복구 프로세스는 데이터베이스에 빠져있는 커밋된 변화를 다시 적용하고 데이터베이스에 저장되어 있는 커밋되지 않은 변경을 제거한다. 데이터베이스 시스템의 일반적인 연산은 복구가 끝나고 난 후 재개된다. 이러한 복구 프로세스는 어떠한 보관 로그나 데이터베이스 백업도 사용하지 않는다.

데이터베이스는 client/server 환경에서는 서버 유틸리티를 이용해 재구동할 수 있다.

매체 오류

매체에 오류가 발생 한 후에 데이터베이스를 다시 구동시키는 데는 사용자의 개입이 다소 필요하다. 첫 번째 단계는 좋은 상태로 알려진 데이터베이스의 백업을 설치하여 데이터베이스를 복원하는 것이다. CUBRID에서 가장 최근의 로그 파일(마지막 백업 이후의 것)을 설치하는 것을 필요로 한다. 이 특정 로그(보관, 활성화)는 데이터베이스의 백업 복사본에 적용된다. 복원이 커밋된 후 데이터베이스는 일반적인 종료의 경우와 마찬가지로 재구동할 수 있다.

데이터베이스를 특정 주기로 백업하는 것이 중요하다. 백업을 하는 주기는 데이터베이스의 갱신이 일어나는 수준에 따라 다르다. 데이터베이스의 백업이 만들어지면 CUBRID는 현재 데이터베이스 백업을 이용하여 데이터베이스를 복원하는데 더 이상 필요 없는 보관 로그를 지정한다. 그러나 CUBRID는 이 보관 로그를 삭제하지는 않는다. 데이터베이스 관리자는 데이터베이스 백업과 보관 로그를 삭제하려고 할 때는 세밀한 주의를 기울여야 한다. 어떤 경우에는 가장 최근의 데이터베이스 백업이 실패할 수도 있다.

참고 데이터베이스의 정보를 잃어버리는 가능성을 줄이기 위해서 보관 로그가 디스크에서 삭제되기 전에 보관 로그의 스냅샷을 만들고 테이프에 백업할 것을 추천한다. DBA는 **cubrid backupdb**, **cubrid restoredb** 유틸리티를 사용하여 데이터베이스를 백업하고 복원할 수 있다. 이 유틸리티에 대한 상세한 내용을 보려면 [데이터베이스 백업](#)과 [데이터베이스 복구](#)를 참조한다.

데이터베이스 사용자 권한

데이터베이스 사용자

CUBRID는 기본적으로 **DBA**와 **PUBLIC** 두 종류의 기본적인 사용자를 제공한다.

- 모든 사용자는 **PUBLIC** 사용자에게 부여된 권한을 소유한다. 데이터베이스의 모든 사용자는 **PUBLIC**의 멤버가 된다. 사용자에게 권한을 부여하는 방법은 **PUBLIC** 사용자에게 대한 권한을 부여하는 것이다.
- **DBA**는 데이터베이스 관리자를 위한 권한을 소유한다. **DBA**는 자동으로 모든 사용자와 그룹의 멤버가 된다. 즉, **DBA**는 모든 테이블에 대한 접근 권한을 갖는다. 따라서 **DBA**와 **DBA**의 멤버에게 명시적으로 권한을 부여할 필요는 없다. 데이터베이스 사용자는 고유한 이름을 갖는다. 데이터베이스 관리자는 **cubrid createdb** 유틸리티를 이용하여 일괄적으로 사용자를 생성할 수 있다(자세한 내용은 [데이터베이스 관리](#) 참조). 데이터베이스 사용자는 동일한 권한을 갖는 멤버를 소유할 수 없다. 사용자에게 권한이 부여되면 해당 사용자의 모든 멤버는 자동으로 동일한 권한을 소유한다.

사용자 관리

설명

DBA와 **DBA**의 멤버는 SQL 문을 사용하여 사용자를 생성, 변경, 삭제할 수 있다.

구문

```
CREATE USER user_name
[ PASSWORD password ]
[ GROUPS user_name [ {, user_name } ... ] ]
[ MEMBERS user_name [ {, user_name } ... ] ] ;
DROP USER user_name;
ALTER USER user_name PASSWORD password;
```

- *user_name*: 생성, 삭제, 변경할 사용자 이름을 지정한다.
- *password*: 생성 혹은 변경할 사용자의 비밀번호를 지정한다.

예제 1

다음은 사용자 Fred를 생성하고 비밀번호를 변경한 후에 Fred를 삭제하는 예제이다.

```
CREATE USER Fred;
ALTER USER Fred PASSWORD '1234';
DROP USER Fred;
```

예제 2

다음은 사용자를 생성하고 생성된 사용자에게 멤버를 추가하는 예제이다. 다음 문장을 통해 company는 engineering, marketing, design을 멤버로 가지는 그룹이 된다. marketing은 smith, jones를, design은 smith를, engineering은 brown을 멤버로 가지는 그룹이 된다.

```
CREATE USER company;
CREATE USER engineering GROUPS company;
CREATE USER marketing GROUPS company;
```

```
CREATE USER design GROUPS company;
CREATE USER smith GROUPS design, marketing;
CREATE USER jones GROUPS marketing;
CREATE USER brown GROUPS engineering;
```

예제 3

다음은 위와 동일한 그룹을 생성하는 예이지만 **GROUPS** 대신 **MEMBERS** 문을 사용하는 예제이다.

```
CREATE USER smith;
CREATE USER brown;
CREATE USER jones;
CREATE USER engineering MEMBERS brown;
CREATE USER marketing MEMBERS smith, jones;
CREATE USER design MEMBERS smith;
CREATE USER company MEMBERS engineering, marketing, design;
```

권한 부여

설명

CUBRID에서 권한 부여의 최소 단위는 테이블이다. 자신이 만든 테이블에 다른 사용자(그룹)의 접근을 허용하려면 해당 사용자(그룹)에게 적절한 권한을 부여해야 한다.

권한이 부여된 그룹에 속한 모든 멤버는 같은 권한을 소유하므로 모든 멤버에게 개별적으로 권한을 부여할 필요는 없다. **PUBLIC** 사용자가 생성한 (가상) 테이블은 다른 모든 사용자에게 접근이 허용된다. **GRANT** 문을 사용하여 사용자에게 접근 권한을 부여할 수 있다.

구문

```
GRANT operation [ ( { ,operation } ) ] ON table name [ ( { ,table name } ) ]
TO user [ ( { ,user } ) ] [ WITH GRANT OPTION ] [ ; ]
```

- *operation* : 권한을 부여할 때 사용 가능한 연산을 나타낸다.
- **SELECT** : 테이블 정의 내용을 읽을 수 있고 인스턴스 조회가 가능. 가장 일반적인 유형의 권한.
- **INSERT** : 테이블의 인스턴스를 생성할 수 있는 권한.
- **UPDATE** : 테이블에 이미 존재하는 인스턴스를 수정할 수 있는 권한.
- **DELETE** : 테이블의 인스턴스를 삭제할 수 있는 권한.
- **ALTER** : 테이블의 정의를 수정할 수 있고, 테이블의 이름을 변경하거나 삭제할 수 있는 권한.
- **INDEX** : 검색 속도의 향상을 위해 컬럼에 인덱스를 생성할 수 있는 권한.
- **EXECUTE** : 테이블 메소드 혹은 인스턴스 메소드를 호출할 수 있는 권한.
- **ALL PRIVILEGES** : 앞서 설명한 7가지 권한을 모두 포함.
- *table_name* : 권한을 부여할 테이블 혹은 뷰의 이름을 지정한다.
- *user* : 권한을 부여할 사용자나 그룹의 이름을 지정한다. 데이터베이스 사용자의 로그인 이름을 입력하거나 시스템 정의 사용자인 **PUBLIC**을 입력할 수 있다. **PUBLIC**이 명시되면 데이터베이스의 모든 사용자는 부여한 권한을 가진다.
- **WITH GRANT OPTION** : **WITH GRANT OPTION**을 이용하면 권한을 부여받은 사용자가 부여받은 권한을 또다른 사용자에게 부여할 수 있다.

예제 1

다음은 Fred(Fred의 모든 멤버)에게 olympic 테이블의 검색 권한을 부여한 예제이다.

```
GRANT SELECT ON olympic TO Fred;
```

예제 2

다음은 Jeniffer와 Daniel(두 사용자에게 속한 모든 멤버)에게 nation과 athlete 테이블에 대해 **SELECT, INSERT, UPDATE, DELETE** 권한을 부여한 예제이다.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON nation, athlete TO Jeniffer, Daniel;
```

예제 3

다음은 모든 사용자에게 game, event 테이블에 대해 모든 권한을 부여한 예제이다.

```
GRANT ALL PRIVILEGES ON game, event TO public;
```

예제 4

다음 **GRANT** 문은 Ross에게 record, history 테이블에 대한 검색 권한을 부여하고 Ross가 다른 사용자에게 검색 권한을 부여하는 것을 허용하도록 **WITH GRANT OPTION**을 사용한 예제이다. 이후 Ross는 다른 사용자에게 자신이 받은 권한 내에서 권한을 부여할 수 있다.

```
GRANT SELECT ON record, history TO Ross WITH GRANT OPTION;
```

주의 사항

- 권한을 부여하는 사용자는 권한 부여 전에 나열된 모든 테이블의 소유자이거나, **WITH GRANT OPTION**을 가지고 있어야 한다.
- 뷰에 대한 **SELECT, UPDATE, DELETE, INSERT** 연산을 부여하기 전에 뷰의 소유자는 뷰의 질의 명세부에 포함된 질의에 포함되어 있는 모든 테이블에 대해서 **SELECT** 권한과 부여할 권한을 가져야 한다. **DBA** 사용자와 **DBA** 그룹에 속한 멤버는 자동적으로 모든 테이블에 대한 모든 권한을 가진다.

권한 해지

설명

REVOKE 문을 사용하여 권한을 해지할 수 있다. 사용자에게 부여된 권한은 언제든지 해지가 가능하다. 한 사용자에게 두 종류 이상의 권한을 부여했다면 권한 중 일부 또는 전부를 해지할 수 있다. 또한 하나의 **GRANT** 문으로 여러 사용자에게 여러 테이블에 대한 권한을 부여한 경우라도 일부 사용자와 일부 테이블에 대해 선택적인 권한 해지가 가능하다.

권한을 부여한 사용자에게서 권한(**WITH GRANT OPTION**)을 해지하면, 권한을 해지당한 사용자로부터 권한을 받은 사용자도 권한을 해지당한다.

구문

```
REVOKE operation [ { , operation } _ ] ON table_name [ { , class_name } _ ]
```

```
FROM user [ { , user } _ ] [ ; ]
```

- *operation* : 권한을 부여할 때 부여할 수 있는 연산의 종류이다(자세한 내용은 [권한 부여](#) 참조).
- *table_name* : 권한을 부여할 테이블 혹은 뷰의 이름을 지정한다.
- *user* : 권한을 부여할 사용자나 그룹의 이름을 지정한다.

예제 1

다음은 Fred, John 사용자에게 nation, athlete 두 테이블에 대해 **SELECT, INSERT, UPDATE, DELETE** 권한을 부여하는 예제이다.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON nation, athlete TO Fred, John;
```

예제 2

다음은 예제 1에서 Fred에게 부여된 모든 권한은 남겨두고, John에게는 조회 권한만을 부여하기 위해 **REVOKE** 문장을 수행하는 예제이다. 만약 John이 다른 사용자에게 권한을 부여했다면 권한받은 사용자 또한 조회만 가능하다.

```
REVOKE INSERT, UPDATE, DELETE ON nation, athlete FROM John;
```

예제 3

다음은 예제 1에서 Fred에게 부여한 모든 권한을 해지하기 위해 **REVOKE** 문을 수행하는 예제이다. 이 문장이 수행되면 Fred는 nation, athlete 테이블에 대한 어떠한 연산도 허용되지 않는다.

```
REVOKE ALL PRIVILEGES ON nation, athlete FROM Fred;
```

사용자 권한 관리 메소드

설명

데이터베이스 관리자(DBA)는 데이터베이스 사용자에 대한 정보를 저장하는 **db_user** 또는 시스템 권한 클래스인 **db_authorizations**에 정의된 권한 관련 메소드들을 호출하여 사용자 권한을 조회 및 수정할 수 있다. 호출하고자 하는 메소드에 따라 **db_user** 또는 **db_authorizations** 클래스를 명시할 수 있으며, 메소드의 리턴 값을 변수에 저장할 수 있다. 또한, 일부 메소드는 **DBA**와 **DBA** 그룹의 멤버에 의해서만 호출될 수 있음을 유의한다.

구문

```
CALL method_definition ON CLASS auth class [ TO variable ] [ ; ]  
CALL method_definition ON variable [ ; ]
```

login() 메소드

login() 메소드는 **db_user** 클래스의 클래스 메소드로서, 현재 데이터베이스에 접속한 사용자를 변경하고자 할 때 사용된다. 새로 접속할 사용자 이름과 비밀번호가 인자로 주어져서, 스트링 타입이어야 한다. 비밀번호가 없는 경우 인자에 공백 스트링("")을 입력할 수 있다. **DBA**나 **DBA** 그룹의 멤버는 비밀번호를 입력하지 않고 **login()** 메소드를 호출할 수 있다.

```
-- 비밀번호가 없는 DBA 사용자로 접속하기
CALL login ('dba', '') ON CLASS db_user;
-- 비밀번호가 cubrid인 user_1 사용자로 접속하기
CALL login ('user_1', 'cubrid') ON CLASS db_user;
```

add_user() 메소드

add_user() 메소드는 **db_user** 클래스의 클래스 메소드로서, 새로운 사용자를 추가할 때 사용된다. 새로 추가할 사용자 이름과 비밀번호가 인자로 주어지며, 스트링 타입이어야 한다. 이때, 추가할 사용자 이름은 이미 등록된 데이터베이스 사용자 이름과 중복되어서는 안 된다. 한편, **add_user()** 메소드는 **DBA** 사용자와 **DBA** 그룹에 속한 멤버만 호출할 수 있다.

```
-- 비밀번호가 없는 user_2 추가하기
CALL add_user ('user_2', '') ON CLASS db_user;
-- 비밀번호가 없는 user_3 추가하고, 메소드 리턴 값을 admin 변수에 저장하기
CALL add_user ('user_2', '') ON CLASS db_user to admin;
```

drop_user() 메소드

drop_user() 메소드는 **db_user** 클래스의 클래스 메소드로서, 기존 사용자를 삭제할 때 사용된다. 삭제하고자 하는 사용자 이름만 인자로 주어지며, 스트링 타입이어야 한다. 이때, 클래스의 소유자는 삭제할 수 없으므로, **DBA**는 관련 클래스의 소유자를 변경한 후, 해당 사용자를 삭제할 수 있다. **drop_user()** 메소드 역시 **DBA** 사용자와 **DBA** 그룹에 속한 멤버만 호출할 수 있다.

```
-- user_2 삭제하기
CALL drop_user ('user_2') ON CLASS db_user;
```

find_user() 메소드

find_user() 메소드는 **db_user** 클래스의 클래스 메소드로서, 인자로 주어진 사용자를 검색할 때 사용된다. 찾고자 하는 사용자 이름이 인자로 주어지며, **TO** 뒤에 지정된 변수에 메소드의 리턴 값을 저장하여 다음 질의 수행 시 변수에 저장된 값을 이용할 수 있다.

```
-- user_2를 찾아서 admin이라는 변수에 저장하기
CALL find_user ('user_2') ON CLASS db_user TO admin;
```

set_password() 메소드

set_password() 메소드는 사용자 인스턴스 각각에 대해 호출할 수 있는 인스턴스 메소드로서, 사용자의 비밀번호를 변경할 때 사용된다. 지정된 사용자의 새로운 비밀번호가 인자로 주어진다. **DBA**와 **DBA** 그룹의 멤버를 제외한 일반 사용자는 자신의 비밀번호만 변경할 수 있다.

```
-- user_4 를 추가하고 user_common 변수에 저장하기
CALL add_user ('user_4', '') ON CLASS db_user to user_common;
-- user_4의 비밀번호를 'abcdef'로 변경하기
CALL set_password('abcdef') on user_common;
```

change_owner() 메소드

change_owner() 메소드는 **db_authorizations** 클래스의 클래스 메소드로서, 클래스 소유자를 변경할 때 사용된다. 소유자를 변경하고자 하는 클래스 이름과 새로운 소유자의 이름이 각각 인자로 주어진다. 이때, 데이터베이스에 존재하는 클래스와 소유자가 인자로 지정되어야 하며, 그렇지 않은 경우 에러가 발생된다.

change_owner() 메소드는 **DBA**와 **DBA** 그룹의 멤버만 호출할 수 있다.

```
-- table_1의 소유자를 user_4로 변경하기
CALL change_owner ('table_1', 'user_4') ON CLASS db_authorizations;
```

예제

다음 예제는 특정 데이터베이스 사용자의 존재 여부를 판단하기 위해 시스템 클래스인 **db_user**에 등록된 메소드인 **find_user**를 호출하는 **CALL** 문의 수행을 보여준다. 첫 번째 문장은 **db_user** 클래스에 정의된 클래스 메소드를 호출한다. 찾고자 하는 대상 사용자가 데이터베이스에 등록되어 있을 경우 x에는 해당 클래스 이름(여기에서는 **db_user**)이 저장되고, 없을 경우엔 **NULL**이 저장된다.

두 번째 문장은 변수 x에 저장된 값을 출력하는 방법이다. 이 질의문에서 **DB_ROOT**는 시스템 클래스로서, 하나의 인스턴스만이 존재하여 **sys_date**나 등록된 변수의 값을 출력하는 데 사용할 수 있다. 이러한 용도로 쓰일 경우 **DB_ROOT**는 인스턴스가 하나인 다른 테이블로 대체할 수 있다.

```
CALL find_user('dba') ON CLASS db_user to x;
;xrun

=== <Result of CALL Command in Line 1> ===
Result
=====
db_user

SELECT x FROM db_root;
;xrun

=== <Result of SELECT Command in Line 1> ===
x
=====
db_user
```

find_user를 이용하면 결과값이 **NULL**인지 아닌지에 따라 해당 사용자가 데이터베이스에 존재하는지 여부를 판단할 수 있다.

질의 최적화

통계 정보 갱신

설명

UPDATE STATISTICS ON 문은 질의 처리기에서 사용되는 내부 통계 정보를 생성한다. 이러한 통계 정보는 데이터베이스 시스템이 질의를 처리하는데 효과적인 방법을 사용할 수 있게 한다.

구문

```
UPDATE STATISTICS ON { table_spec [ {, table_spec } ] | ALL CLASSES | CATALOG CLASSES }
[ ; ]
table_spec :
single table_spec
( single table_spec [ {, single table_spec } ] )
single table_spec :
[ ONLY ] table_name
| ALL table_name [ ( EXCEPT table_name ) ]
```

- **ALL CLASSES** : 키워드 **ALL CLASSES**를 지정하였을 경우 데이터베이스 안에 존재하는 모든 테이블에 대한 통계 정보가 갱신된다.

SQL 힌트 사용

설명

사용자는 질의문에 힌트를 주어 해당 질의 성능을 높일 수 있다. 질의 최적화기는 질의문에 대한 최적화 작업을 수행할 때 SQL 힌트를 참고하여 효율적인 실행 계획을 생성한다. CUBRID에서 지원하는 SQL 힌트는 테이블 조인 관련 힌트, 인덱스 관련 힌트, 통계 정보 관련 힌트가 있다.

구문

```
CREATE /** NO_STATS */ [TABLE | CLASS] ...;
ALTER /** NO_STATS */ [TABLE | CLASS] ...;

CREATE /** NO_STATS */ INDEX ...;
ALTER /** NO_STATS */ INDEX ...;
DROP /** NO_STATS */ INDEX ...;

SELECT /** hint [ { hint } ... ] */
SELECT --+ hint [ { hint } ... ]
SELECT /** hint [ { hint } ... ]

hint :
USE_NL[(spec-name[{, spec-name}...])]
USE_IDX[(spec-name[{, spec-name}...])]
USE_MERGE[(spec-name[{, spec-name}...])]
ORDERED
```

SQL 힌트는 더하기 기호(+)와 주석을 사용하여 지정한다. CUBRID는 이 주석을 공백에 의해 구분된 힌트의 리스트로 해석한다. 힌트 주석은 반드시 키워드 **SELECT**, **CREATE**, **ALTER** 등의 키워드 다음에 나타나야 하고, 더하기 기호(+)가 주석에서 첫 번째 문자로 시작되어야 한다.

- *hint*: 다음 힌트가 지정될 수 있다.
- **USE_NL**: 테이블 조인과 관련한 힌트로서, 질의 최적화기 중첩 루프 조인 실행 계획을 만든다.
- **USE_MERGE**: 테이블 조인과 관련한 힌트로서, 질의 최적화기는 정렬 병합 조인 실행 계획을 만든다.
- **ORDERED**: 테이블 조인과 관련한 힌트로서, 질의 최적화기는 **FROM** 절에 명시된 테이블의 순서대로 조인하는 실행 계획을 만든다. **FROM** 절에서 왼쪽 테이블은 조인의 외부 테이블이 되고, 오른쪽 테이블은 내부 테이블이 된다.
- **USE_IDX**: 인덱스 관련한 힌트로서, 질의 최적화기는 명시된 테이블에 대해 인덱스 조인 실행 계획을 만든다.
- **NO_STATS**: 통계 정보 관련 힌트로서, 질의 최적화기는 통계 정보를 갱신하지 않는다. 따라서, 해당 질의의 성능은 향상되나, 통계 정보를 갱신하지 않으므로 질의 계획이 최적화되지 않음에 유의한다.
- *spec_name*: *spec_name*이 **USE_NL**, **USE_IDX**, **USE_MERGE**와 함께 지정될 경우 해당 조인 방법은 *spec_name*에 대해서만 적용된다. 만약 **USE_NL**과 **USE_MERGE**가 함께 지정된 경우 주어진 힌트는 무시된다. 일부 경우에 질의 최적화기는 주어진 힌트에 따라 질의 실행 계획을 만들지 못할 수 있다. 예를 들어 오른쪽 외부 조인에 대해 **USE_NL**을 지정한 경우 이 질의는 내부적으로 왼쪽 외부 조인 질의로 변환이 되어 조인 순서는 보장되지 않을 수 있다.

예제 1

다음은 심권호 선수가 메달을 획득한 연도와 메달 종류를 구하는 예제이다. 단, **athlete** 테이블을 외부 테이블로 하고 **game** 테이블을 내부 테이블로 하는 중첩 루프 조인 실행 계획을 만들어야 한다. 다음과 같은 질의로 표현이 되는데, 질의최적화기는 **game** 테이블을 외부 테이블로 하고, **athlete** 테이블을 내부 테이블로 하는 중첩 루프 조인 실행 계획을 만든다.

```
SELECT /*+ USE_NL ORDERED */ a.name, b.host_year, b.medal
FROM athlete a, game b WHERE a.name = 'Sim Kwon Ho' AND a.code = b.athlete_code;
=== <Result of SELECT Command in Line 1> ===
      name                host year    medal
=====
'Sim Kwon Ho'              2000    'G'
'Sim Kwon Ho'              1996    'G'
2 rows selected.
```

예제 2

다음은 데이터가 없는 분할 테이블(before_2008)의 삭제 성능을 높이기 위해 **NO_STATS** 힌트를 사용하여 질의 실행 시간을 확인하는 예제이다. participant2 테이블에는 100만 건 이상의 데이터가 있는 것으로 가정한다. 아래 실행 시간의 차이는 시스템 성능 및 데이터베이스 구성 방법에 따라 다를 수 있다.

```
-- NO_STATS 힌트 미사용
ALTER TABLE participant2 DROP partition before 2008;
SQL statement execution time:      31.684550 sec
Current transaction has been committed.
1 command(s) successfully processed.

-- NO_STATS 힌트 사용
ALTER /*+ NO_STATS */ TABLE participant2 DROP partition before_2008;
SQL statement execution time:       0.025773 sec
Current transaction has been committed.
1 command(s) successfully processed.
```

쿼리 실행 계획 보기

설명

CUBRID SQL 질의에 대한 실행 계획(쿼리 플랜)을 보기 위해서는 **SET OPTIMIZATION** 구문을 이용해서 최적화 수준(optimization level) 값을 변경시킨다. 현재의 최적화 수준 값은 **GET OPTIMIZATION** 구문으로 얻을 수 있다.

CUBRID 질의 최적화기는 사용자에게 의해 설정된 최적화 수준 값을 참조하여 최적화 여부와 쿼리 실행 계획의 출력 여부를 결정한다. 쿼리 실행 계획은 표준 출력으로 표시되므로 CSQL 인터프리터와 같은 터미널 기반의 프로그램에서 사용하는 것을 가정하고 설명한다. CUBRID 매니저를 이용해서 쿼리 플랜을 보는 방법에 대해서는 [질의 실행 계획 보기](#)를 참고한다.

구문

```
SET OPTIMIZATION LEVEL opt-level [;]
GET OPTIMIZATION LEVEL [ { TO | INTO } variable ] [;]
```

- *opt-level*: 최적화 수준을 지정하는 값으로 다음과 같은 의미를 갖는다.
 - 0 : 질의 최적화를 수행하지 않는다. 실행하는 쿼리는 가장 단순한 형태의 실행 계획을 가지고 실행된다. 디버깅의 용도 이외에는 사용되지 않는다.
 - 1 : 질의 최적화를 수행한다. CUBRID에서 사용되는 기본 설정 값으로 대부분의 경우 변경할 필요가 없다.
 - 2: 질의 최적화를 수행하여 실행 계획을 생성하나 쿼리 자체는 수행되지 않는다. 일반적으로 사용되지 않고 다음 쿼리 실행 계획 보기를 위한 설정값과 같이 설정되어 사용된다.
 - 257 : 질의 최적화를 수행하여 생성된 쿼리 실행 계획(플랜)을 출력한다. 256+1의 값으로 해석하여 값을 1로 설정하고 쿼리 플랜 출력을 지정한 것과 같다.
 - 258 : 질의 최적화를 수행하여 생성된 쿼리 실행 계획을 출력한다. 257 설정과의 차이는 쿼리를 수행하지 않는다는 것이다. 즉, 256+2의 값으로 해석하여 값을 2로 설정하고 쿼리 플랜 출력을 지정한 것과 같다. 쿼리 실행 계획을 살펴보고자 하나 실행 결과에는 관심이 없을 경우 유용한 설정이다.
 - 513 : 질의 최적화를 수행하고 상세 쿼리 실행 계획을 출력한다. 512+1의 의미이다.
 - 514 : 질의 최적화를 수행하고 상세 쿼리 실행 계획을 출력하나 쿼리는 실행하지는 않는다. 512+2의 의미이다.

예제

다음은 심권호 선수가 메달을 획득한 연도와 메달 종류를 구하는 예제를 이용해 쿼리 실행 계획 보기를 수행한 것이다.

```
GET OPTIMIZATION LEVEL

=== < Result of GET OPTIMIZATION Command in Line 1> ===
      Result
=====
                        1
```

```
SET OPTIMIZATION LEVEL 258

SELECT a.name, b.host year, b.medal
FROM athlete a, game b WHERE a.name = 'Sim Kwon Ho' AND a.code = b.athlete code
Query plan:
  Nested loops
    Sequential scan(game b)
    Index scan(athlete a, pk athlete code, a.code=b.athlete code)

=== < Result of SELECT Command in Line 1> ===
There are no results.
0 rows selected.
```

트리거

CREATE TRIGGER

트리거 정의를 위한 가이드라인

트리거 정의로 다양하고 강력한 기능을 만들 수 있다. 트리거를 생성하기 전에 다음과 같은 정의 사항을 고려해야 한다.

- 트리거의 조건 영역 표현식이 데이터베이스에 예측할 수 없는 결과(side effect)를 가져오지는 않는가?
SQL 문을 예측이 가능한 범위 내에서 사용해야 한다.
- 트리거의 실행 영역이 트리거의 이벤트 대상으로 주어진 테이블을 변경하지는 않는가?
이러한 유형의 설계가 트리거의 정의에서 사용이 금지되어 있지는 않지만, 무한 루프로 빠지는 트리거가 만들어질 수 있어 주의해서 사용해야 한다. 트리거 실행 영역이 이벤트 대상 테이블을 수정할 때, 같은 트리거가 다시 불려질 수 있다. 또한 **WHERE** 절을 포함하는 문장에 의해 트리거가 발생하면, 해당 트리거는 **WHERE** 절에 의해 수행되는 테이블에는 일반적으로 부작용이 없다.
- 트리거가 불필요한 오버헤드를 만들어 내고 있지는 않는가?
원하는 동작이 소스 내에서 조금 더 효과적으로 표현될 수 있다면 직접 소스 내에서 구현하도록 한다.
- 트리거가 재귀적으로 실행되고 있지는 않는가?
트리거의 실행 영역이 트리거를 부르고 이 트리거가 다시 처음 트리거를 부르면 재귀 루프(recursive loop)가 데이터베이스에 만들어 진다. 재귀 루프가 만들지면, 트리거가 정확히 수행되지 않거나 진행 중인 루프를 단절하기 위해 현재 세션을 강제로 종료해야 할 수도 있다.
- 트리거의 정의는 유일한가?
동일한 테이블에서 정의된 트리거나, 동일한 실행 영역에서 시작된 트리거는 복구할 수 없는 에러의 원인이 된다. 동일한 테이블에 있는 트리거는 다른 트리거 이벤트를 가져야 한다. 또한, 트리거의 우선순위는 명시적으로 정의되어 있거나 모호하지 않아야 한다.

트리거 정의 구문

설명

CREATE TRIGGER 문을 사용하여 새로운 트리거를 생성하고, 트리거 대상, 실행 조건과 수행할 내용을 정의할 수 있다. 트리거는 데이터베이스 객체로서, 특정 이벤트가 대상 테이블에 대해 발생하면 정의된 동작을 수행한다.

구문

```
CREATE TRIGGER trigger_name
[ STATUS { ACTIVE | INACTIVE } ]
[ PRIORITYkey ]
event_time event_type[ event_target ]
[ IFcondition ]
```

```
EXECUTE [ AFTER | DEFERRED ] action [ ; ]
```

```
event time:
  • BEFORE
  • AFTER
  • DEFERRED
```

```
event type:
  • INSERT
  • STATEMENT INSERT
  • UPDATE
  • STATEMENT UPDATE
  • DELETE
  • STATEMENT DELETE
  • ROLLBACK
  • COMMIT
```

```
event target:
  • ON table_name
  • ON table name [ (column name) ]
```

```
condition:
  • expression
```

```
action:
  • REJECT
  • INVALIDATE TRANSACTION
  • PRINT message_string
  • INSERT statement
  • UPDATE statement
  • DELETE statement
```

- *trigger_name* : 정의하려는 트리거의 이름을 지정한다.
- [**STATUS** { **ACTIVE** | **INACTIVE** }] : 트리거의 상태를 정의한다(정의하지 않을 경우 디폴트 값은 **ACTIVE**).
 - **ACTIVE** 상태인 경우 관련 이벤트가 발생할 때마다 트리거를 실행한다.
 - **INACTIVE** 상태인 경우 관련 이벤트가 발생하여도 트리거를 실행하지 않는다. 트리거의 활성 여부는 변경할 수 있다. 자세한 내용은 [트리거 정의 변경](#)을 참조한다.
- [**PRIORITY** *key*] : 하나의 이벤트에 대해서 다수의 트리거가 불러질 경우 실행되는 우선순위를 부여한다. *key* 값은 반드시 음수가 아닌 부동 소수점 값이어야 한다. 우선순위를 정의하지 않을 경우 가장 낮은 우선순위인 0을 할당한다. 같은 우선순위를 가지는 트리거는 임의의 순서로 실행된다. 트리거의 우선순위는 변경할 수 있다. 자세한 내용은 [트리거 정의 변경](#)을 참조한다.
- *event_time* : 트리거의 조건 영역과 실행 영역이 실행되는 시점을 지정하며 **BEFORE**, **AFTER**, **DEFERRED**가 있다. 자세한 내용은 [이벤트 시점](#)을 참조한다.
- *event_type* : 트리거 타입은 사용자 트리거와 테이블 트리거로 나뉜다. 자세한 내용은 [트리거 이벤트 타입](#)을 참조한다.
- *event_target* : 이벤트 대상은 트리거가 호출되기 위한 대상을 지정할 때 쓰인다. 자세한 내용은 [트리거 이벤트 대상](#)을 참조한다.
- *condition* : 트리거의 조건영역을 지정한다. 자세한 내용은 [트리거 조건 영역](#)을 참조한다.
- *action* : 트리거의 실행영역을 지정한다. 자세한 내용은 [트리거 실행 영역](#)을 참조한다.

예제 1

다음은 participant 테이블의 레코드를 갱신할 때 획득 메달의 개수가 0보다 작을 경우 갱신을 거절하는 트리거를 생성하는 예제이다.

예로 2004년도 올림픽에 한국이 획득한 금메달의 개수를 음수로 갱신할 경우 갱신이 거절되는 것을 알 수 있다.

```
CREATE TRIGGER medal_trigger
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;

csql> UPDATE participant SET gold = -5 WHERE nation code = 'KOR'
AND host_year = 2004;
csql> ;x
In line 1, column 1,
ERROR: The operation has been rejected by trigger "medal_trigger".
```

예제 2

다음은 glo 객체를 이용하여 특정 경로에 있는 파일을 DB에 저장하는 경우, 사용자가 정의한 테이블에서 glo 데이터를 삭제할 때, 시스템 테이블 중 하나인 glo 테이블에서 해당 데이터에 관한 데이터를 함께 삭제하는 트리거를 생성하는 예제이다. glo 타입의 데이터가 입력되면 CUBRID는 glo 테이블에 해당 파일을 저장하고 반환되는 OID 값을 **GLOData** 타입으로 사용한 컬럼에 저장한다. 따라서 glo를 사용할 때에는 항상 glo 테이블에 존재하는 관련 데이터를 함께 삭제하는 트리거를 생성하는 것을 권장한다.

```
CREATE TABLE person (name string, photo glo);

CREATE TRIGGER person_delete_glo
BEFORE DELETE ON person
IF obj.photo is not null
EXECUTE DELETE FROM glo where glo=obj.photo;

csql> ;sc person

=== <Help: Schema of a Class> ===

<Class Name>

    person

<Attributes>

    name                CHARACTER VARYING(1073741823)
    photo                glo

<Triggers>

    person_delete_glo : BEFORE DELETE
```

이벤트 시점

설명

트리거의 조건 영역과 실행 영역이 실행되는 시점을 지정한다. 이벤트 시점의 종류에는 **BEFORE**, **AFTER**, **DEFERRED**가 있다.

- **BEFORE** : 이벤트가 처리되기 이전에 조건을 검사한다.
- **AFTER** : 이벤트가 처리된 후에 조건을 검사한다.
- **DEFERRED** : 이벤트에 대한 트랜잭션의 끝에서 조건을 검사한다. **DEFERRED**로 지정할 경우 이벤트 타입에 **COMMIT**이나 **ROLLBACK**을 사용할 수 없다.

트리거 타입

사용자 트리거(User Trigger)

- 데이터베이스의 특정 사용자와 관련된 트리거를 사용자 트리거(user trigger)라고 한다.
- 사용자 트리거는 이벤트 대상이 없으며 트리거의 소유자(트리거를 생성한 사용자)에 의해서만 실행된다.
- 사용자 트리거를 정의하는 이벤트 타입은 **COMMIT**과 **ROLLBACK**이 있다.

테이블 트리거(Table Trigger)

- 특정 테이블을 이벤트 대상으로 가지는 트리거를 테이블 트리거(클래스 트리거)라 한다.
- 테이블 트리거는 대상 테이블에 **SELECT** 권한을 가지는 모든 사용자가 볼 수 있다.
- 테이블 트리거를 정의하는 이벤트 타입은 인스턴스 이벤트와 문장 이벤트가 있다.

트리거 이벤트 타입

설명

- 인스턴스 이벤트(instance event) : 인스턴스 이벤트는 이벤트 연산의 단위가 인스턴스인 이벤트 타입을 말한다. 인스턴스 이벤트의 종류는 다음과 같다.
 - **INSERT**
 - **UPDATE**
 - **DELETE**
- 문장 이벤트(statement event) : 이벤트 타입을 문장 이벤트로 정의하면 주어진 문장(이벤트)에 의해 영향을 받는 객체(인스턴스)가 많더라도, 트리거는 문장이 시작할 때 한 번만 불려지게 된다. 문장 이벤트의 종류는 다음과 같다.
 - **STATEMENT INSERT**
 - **STATEMENT UPDATE**
 - **STATEMENT DELETE**
- 기타 이벤트 : **COMMIT**과 **ROLLBACK**은 개별적인 인스턴스에는 적용할 수 없다.
 - **COMMIT**
 - **ROLLBACK**

예제 1

다음은 인스턴스 이벤트를 사용하는 예제이다. example 트리거는 데이터베이스 갱신에 의해 영향을 받는 각각의 인스턴스에 대해서 한번씩 불려진다. 예를 들어, history 테이블의 다섯 개 인스턴스의 score를 변경했다면, 이 트리거는 다섯 번 불려진다. 만약 score 컬럼의 첫 번째 인스턴스가 갱신되기 전에 트리거가 한 번만 불려지게 하려면, 예제 2와 같이 **STATEMENT UPDATE** 형식을 사용한다.

```
CREATE TRIGGER example
...
BEFORE UPDATE ON history(score)
...
```


예제 2

다음은 문장 이벤트를 사용하는 예제이다. 문장 이벤트를 지정하면 갱신의 영향을 받는 인스턴스가 많더라도, 첫 번째 인스턴스가 갱신되기 전에 트리거가 한 번만 불려지게 된다.

```
CREATE TRIGGER example
...
BEFORE STATEMENT UPDATE ON history(score)
...
```

주의 사항

- 이벤트 타입으로 인스턴스 이벤트와 문장 이벤트를 지정할 경우에는 반드시 이벤트 대상을 명시해야 한다.
- COMMIT**과 **ROLLBACK**은 이벤트 대상을 가질 수 없다.

트리거 이벤트 대상

설명

이벤트 대상은 트리거가 호출되기 위한 대상을 지정할 때 쓰인다. 트리거 이벤트의 대상은 테이블명 혹은 테이블명과 컬럼명으로 지정할 수 있으며 컬럼명을 지정하면 해당 컬럼이 이벤트의 영향을 받을 때에만 트리거가 불려진다. 만약 컬럼을 지정하지 않으면 지정된 테이블 내에 어떤 컬럼이 영향을 받더라도 트리거가 호출된다. 오직 **UPDATE**, **STATEMENT UPDATE** 이벤트만이 이벤트 대상에 컬럼을 지정할 수 있다.

예제

다음은 example 트리거의 이벤트 대상을 history 테이블의 score 컬럼으로 지정한 예제이다.

```
CREATE TRIGGER example
...
BEFORE UPDATE ON history(score)
...
```

이벤트 타입과 대상 조합

설명

트리거를 호출하는 데이터베이스 이벤트는 트리거 이벤트 타입과 트리거 정의 내의 이벤트 대상에 의해 식별된다. 다음은 트리거 이벤트 타입과 대상 조합, 트리거 이벤트가 나타내는 CUBRID 데이터베이스 이벤트의 활동을 표로 정리한 것이다.

이벤트 타입	이벤트 대상	대응되는 데이터베이스 활동
UPDATE	테이블	테이블의 어떤 컬럼에 대해서도 갱신되면 트리거가 호출된다.
STATEMENT UPDATE	테이블	테이블에 대해 UPDATE 문이 실행되었을 때 트리거가 호출된다.

INSERT	테이블	테이블의 인스턴스가 생성되었을 때 트리거가 호출된다.
STATEMENT INSERT	테이블	테이블에 INSERT 문이 실행되었을 때 트리거가 호출된다.
DELETE	테이블	테이블의 인스턴스가 삭제되었을 때 트리거가 호출된다.
STATEMENT DELETE	테이블	테이블에 DELETE 문이 실행되었을 때 트리거가 호출된다.
COMMIT	없음	데이터베이스 트랜잭션이 커밋되었을 때 트리거가 호출된다. COMMIT WORK 문이 트리거를 시작시킨다.
ROLLBACK	없음	데이터베이스의 트랜잭션이 롤백되었을 때 트리거가 호출된다. ROLLBACK WORK 문이 트리거를 시작시킨다.

트리거 조건 영역

설명

트리거를 정의할 때 조건 영역을 정의하여 트리거의 수행 영역에 대한 수행 여부를 결정한다.

- 트리거 조건 영역이 기술된다면, 참 또는 거짓을 평가할 수 있는 단독적인 복합 표현식으로 쓰여질 수 있다. 이 경우에 표현식은 **SELECT** 문의 **WHERE** 절에 허용되는 산술 연산자와 논리 연산자를 포함할 수 있다. 조건 영역이 참이면, 트리거 실행 영역이 수행되고, 거짓이면 실행되지 않는다.
- 트리거의 조건 영역을 생략하면 조건 없는 트리거(unconditional trigger)가 되며 이는 트리거가 불려질 때 항상 트리거의 실행 영역이 수행된다.

예제 1

다음은 조건 영역 내의 표현식에 상관명을 이용한 예제이다. 이벤트 타입이 **INSERT**, **UPDATE**, **DELETE**인 경우에, 조건 영역 내의 표현식은 특정 컬럼 값에 접근하기 위하여 상관명 **obj**, **new**, **old**를 사용할 수 있다. 예제에서 example 트리거는 record 컬럼의 현재 값을 이용해서 조건 영역을 검사하기 위해 트리거 조건 영역에 **obj**를 컬럼 이름 앞에 사용하였다.

```
CREATE TRIGGER example
.....
IF obj.record * 1.20 < 500
.....
```

예제 2

다음은 조건 영역 내의 표현식에 **SELECT** 문을 사용한 예제이다. 예제의 트리거는 집계함수 **COUNT(*)**를 사용하는 **SELECT** 문을 사용하여 그 값과 상수를 비교한다. **SELECT** 문은 반드시 괄호로 싸여 있어야 하고, 표현식의 마지막에 위치해야 한다.

```
CREATE TRIGGER example
.....
IF 1000 > (SELECT COUNT( * ) FROM participant)
.....
```

주의 사항

트리거 조건 영역에 주어진 표현식은 조건 영역이 수행되는 동안에 메소드가 호출되면 데이터베이스에 부작용을 초래할 수 있다. 트리거 조건 영역은 데이터베이스에 생각지 못한 부작용이 발생하지 않도록 구성해야 한다.

상관명(correlation name)

트리거를 정의할 때 상관명을 사용하여 대상 테이블의 컬럼 값에 접근할 수 있다. 상관명은 실제로 트리거를 부르는 데이터베이스 연산에 의해 영향을 받는 인스턴스를 나타낸다. 상관명은 트리거의 조건 영역이나 실행 영역에도 기술할 수 있다.

상관명의 종류에는 **new**, **old**, **obj**가 있으며 이러한 상관명은 인스턴스 트리거에서 **INSERT**, **UPDATE**, **DELETE**의 이벤트 타입을 가지고 있는 트리거에서만 사용할 수 있다.

상관명의 사용은 아래 표와 같이 트리거 조건 영역에 정의된 이벤트 시점에 의해 더욱 제한된다.

	BEFORE	AFTER or DEFERRED
INSERT	new	obj
UPDATE	obj new	obj old(AFTER)
DELETE	obj	NA

상관명	대표 속성 값
obj	인스턴스의 현재 속성 값을 나타낸다. 인스턴스가 갱신되거나 삭제되기 전에 속성값에 접근하기 위해서 사용한다. 그리고 인스턴스가 갱신되거나 삽입된 후에 속성 값에 접근하기 위해 사용한다.
new	삽입이나 갱신 연산에 의해 제시되는 속성값을 나타낸다. 새로운 값은 인스턴스가 실제로 삽입되거나 갱신되기 전에만 접근할 수 있다.
old	갱신 연산의 완료 전에 존재하던 속성값을 나타낸다. 이 값은 트리거가 수행되는 동안만 유지된다. 트리거가 종료되면 old 값은 잃어버리게 된다.

트리거 실행 영역

설명

트리거 실행 영역은 트리거의 조건 영역이 참이거나 조건 영역이 생략된 경우 수행될 내용을 기술하는 영역이다. 실행 영역 절에 특정 시점(**AFTER**나 **DEFERRED**)이 주어지지 않으면, 실행 영역은 트리거 이벤트와 같은 시점에서 수행된다.

아래 목록은 트리거를 정의할 때 사용할 수 있는 실행 영역의 목록이다.

- **REJECT** : 트리거에서 조건 영역이 참이 아닌 경우 트리거를 발동시킨 연산은 거절되고 데이터베이스의 이전 상태를 그대로 유지한다. 연산이 수행된 후에는 거절할 수 없기 때문에 **REJECT**는 실행 시점이 **BEFORE**일 때만 허용된다. 따라서 실행 시점이 **AFTER**나 **DEFERRED**인 경우 **REJECT**를 사용해서는 안 된다.
- **INVALIDATE TRANSACTION** : 트리거를 부른 이벤트 연산은 수행되지만, 커밋을 포함하고 있는 트랜잭션은 수행되지 않도록 한다. 트랜잭션이 유효하지 않으면 반드시 **ROLLBACK** 문으로 취소시켜야 한다. 이러한 실행은 데이터를 변경하는 이벤트가 발생한 후에 유효하지 않은 데이터를 가지는 것으로부터 데이터베이스를 보호하기 위해 사용된다.
- **PRINT** : 터미널 화면에 텍스트 메시지로 트리거 활동을 가시적으로 보여주기 때문에 트리거의 개발이나 시험하는 도중에 사용될 수 있다. 이벤트 연산의 결과를 거절하거나 무효화시키지는 않는다.
- **INSERT** : 테이블에 하나 혹은 그 이상의 새로운 인스턴스를 추가한다.
- **UPDATE** : 테이블에 있는 하나 혹은 그 이상의 컬럼 값을 변경한다.
- **DELETE** : 테이블로부터 하나 혹은 그 이상의 인스턴스를 제거한다.

예제

다음은 트리거 생성 시에 실행영역의 정의 방법을 보여주는 예제이다. medal_trig 트리거는 실행 영역에 **REJECT**를 지정하였다. **REJECT**는 실행 시점이 **BEFORE**일 때만 지정 가능하다.

```
CREATE TRIGGER medal_trig
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;
```

주의 사항

- **INSERT** 이벤트가 정의된 트리거의 실행 영역에 **INSERT**를 사용할 때는 트리거가 무한 루프에 빠질 수 있으므로 주의해야 한다.
- **UPDATE** 이벤트가 정의된 트리거가 분할된 테이블에서 동작하는 경우, 정의된 분할이 깨지거나 의도하지 않은 오동작이 발생할 수 있으므로 주의해야 한다. 이를 방지하기 위해 CUBRID는 트리거가 동작중인 경우 분할 변경을 야기하는 **UPDATE**가 실행되지 않도록 오류 처리한다. **UPDATE** 이벤트가 정의된 트리거의 실행 영역에 **UPDATE**를 사용할 때는 무한 루프에 빠질 수 있으므로 주의해야 한다.

ALTER TRIGGER

설명

트리거 정의에서 **STATUS**와 **PRIORITY** 옵션에 대해 **ALTER** 구문을 이용하여 변경할 수 있다. 만약 트리거의 다른 부분에 대해 변경(이벤트 대상 또는 조건 표현식)이 필요하다면, 트리거를 삭제한 후, 재생성해야 한다.

구문

```
ALTER TRIGGER trigger_name trigger_option [ ; ]
```

trigger_option :

- **STATUS** { **ACTIVE** | **INACTIVE** }
- **PRIORITY** key

- *trigger_name*: 변경할 트리거의 이름을 지정한다.
- *trigger_option*:
 - **STATUS** { **ACTIVE** | **INACTIVE** }: 트리거의 상태를 변경한다.
 - **PRIORITY** key: 우선순위를 변경한다.

예제

다음은 medal_trig 트리거를 생성하고 트리거의 상태를 **INACTIVE**로, 우선순위를 0.7로 변경하는 예제이다.

```
CREATE TRIGGER medal_trig
STATUS ACTIVE
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;

ALTER TRIGGER medal_trig STATUS INACTIVE;
ALTER TRIGGER medal_trig PRIORITY 0.7;
```

주의 사항

- 같은 **ALTER TRIGGER** 문 내에서는 한 개 옵션만 기술할 수 있다.
- 만약 테이블 트리거를 변경하려면, 해당 트리거의 소유자이거나, 해당 트리거가 있는 테이블에 대해 **ALTER** 권한이 부여되어 있어야 한다.
- 사용자 트리거를 변경하기 위해서는 반드시 해당 트리거의 소유자여야 한다. 옵션에 대한 자세한 내용은 [CREATE TRIGGER \(구문\)](#)을 참조한다. **PRIORITY** 옵션과 같이 기술하는 key는 반드시 음이 아닌 부동 소수점 값(non-negative floating point value)이어야 한다.

DROP TRIGGER

설명

DROP TRIGGER 구문을 이용하여 트리거를 삭제한다.

구문

```
DROP TRIGGER trigger_name [ ; ]
```

- *trigger_name* : 삭제할 트리거의 이름을 지정한다.

예제

다음은 medal_trig 트리거를 삭제하는 예제이다.

```
DROP TRIGGER medal_trig;
```

주의 사항

- 트리거가 사용자 트리거(즉 트리거 이벤트가 **COMMIT**이거나 **ROLLBACK**)이면, 트리거의 소유자만 볼 수 있고 소유자만 제거할 수 있다.
- 한 개의 **DROP TRIGGER** 문에서는 한 개의 트리거만 제거할 수 있다. 테이블 트리거는 트리거가 속해 있는 테이블에 대해 **ALTER** 권한이 있는 사용자에 의해 제거될 수 있다.

RENAME TRIGGER

설명

트리거의 이름은 **RENAME** 구문의 **TRIGGER** 예약어를 이용해서 변경한다.

구문

```
RENAME TRIGGER old_trigger_name AS new_trigger_name [ ; ]
```

- *old_trigger_name* : 트리거의 현재 이름을 입력한다.
- *new_trigger_name* : 변경할 트리거의 이름을 지정한다.

예제

```
RENAME TRIGGER medal_trigger AS medal_trig;
```

주의 사항

- 트리거 이름은 모든 트리거 사이에서 유일해야 한다. 하지만 데이터베이스 내의 테이블 이름과 같은 이름을 가질 수는 있다.
- 만약 테이블 트리거의 이름을 변경하려면, 트리거의 소유자이거나, 해당 트리거가 있는 테이블에 대해 **ALTER** 권한이 부여되어 있어야 한다. 사용자 트리거는 트리거의 소유자만 이름을 변경할 수 있다.

지연된 트리거

정의

지연된 트리거 실행영역과 조건 영역은 나중에 실행되거나 취소될 수 있다. 이러한 트리거들은 이벤트 시점(event time)이나 실행 영역(action) 절에 **DEFERRED** 시간 옵션을 포함하고 있다. **DEFERRED** 옵션이 이벤트 시점에 기술되고, 실행 영역 앞에 시간이 생략되었다면, 실행 영역은 자동으로 지연된다.

지연된 영역 실행

설명

지연된 트리거의 조건 영역이나 실행 영역을 즉시 실행시킨다.

구문

```
EXECUTE DEFERRED TRIGGER trigger_identifier [ ; ]
```

trigger_identifier :

- trigger name
- **ALL TRIGGERS**

• trigger_identifier :

- trigger_name : 트리거의 이름을 지정하면 지정된 트리거의 지연된 활동이 실행된다.
- **ALL TRIGGERS** : 현재 모든 지연된 활동이 실행된다.

지연된 영역 취소

설명

지연된 트리거의 조건 영역과 실행 영역을 취소한다.

구문

```
DROP DEFERRED TRIGGER trigger_identifier [ ; ]
```

trigger_option :

- trigger name
- **ALL TRIGGERS**

• trigger_option :

- trigger_name : 트리거의 이름을 지정하면 지정된 트리거의 지연된 활동이 취소된다.
- **ALL TRIGGERS** : 현재 모든 지연된 활동이 취소된다.

트리거 권한 부여

설명

트리거에 대한 권한은 명시적으로 부여되지 않는다. 트리거의 정의에 기술된 이벤트 대상 테이블에 권한이 부여되었을 때 사용자는 테이블 트리거에 대한 권한을 자동적으로 획득한다. 다시 말하자면, 테이블 대상(**INSERT**, **UPDATE** 등)을 가지는 트리거는 해당 테이블에 적절한 권한을 가지는 모든 사용자에게 보인다. 사용자 트리거(**COMMIT**과 **ROLLBACK**)는 트리거를 정의한 사용자만 볼 수 있다. 트리거의 소유자이면 모든 권한은 자동적으로 부여된다.

주의 사항

- 테이블 트리거를 정의하기 위해서는 관련된 테이블에 **ALTER** 권한이 반드시 있어야 한다.

- 사용자 트리거를 정의하기 위해서는 유효한 사용자를 이용하여 데이터베이스에 접근하는 것이 필요하다.

트리거 디버깅

정의와 예제

설명

트리거를 정의한 후에는 트리거가 의도한 대로 동작하는지 검사하는 것이 좋다. 종종 트리거가 기대했던 것보다 처리하는데 오랜 시간이 걸리는 경우가 있다. 이는 시스템에 너무 많은 오버헤드를 주거나, 재귀적 루프에 빠졌다는 뜻이다. 이 절에서는 트리거를 디버깅하는 몇 가지 방법을 설명한다.

예제

다음은 호출되면 재귀적으로 루프에 빠지도록 정의한 트리거이다. loop 트리거는 목적이 다소 인위적이지만 트리거를 디버깅하는 예제로 사용될 수 있다.

```
CREATE TRIGGER loop_tgr
BEFORE UPDATE ON participant(gold)
IF new.gold > 0
EXECUTE UPDATE participant
    SET gold = new.gold - 1
    WHERE nation_code = obj.nation_code AND host_year = obj.host_year;
```

트리거 실행 로그 보기

설명

SET TRIGGER TRACE 문을 이용하여 터미널에서 트리거의 실행 로그를 볼 수 있다.

구문

```
SET TRIGGER TRACE switch [ ; ]
```

switch:

- **ON**
- **OFF**

- *switch* :
 - **ON** : **TRACE**가 작동되며 **OFF**하거나 현재 데이터베이스 세션을 종료할 때까지 계속 유지된다.
 - **OFF** : **TRACE**의 작동을 멈춘다.

예제

다음 예제는 트리거의 실행 로그를 보기 위해 **TRACE**를 작동시키고 loop 트리거를 작동시키는 예제이다. 트리거가 불려질 때 수행된 각각의 조건 영역과 실행 영역에 대한 추적을 식별하기 위한 메시지가 터미널에 나타난다. loop 트리거는 gold 값이 0이 될때까지 실행되므로 예제에서는 아래의 메시지가 15번 나타난다.


```
SET TRIGGER TRACE ON;

UPDATE participant SET gold = 15 WHERE nation code = 'KOR' AND host year = 1988;
TRACE: Evaluating condition for trigger "loop".
TRACE: Executing action for trigger "loop".
```

중첩된 트리거 제한

설명

SET TRIGGER 문의 **MAXIMUM DEPTH** 키워드를 이용하여 단계적으로 발동되는 트리거 수를 제한할 수 있다.

이를 이용하면 재귀적으로 호출되는 트리거가 무한루프에 빠지는 것을 막을 수 있다.

구문

```
SET TRIGGER [ MAXIMUM ] DEPTH count [ ; ]
```

count:

- *unsigned integer literal*
- **INFINITE**

• *count*:

- *unsigned_integer_literal*: 양의 정수값으로 트리거가 다른 트리거나 자신을 재귀적으로 발동할 수 있는 횟수를 지정한다. 트리거의 수가 최대 깊이에 도달하면 데이터베이스 요청은 중단되고 트랜잭션은 유효하지 않은 것처럼 표시된다. 설정된 **DEPTH**는 현재 세션을 제외한 나머지 모든 트리거에 적용된다.
- **INFINITE**: 설정한 횟수 제한을 제거한다.

예제

다음은 재귀적 트리거 호출의 최대 값을 10으로 설정하는 예제이다. 이는 이후에 발동하는 모든 트리거에 적용된다. 이 예제에서 gold 컬럼에 대한 값은 15로 갱신되어 트리거는 총 16번 불려지게 된다. 이는 현재 설정된 최대 깊이를 초과하게 되고 아래와 같은 에러 메시지가 발생한다.

```
SET TRIGGER MAXIMUM DEPTH 10;

csql> UPDATE participant SET gold = 15 WHERE nation code = 'KOR' AND host year = 1988;
csql> ;x

In line 3, column 2,
ERROR: Maximum trigger depth 10 exceeded at trigger "loop_tgr".
```

트리거를 이용한 응용

설명

여기에서는 데모 데이터베이스에 있는 트리거 정의에 대해 알아본다. demodb 데이터베이스에 생성되어 있는 트리거는 그리 복잡하지는 않지만 CUBRID에서 사용할 수 있는 대부분의 기능을 활용한다. 이러한 트리거를 테스트할 때, demodb 데이터베이스의 원형을 유지하고 싶다면 데이터에 변경이 발생한 후 롤백을 수행해야 한다.

사용자 데이터베이스에 직접 생성한 트리거는 사용자가 만든 응용 프로그램만큼이나 강력할 수 있다.

예제 1

participant 테이블에 만들어진 아래 트리거는 제시된 값이 0보다 작을 때 메달 컬럼(gold, silver, bronze)에 대한 업데이트를 거절한다. 트리거의 조건에 상관명 new가 사용되었기 때문에 시작 시점(evaluation time)은 반드시 **BEFORE**가 되어야 한다. 비록 기술하지는 않았지만, 이 트리거에서 실행 시점(action time) 또한 **BEFORE**이다.

```
CREATE TRIGGER medal_trigger
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;
```

국가코드가 'BLA'인 나라의 금메달(gold) 수를 업데이트 할 때, medal_trigger 트리거가 발동한다. 금메달 수가 음수인 경우를 허용하지 않도록 트리거를 생성하였으므로, 업데이트를 허용하지 않는다.

```
UPDATE participant
SET gold = -10
WHERE nation_code = 'BLA';
```

예제 2

아래 트리거는 위의 예제와 같은 조건인데, **STATUS ACTIVE**가 추가된 경우이다. **STATUS** 문이 생략될 경우 기본값은 **ACTIVE**이며, **ALTER TRIGGER** 문에 의해 **STATUS**를 **INACTIVE**로 변경할 수 있다.

STATUS의 값에 따라 트리거의 실행 여부를 지정할 수 있다.

```
CREATE TRIGGER medal_trig
STATUS ACTIVE
BEFORE UPDATE ON participant
IF new.gold < 0 OR new.silver < 0 OR new.bronze < 0
EXECUTE REJECT;

ALTER TRIGGER medal_trig
STATUS INACTIVE;
```

예제 3

다음 트리거는 트랜잭션이 커밋되었을 때 어떻게 무결성 제약 조건을 강제적으로 수행하는지 보여 준다. 하나의 트리거가 여러 테이블에 대해 지정 조건을 넣을 수 있다는 점이 이전 예제와 다르다.

```
CREATE TRIGGER check_null_first
BEFORE COMMIT
IF 0 < (SELECT count(*) FROM athlete WHERE gender IS NULL)
OR 0 < (SELECT count(*) FROM game WHERE nation code IS NULL)
EXECUTE REJECT;
```

예제 4

다음 트리거는 record 테이블에 대해서 트랜잭션이 커밋될 때까지 업데이트 무결성 제약조건 검사를 지연시킨다. **DEFERRED** 키워드가 이벤트 시점으로 주어졌기 때문에 업데이트 실행 시점에 즉시 트리거가 실행되지 않는다. 정확하게 시작될 필요는 없다.

```
CREATE TRIGGER deferred_check_on_record
DEFERRED UPDATE ON record
```

```
IF obj.score = '100'  
EXECUTE INVALIDATE TRANSACTION;
```

record 테이블에서 업데이트가 완료되었을 때, 해당 업데이트는 현재 트랜잭션의 마지막(커밋이나 롤백할 때)에 확인하게 된다. 상관명 **old**는 **DEFERRED UPDATE**를 사용하는 트리거의 조건 절에 사용할 수 없다. 따라서 아래와 같은 트리거는 생성할 수 없다.

```
CREATE CLASS foo (n int);  
CREATE TRIGGER foo trigger  
  DEFERRED UPDATE ON foo  
  IF old.n = 100  
  EXECUTE PRINT 'foo_trigger';
```

위와 같이 트리거를 생성하려고 하면 다음과 같은 에러 메시지를 보여주고, 실패한다.

```
ERROR: Error compiling condition for 'foo_trigger' : old.n is not defined.
```

상관명 **old**는 트리거의 조건 시간이 **AFTER**일 때에만 사용될 수 있다.

Java 저장 함수/프로시저

개요

저장 함수와 저장 프로시저를 사용하면 SQL로 구현하지 못하는 복잡한 프로그램의 로직을 구현할 수 있으며, 사용자가 보다 쉽게 데이터를 조작하게 할 수 있다. 함수와 프로시저는 데이터를 조작하기 위해 실행 명령의 흐름이 있고, 쉽게 조작할 수 있고, 관리할 수 있는 블록 단위라고 할 수 있다.

CUBRID는 Java로 저장 함수와 프로시저를 개발할 수 있도록 지원한다. Java 저장 함수와 프로시저는 CUBRID에서 호스팅한 Java 가상 머신(JVM, Java Virtual Machine)에서 실행된다.

Java 저장 함수/프로시저는 SQL에서도 호출할 수 있으며, JDBC를 사용하여 쉽게 Java 응용 프로그램에서 호출할 수 있다.

Java 저장 함수/프로시저를 사용할 때 얻을 수 있는 이점은 다음과 같다.

- **생산성과 사용성** : Java 저장 함수/프로시저는 한번 만들어 놓으면 계속해서 사용할 수 있다. 사용자가 저장 함수와 저장 프로시저를 SQL에서도 호출하여 사용할 수 있고, JDBC를 사용하여 쉽게 Java 응용 프로그램에서 호출할 수 있다.
- **뛰어난 상호 운용성, 이식성** : Java 저장 함수/프로시저는 Java 가상 머신을 사용하므로, 시스템에 Java 가상 머신을 사용할 수만 있다면 언제 어디서나 사용할 수 있다.

Java 저장 함수/프로시저 사용을 위한 환경 설정

CUBRID에서 Java 저장 함수/프로시저를 사용하기 위해서는 CUBRID 서버가 설치되는 환경에 Java Runtime Environment (JRE) 1.5 이상 버전이 설치되어야 한다. JRE는 Developer Resources for Java Technology 사이트(<http://java.sun.com>)에서 다운로드할 수 있다.

CUBRID 환경 설정 파일(cubrid.conf)에 java_stored_procedure 파라미터가 yes로 설정되어 있으면, CUBRID 64비트 버전에는 JRE 64비트 버전이 필요하고, CUBRID 32비트 버전에는 JRE 32비트 버전이 필요하다. JRE 32비트 버전이 설치된 컴퓨터에서 CUBRID 64비트 버전을 실행하면 아래와 같은 에러 메시지가 출력된다.

```
% cubrid server start demodb

This may take a long time depending on the amount of recovery works to do.
WARNING: Java VM library is not found :
/usr/java/jdk1.5.0_15/jre/lib/amd64/server/libjvm.so: cannot open shared object file: No
such file or directory.
Consequently, calling java stored procedure is not allowed
```

JRE가 이미 설치되어 있다면, 아래와 같은 명령으로 버전을 확인한다.

```
% java -version
Java(TM) SE Runtime Environment (build 1.6.0_05-b13)
Java HotSpot(TM) 64-Bit Server VM (build 10.0-b19, mixed mode)
```

Windows 환경

CUBRID는 Windows 환경에서 **jvm.dll** 파일을 로딩하여 Java 가상 머신을 실행시킨다. CUBRID는 먼저 시스템의 **Path** 환경 변수에서 **jvm.dll**을 찾아 로딩한다. 만약 찾지 못하면 시스템 레지스트리에 등록된 Java 런타임 정보를 이용한다.

아래와 같이 명령어를 실행하여 **JAVA_HOME** 환경 변수를 설정하고 Java 실행 파일이 있는 디렉터리를 **Path** 환경 변수에 추가할 수 있다. GUI를 이용해서 환경 변수를 설정하는 방법은 [JDBC 환경 설정](#)을 참고한다.

- JDK 1.5 64비트 버전을 설치하고, 환경 변수를 설정한 예

```
% set JAVA_HOME=C:\jdk1.5.0
% set PATH=%PATH%;%JAVA_HOME%\jre\bin\server
```

- JDK 1.5 32비트 버전을 설치하고, 환경 변수를 설정한 예

```
% set JAVA_HOME=C:\jdk1.5.0
% set PATH=%PATH%;%JAVA_HOME%\jre\bin\client
```

SUN의 Java 가상 머신을 사용하지 않고 다른 벤더의 구현을 사용하려면 해당 벤더의 설치에서 **jvm.dll** 파일의 경로를 **PATH**에 추가해 주어야 한다.

Linux/Unix 환경

CUBRID는 Linux/Unix 환경에서 **libjvm.so** 파일을 로딩하여 Java 가상 머신을 실행시킨다. CUBRID는 먼저 **LD_LIBRARY_PATH** 환경 변수에서 **libjvm.so** 파일을 찾아 로딩한다. 만약 찾지 못하면 **JAVA_HOME** 환경 변수를 이용하여 찾는다. 리눅스의 경우 glibc 2.3.4 이상만 지원되며, 아래는 리눅스 환경 설정 파일(예 : **.profile**, **.cshrc**, **.bashrc**, **.bash_profile** 등)에 환경 변수를 설정하는 예이다.

- JDK 1.5 64비트 버전을 설치하고, bash 셸에서 환경 변수를 설정한 예

```
% JAVA_HOME=/usr/java/jdk1.5.0_10
%
LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/amd64:$JAVA_HOME/jre/lib/amd64/server:$LD_LIBRARY_PATH
% export JAVA_HOME
% export LD_LIBRARY_PATH
```

- JDK 1.5 32비트 버전을 설치하고, bash 셸에서 환경 변수를 설정한 예

```
% JAVA_HOME=/usr/java/jdk1.5.0_10
%
LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/i386:$JAVA_HOME/jre/lib/i386/client:$LD_LIBRARY_PATH
% export JAVA_HOME
% export LD_LIBRARY_PATH
```

- JDK 1.5 64비트 버전을 설치하고, csh 셸에서 환경 변수를 설정한 예

```
% setenv JAVA_HOME /usr/java/jdk1.5.0_10
% setenv LD_LIBRARY_PATH
$JAVA_HOME/jre/lib/amd64:$JAVA_HOME/jre/lib/amd64/server:$LD_LIBRARY_PATH
% set path=($path $JAVA_HOME/bin .)
```

- JDK 1.5 32비트 버전을 설치하고, csh 셸에서 환경 변수를 설정한 예

```
% setenv JAVA_HOME /usr/java/jdk1.5.0_10
% setenv LD_LIBRARY_PATH
$JAVA_HOME/jre/lib/i386:$JAVA_HOME/jre/lib/i386/client:$LD_LIBRARY_PATH
% set path=($path $JAVA_HOME/bin .)
```

SUN 이외의 다른 벤더가 제공하는 Java 가상 머신을 사용하는 경우, Library Path에 Java VM(**libjvm.so**)이 있는 경로를 추가해 주어야 한다. 이때, **libjvm.so** 파일의 경로는 OS 플랫폼, 지원 비트마다 다르므로 주의하여 설정한다. 예를 들어 SUN Sparc 머신에서 **libjvm.so** 파일의 경로는 **\$JAVA_HOME/jre/lib/sparc**이다.

Java 저장 함수/프로시저 작성 단계

Java 저장 함수/프로시저를 작성하는 순서는 다음과 같다.

- [cubrid.conf 확인](#)
- [Java 소스 작성 및 컴파일](#)
- [CUBRID로 Java 클래스 로드](#)
- [로딩한 Java 클래스 등록](#)
- [Java 저장 함수/프로시저 호출](#)

cubrid.conf 확인

cubrid.conf에 있는 **java_stored_procedure**의 설정값은 **no**가 기본이다. Java 저장함수/프로시저를 사용하기 위해서는 이 값을 **yes**로 변경해야 한다. 이 값과 관련한 자세한 설명은 데이터베이스 서버 설정의 [기타 파라미터](#)를 참조한다.

Java 소스 작성 및 컴파일

다음과 같이 SpCubrid.java를 컴파일 한다.

```
public class SpCubrid{
    public static String HelloCubrid() {
        return "Hello, Cubrid !!";
    }
    public static int SpInt(int i) {
        return i + 1;
    }
    public static void outTest(String[] o) {
        o[0] = "Hello, CUBRID";
    }
}

%javac SpCubrid.java
```

이 때, Java 클래스의 메소드는 반드시 public static이어야 한다.

CUBRID 로 컴파일된 Java 클래스 로드

컴파일된 Java 클래스를 CUBRID로 로딩한다.

```
% loadjava demodb SpCubrid.class
```

로딩한 Java 클래스 등록

다음과 같이 CUBRID 저장 함수를 생성하여 Java 클래스를 등록한다.

```
csql> create function hello() return string
csql> as language java
```

```
csql> name 'SpCubrid.HelloCubrid() return java.lang.String';
csql> ;xrun
```

Java 저장 함수/프로시저 호출

다음과 같이 등록된 Java 저장 함수를 호출한다.

```
csql> call hello() into :Hello;
csql> ;xrun

=== < Result of CALL Command in Line 1 > ===

      Result
=====
'Hello, Cubrid !!'
```

서버 내부 JDBC 드라이버 사용

Java 저장 함수/프로시저에서 데이터베이스에 접근하기 위해서는 서버 측 JDBC 드라이버(Server-Side JDBC Driver)를 사용해야 한다. Java 저장 함수/프로시저가 데이터베이스 내에서 실행되기 때문에 서버 측 JDBC 드라이버는 다시 연결을 설정할 필요가 없다. 서버 측 JDBC 드라이버로 해당 데이터베이스의 Connection을 얻는 방법은 아래와 같다. 첫 번째 방법은 JDBC 연결 URL로 "jdbc:default:connection:"을 사용하는 것이고, 두 번째는 cubrid.jdbc.driver.CUBRIDDriver 클래스의 getDefaultConnection() 메소드를 호출하는 것이다.

```
Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
Connection conn = DriverManager.getConnection("jdbc:default:connection:");
```

또는

```
cubrid.jdbc.driver.CUBRIDDriver.getDefaultConnection();
```

서버 측 JDBC Driver에서 위와 같은 방법으로 데이터베이스에 연결하면 Java 저장 함수/프로시저 내에 존재하는 트랜잭션 관련 사항이 무시된다. 즉, Java 저장 함수/프로시저에서 수행되는 데이터베이스 연산은 Java 저장 함수/프로시저를 호출한 트랜잭션에 포함된다는 것을 의미한다. 아래의 Athlete 클래스에서 conn.commit()은 무시된다.

```
import java.sql.*;
public class Athlete{
    public static void Athlete(String name, String gender, String nation_code, String
event) throws SQLException{
        String sql="INSERT INTO ATHLETE(NAME, GENDER, NATION CODE, EVENT)" + "VALUES
(?, ?, ?, ?)";
        try{
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
            Connection conn = DriverManager.getConnection("jdbc:default:connection:");
            PreparedStatement pstmt = conn.prepareStatement(sql);

            pstmt.setString(1, name);
            pstmt.setString(2, gender);
            pstmt.setString(3, nation_code);
            pstmt.setString(4, event);
            pstmt.executeUpdate();

            pstmt.close();
            conn.commit();
            conn.close();
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
}
```

```
}  
}
```

다른 데이터베이스 연결

서버 측 JDBC 드라이버를 사용하더라도 현재 연결된 데이터베이스를 사용하지 않고, 외부의 다른 데이터베이스에 연결할 수도 있다. 외부의 데이터베이스에 대한 Connection을 얻는 것은 일반적인 JDBC Connection과 다르지 않다. 이에 대한 자세한 내용은 JDBC API를 참조한다.

다른 데이터베이스에 연결하는 경우, Java 메소드의 수행이 종료되더라도 CUBRID 데이터베이스와의 Connection이 자동으로 종료되지 않는다. 따라서, 반드시 Connection 종료를 명시해주어야 **COMMIT**, **ROLLBACK**과 같은 트랜잭션 연산이 해당 데이터베이스에 반영된다. 즉, Java 저장 함수/프로시저를 호출한 데이터베이스와 실제 연결된 데이터베이스가 다르기 때문에 별도의 트랜잭션으로 수행되는 것이다.

```
import java.sql.*;  
  
public class SelectData {  
    public static void SearchSubway(String[] args) throws Exception {  
  
        Connection conn = null;  
        Statement stmt = null;  
        ResultSet rs = null;  
  
        try {  
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");  
            conn =  
DriverManager.getConnection("jdbc:CUBRID:localhost:33000:demodb:::", "", "");  
  
            String sql = "select line id, line from line";  
            stmt = conn.createStatement();  
            rs = stmt.executeQuery(sql);  
            while(rs.next()) {  
                int host year = rs.getString("host year");  
                String host nation = rs.getString("host nation");  
                System.out.println("Host Year ==> " + host year);  
                System.out.println(" Host Nation==> " + host nation);  
                System.out.println("\n\n=====\n");  
            }  
            rs.close();  
            stmt.close();  
            conn.close();  
        } catch ( SQLException e ) {  
            System.err.println(e.getMessage());  
        } catch ( Exception e ) {  
            System.err.println(e.getMessage());  
        } finally {  
            if ( conn != null ) conn.close();  
        }  
    }  
}
```

수행 중인 Java 저장 함수/프로시저가 데이터베이스 서버의 JVM에서만 구동되어야 할 때, Java 프로그램 소스에서 System.getProperty("cubrid.server.version")를 호출함으로써 어디서 수행되는 지를 점검할 수 있다. 결과 값은 데이터베이스에서 호출하면 데이터베이스 버전이 되고, 그 외는 **NULL**이 된다.

loadjava 유틸리티

설명

컴파일된 Java 파일이나 JAR(Java Archive) 파일을 CUBRID로 로딩하기 위해서 **loadjava** 유틸리티를 사용한다. **loadjava** 유틸리티를 사용하여 Java *.class 파일이나 *.jar 파일을 로딩하면 해당 파일이 해당 데이터베이스 경로로 이동한다.

구문

```
loadjava <option> database-name java-class-file
```

- *database-name* : Java 파일을 로드 하려고 하는 데이터베이스 이름
- *java-class-file* : 로드 하려는 Java 클래스 파일 이름 또는 jar 파일 이름
- *< option>* :
 - **-y** : 이름이 같은 클래스 파일이 존재하면 자동으로 덮어쓰기 한다. 디폴트는 **no**이다. 만약 **-y** 옵션을 명시하지 않고 로딩할 때 이름이 같은 클래스 파일이 존재하면 덮어쓰기를 할 것인지 묻는다.

로딩한 Java 클래스 등록

개요

CUBRID는 클라이언트나 SQL 문이나 Java 응용 프로그램에서 Java 메소드를 호출할 수 있도록 Java 클래스를 등록(publish)하는 과정이 필요하다. Java 클래스를 로딩했을 때 SQL 문이나 Java 응용 프로그램에서 클래스 내의 함수를 어떻게 호출할지 모르기 때문에 Call Specifications를 사용하여 등록해야 한다.

Call Specifications

CUBRID에서는 Java 저장 함수/프로시저를 사용하기 위해서는 Call Specifications를 작성해야 한다. Call Specifications는 Java 함수 이름과 인자 타입 그리고 리턴 값과 리턴 값의 타입을 SQL 문이나 Java 응용프로그램에서 접근할 수 있도록 해주는 역할을 한다. Call Specifications를 작성하는 구문은 **CREATE FUNCTION** 또는 **CREATE PROCEDURE** 구문을 사용하여 작성한다. Java 저장 함수/프로시저의 이름은 대소문자를 구별하지 않는다. Java 저장 함수/프로시저의 이름의 최대 길이는 256자이다. 또한 하나의 Java 저장 함수/프로시저가 가질 수 있는 인자의 최대 개수는 64개이다.

구문

```
CREATE {PROCEDURE procedure_name[(param[, param] ...)] | FUNCTION function_name[(param[, param]...)] RETURN sql_type }  
{IS | AS} LANGUAGE JAVA  
NAME 'method_fullname (java_type_fullname[,java_type_fullname]...) [return java_type_fullname]';  
  
parameter_name [IN|OUT|IN OUT|INOUT] sql_type  
(default IN)
```

Java 저장 함수/프로시저의 인자를 **OUT**으로 설정한 경우 길이가 1인 1차원 배열로 전달된다. 그러므로 Java 메소드는 배열의 첫번째 공간에 전달할 값을 저장해야 한다.

예제

```
CREATE FUNCTION Hello() RETURN VARCHAR
AS LANGUAGE JAVA
NAME 'SpCubrid.HelloCubrid() return java.lang.String';

CREATE FUNCTION Sp_int(i int) RETURN int
AS LANGUAGE JAVA
NAME 'SpCubrid.SpInt(int) return int';

CREATE PROCEDURE Athlete_Add(name varchar,gender varchar, nation_code varchar, event
varchar)
AS LANGUAGE JAVA
NAME 'Athlete.Athlete(java.lang.String, java.lang.String, java.lang.String,
java.lang.String)';

CREATE PROCEDURE test_out(x OUT STRING)
AS LANGUAGE JAVA
NAME 'SpCubrid.outTest(java.lang.String[] o)';
```

Java 저장 함수/프로시저를 등록할 때, Java 저장 함수/프로시저의 반환 정의와 Java 파일의 선언부의 반환 정의와의 일치 여부는 검사하지 않는다. 따라서, Java 저장 함수/프로시저의 경우 등록할 때의 *sql_type* 반환 정의를 따르고, Java 파일 선언부의 반환 정의는 사용자 정의 정보로서만 의미를 가지게 된다.

데이터 타입 매핑

Call Specifications에는 SQL의 데이터 타입과 Java의 매개변수와 리턴 값의 데이터 타입이 맞게 대응되어야 한다. CUBRID에서 허용되는 SQL과 Java의 데이터 타입의 관계는 다음의 표와 같다.

데이터 타입 매핑

SQL Type	Java Type
CHAR, VARCHAR	java.lang.String, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.math.BigDecimal, byte, short, int, long, float, double
NUMERIC, SHORT, INT, FLOAT, DOUBLE, CURRENCY	java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.math.BigDecimal, java.lang.String, byte, short, int, long, float, double
DATE, TIME, TIMESTAMP	java.sql.Date, java.sql.Time, java.sql.Timestamp, java.lang.String
SET, MULTISSET, SEQUENCE	java.lang.Object[], java primitive type array, java.lang.Integer[] ...
OBJECT	cubrid.sql.CUBRIDOID
CURSOR	cubrid.jdbc.driver.CUBRIDResultSet

등록된 Java 저장 함수/프로시저의 정보 확인

등록된 Java 저장 함수/프로시저의 정보는 **db_stored_procedure** 시스템 가상 클래스와

db_stored_procedure_args 시스템 가상 클래스에서 확인할 수 있다. **db_stored_procedure** 시스템 가상 클래스에서는 저장 함수/프로시저의 이름과 타입, 반환 타입, 인자의 수, Java 클래스에 대한 명세, Java 저장 함수/프로시저의 소유자에 대한 정보를 확인할 수 있다. **db_stored_procedure_args** 시스템 가상 클래스에서는 저장 함수/프로시저에서 사용하는 인자에 대한 정보를 확인할 수 있다.

```
csql> select * from db stored procedure
csql> ;xrun

=== <Result of SELECT Command in Line 2> ===

sp name      sp type      return type      arg count
sp name      sp type      return_type      arg_count  lang
target              owner
=====
'hello'      'FUNCTION'      'STRING'          0  'JAVA' 'SpCu
brid.HelloCubrid() return java.lang.String' 'DBA'

'sp int'      'FUNCTION'      'INTEGER'         1  'JAVA' 'SpCu
brid.SpInt(int) return int' 'DBA'

'athlete_add' 'PROCEDURE'      'void'            4  'JAVA' 'Athl
ete.Athlete(java.lang.String, java.lang.String, java.lang.String,
java.lang.String)' 'DBA'

3 rows selected.

Current transaction has been committed.

1 command(s) successfully processed.

csql> select * from db stored procedure args
csql> xrun

=== < Result of SELECT Command in Line 1> ===

sp_name      index_of  arg_name  data_type      mode
=====
'sp int'      0  'i'      'INTEGER'      'IN'
'athlete_add' 0  'name'   'STRING'       'IN'
'athlete_add' 1  'gender' 'STRING'       'IN'
'athlete_add' 2  'nation_code' 'STRING'      'IN'
'athlete_add' 3  'event'  'STRING'       'IN'

5 rows selected.

Current transaction has been committed.

1 command(s) successfully processed.
```

Java 저장 함수/프로시저의 삭제

CUBRID에서는 등록한 Java 함수/저장 프로시저를 삭제할 수 있다. **DROP FUNCTION** *function_name* 또는 **DROP PROCEDURE** *procedure_name* 구문을 사용하여 Java 저장 프로시저를 삭제할 수 있다. 또한 여러 개의 *function_name*이나 *procedure_name*을 콤마(,)로 구분하여 한꺼번에 여러 개의 Java 저장 함수/프로시저를 삭제할 수 있다.

Java 저장 함수/프로시저의 삭제는 Java 저장 함수/프로시저를 등록한 사용자와 DBA의 구성원만 삭제할 수 있다. 예를 들어 'sp_int' Java 저장 함수를 PUBLIC이 등록을 하였다고 한다면, **PUBLIC** 또는 **DBA**의 구성원만이 'sp_int' Java 저장 함수를 삭제할 수 있다.

```
drop function hello[, sp_int]
drop procedure Athlete_Add
```

Java 저장 함수/프로시저 호출

CALL 사용

등록된 Java 저장 함수/프로시저는 **CALL** 문을 사용하거나, SQL 문에서 호출하거나, Java 응용프로그램에서 호출될 수 있다.

다음과 같이 **CALL** 문을 사용하여 호출할 수 있다. **CALL** 문에서 호출되는 Java 저장 함수/프로시저의 이름은 대소문자를 구분하지 않는다.

구문

```
CALL {procedure_name ([param[, param]...]) | function_name ([param[, param]...])
INTO :host_variable
param {literal | :host_variable}
```

예제

```
call Hello() into :HELLO;
call Sp_int(3) into :i;
call phone_info('Tom','016-111-1111');
```

CUBRID에서는 Java 저장 함수/프로시저를 같은 **CALL** 문을 이용해 호출한다. 따라서 다음과 같이 **CALL** 문을 처리하게 된다.

- **CALL** 문에 대상 클래스가 있는 경우 메소드로 처리한다.
- **CALL** 문에 대상 클래스가 없는 경우 먼저 Java 저장 함수/프로시저 수행 여부를 검사하고 Java 저장 함수/프로시저가 존재하면 Java 저장 함수/프로시저를 수행한다.
- 2에서 Java 저장 함수/프로시저가 존재하지 않으면 메소드 수행 여부를 검사하여 같은 이름이 존재하면 수행한다.

만약 존재하지 않는 Java 저장 함수/프로시저를 호출하는 경우에는 다음과 같은 에러가 나타난다.

```
csql> call deposit()
csql> ;xrun

In the command from line 1,

ERROR: Stored procedure/function 'deposit' is not exist.

0 command(s) successfully processed.

csql> call deposit('Tom', 3000000)
csql> ;xrun

In line 1, column 6,

ERROR: Methods require an object as their target.
```

```
0 command(s) successfully processed.
```

CALL 문에 인자가 없는 경우는 메소드와 구분되므로 "ERROR: Stored procedure/function 'deposit' is not exist."라는 오류 메시지가 나타난다. 하지만, **CALL** 문에 인자가 있는 경우에는 메소드와 구분할 수 없기 때문에 "ERROR: Methods require an object as their target."이라는 메시지가 나타난다.

그리고, 아래와 같이 Java 저장 함수/프로시저를 호출하는 **CALL** 문 안에 **CALL** 문이 중첩되는 경우와 **CALL** 문을 사용하여 Java 저장 함수/프로시저 호출 시 인자로 서브 질의를 사용할 경우 **CALL** 문은 수행이 되지 않는다.

```
call phone info('Tom', call sp_int(999));
call phone_info((select * from Phone where id='Tom'));
```

Java 저장 함수/프로시저를 호출하여 수행 중 exception이 발생하면 **dbname.java.log** 파일에 exception 내용이 기록되어 저장된다. 만약 화면으로 exception 내용을 확인하고자 할 경우는

\$CUBRID/java/logging.properties 파일의 handlers 값을 " java.lang.logging.ConsoleHandler" 로 수정하면 화면으로 exception 내용을 출력한다.

SQL 문에서 호출

다음과 같이 SQL 문에서 Java 저장 함수를 호출하여 사용할 수 있다.

```
select Hello() from db_root;
select sp_int(99) from db_root;
```

Java 저장 함수/프로시저를 호출할 때 IN/OUT의 데이터 타입에 호스트 변수를 사용할 수 있으며, 사용 예는 다음과 같다.

```
SELECT 'Hi' INTO :out_data FROM db_root;
CALL test out(:out_data);
SELECT :out_data FROM db_root;
```

첫 번째 문장은 파라미터 변수를 이용하여 out 모드의 Java 저장 프로시저를 호출하는 예이고, 두 번째 문장은 할당된 호스트 변수 out_data를 조회하는 질의문이다.

Java 응용 프로그램에서 호출

Java 응용 프로그램에서 Java 저장 함수/프로시저를 호출하기 위해서는 **CallableStatement**를 사용한다.

CUBRID 데이터베이스에 Phone 클래스를 생성한다.

```
create class phone(
    name varchar(20),
    phoneno varchar(20)
)
```

다음의 PhoneNumber.java Java 파일을 컴파일하여 Java 클래스 파일을 CUBRID로 로드하고 등록한다.

```
import java.sql.*;
import java.io.*;

public class PhoneNumber{
    public static void Phone(String name, String phoneno) throws Exception{
        String sql="INSERT INTO PHONE(NAME, PHONENO)+"+"VALUES (?, ?)";
        try{
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
```

```

        Connection conn = DriverManager.getConnection("jdbc:default:connection:");
        PreparedStatement pstmt = conn.prepareStatement(sql);

        pstmt.setString(1, name);
        pstmt.setString(2, phoneno);
        pstmt.executeUpdate();

        pstmt.close();
        conn.commit();
        conn.close();
    } catch (SQLException e) {
        System.err.println(e.getMessage());
    }
}

}

create PROCEDURE phone_info(name varchar, phoneno varchar)
as language java
name 'PhoneNumber.Phone(java.lang.String, java.lang.String)';

```

다음과 같은 Java 응용 프로그램을 작성하고 실행한다.

```

import java.sql.*;

public class StoredJDBC{
    public static void main(){
        Connection conn = null;
        Statement stmt= null;
        int result;
        int i;

        try{
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
            conn =
            DriverManager.getConnection("jdbc:CUBRID:localhost:33000:subway::","","");

            CallableStatement cs;
            cs = conn.prepareCall("call PHONE_INFO(?, ?)");

            cs.setString(1, "Jane");
            cs.setString(2, "010-1111-1111");
            cs.executeUpdate();

            conn.commit();
            cs.close();
            conn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

위의 프로그램 실행한 후 PHONE 클래스 조회를 하면 다음과 같은 결과가 출력된다.

```

csql> select * from phone
csql> ;xrun

=== <Result of SELECT Command in Line 1>===

name                phoneno
=====
'Jane'              '010-111-1111'

1 rows selected.

Current transaction has been committed.

1 command(s) successfully processed.

```

주의 사항

Java 저장 함수/프로시저의 리턴 값 및 IN/OUT 에 대한 타입 자릿수

Java 저장 함수/프로시저의 리턴 값과 IN/OUT의 데이터 타입에 자릿수를 한정하는 경우, CUBRID에서는 다음과 같이 처리한다.

Java 저장 함수/프로시저의 sql_type을 기준으로 확인한다.

Java 저장 함수/프로시저 생성 시 정의한 자릿수는 무시하고 타입만 맞추어 Java에서 반환하는 값을 그대로 데이터베이스에 전달한다. 전달한 데이터에 대한 조작은 사용자가 데이터베이스에서 직접 처리하는 것을 원칙으로 한다.

다음과 같은 **typestring()** Java 저장 함수를 살펴보자.

```
public class JavaSP1{
    public static String typestring(){
        String temp = " ";
        for(int i=0 i< 1 i++)
            temp = temp + "1234567890";
        return temp;
    }
}

create function typestring() return char(5)
as language java
name 'JavaSP1.typestring()' return java.lang.String';

csql> call typestring()
csql> ;xrun

=== < Result of CALL Command in Line 1> ===

    Result
=====
    ' 1234567890'

Current transaction has been committed.

1 command(s) successfully processed.
```

Java 저장 프로시저에서의 java.sql.ResultSet 반환

CUBRID에서는 **java.sql.ResultSet**을 반환하는 Java 저장 함수/프로시저를 선언할 때는 데이터 타입으로 **CURSOR**를 사용해야 한다.

```
create function rset() return cursor
as language java
name 'JavaSP2.TResultSet()' return java.sql.ResultSet'
```

Java 파일에서는 **java.sql.ResultSet**을 반환하기 전에 **CUBRIDResultSet** 클래스로 캐스팅 후 **setReturnable()** 메소드를 호출해야 한다.

```
public static class JavaSP2 {
public static ResultSet TResultSet(){
    try{
        Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
        Connection conn = DriverManager.getConnection("jdbc:default:connection:");
        ((CUBRIDConnection)conn).setCharset("euc kr");
        String sql = "select * from station";
        Statement stmt=con.createStatement();
```

```

        ResultSet rs = stmt.executeQuery(sql);
        ((CUBRIDResultSet)rs).setReturnable();
        return rs;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
}

```

호출하는 쪽에서는 **Types.JAVA_OBJECT**로 OUT 인자를 설정하고 **getObject()** 함수로 가져온 후 **java.sql.ResultSet**으로 변환(Casting)하여 사용해야 한다. 또한, **java.sql.ResultSet**은 JDBC의 **CallableStatement**에서만 사용할 수 있다.

```

import java.sql.*;

public class TestResultSet{
    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt= null;
        int result;
        int i;

        try{
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
            conn = DriverManager.getConnection("jdbc:CUBRID:localhost:33000:demodb::","","");

            CallableStatement cstmt = con.prepareCall("?=CALL rset()");
            cstmt.registerOutParameter(1, Types.JAVA_OBJECT);
            cstmt.execute();
            ResultSet rs = (ResultSet) cstmt.getObject(1);
            while(rs.next()) {
                System.out.println(rs.getString(1));
            }
            rs.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

ResultSet은 입력 인자로 사용할 수 없으며, 이를 IN 인자로 전달할 경우에는 에러가 발생한다. Java가 아닌 환경에서 **ResultSet**을 반환하는 함수를 호출할 경우에도 에러가 발생한다.

Java 저장 함수/프로시저에서 Set 타입의 IN/OUT

CUBRID의 Java 저장 함수/프로시저에서 Set 타입이 IN OUT인 경우 Java에서 인자 값을 변경할 경우 변경 값이 전달이 되도록 Set 타입이 OUT 인자로 전달될 때는 2차원 배열로 전달하도록 해야 한다.

```

Create procedure setoid(x in out set, z object)
as language java name
'SetOIDTest.SetOID(cubrid.sql.CUBRIDOID[][], cubrid.sql.CUBRIDOID';

public static void SetOID(cubrid.sql.CUBRID[][] set, cubrid.sql.CUBRIDOID aoid){
    Connection conn=null;
    Statement stmt=null;
    String ret="";
    Vector v = new Vector();
    cubrid.sql.CUBRIDOID[] set1 = set[0];
    try {
        if(set1!=null) {
            int len = set1.length;
            int i = 0;
            for (i=0 i< len i++)
                v.add(set1[i]);
        }
        v.add(aoid);
    }
}

```



```

set[0]=(cubrid.sql.CUBRIDOID[]) v.toArray(new cubrid.sql.CUBRIDOID[]{});
} catch(Exception e) {
    e.printStackTrace();
    System.err.println("SQLException:"+e.getMessage());
}
}
}

```

Java 저장 함수/프로시저에서 OID 사용

CUBRID 저장 프로시저에서 OID 타입의 값을 IN/OUT으로 사용할 경우 서버의 값을 전달 받아 사용한다.

```

create procedure tOID(i inout object, q string)
as language java
name 'OIDtest.tOID(cubrid.sql.CUBRIDOID[], java.lang.String)';

public static void tOID(CUBRIDOID[] oid, String query)
{
    Connection conn=null;
    Statement stmt=null;
    String ret="";

    try {
        Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
        conn=DriverManager.getConnection("jdbc:default:connection:");

        conn.setAutoCommit(false);
        stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        System.out.println("query:"+ query);

        while(rs.next()) {
            oid[0]=(CUBRIDOID)rs.getObject(1);
            System.out.println("oid:"+oid[0].getTableName());
        }
        stmt.close();
        conn.close();
    } catch (SQLException e) {
        e.printStackTrace();
        System.err.println("SQLException:"+e.getMessage());
    } catch (Exception e) {
        e.printStackTrace();
        system.err.println("Exception:"+ e.getMessage());
    }
}

```

메소드(METHOD)

개요

이 장에서는 큐브리드 데이터베이스 시스템의 기능을 확장하거나 사용자에게 맞게 조정하는 소프트웨어 루틴인 메소드에 대해 기술한다.

메소드는 C로 작성된 프로그램이고, **CALL** 문이나 **EVALUATE** 문에 의해 호출된다. 메소드 프로그램은 메소드가 호출되었을 때 동적 로더에 의해 실행 중인 응용과 함께 로드(load)되고 연결(link)된다. 메소드 실행 결과 생성된 리턴 값(return value)은 호출자(caller)에게 전달된다.

이 장에서는 다음 내용에 대해 기술한다.

- 메소드의 타입
- 메소드 호출하기

메소드 타입

CSQL 언어는 클래스 메소드와 인스턴스 메소드 두 가지 타입의 메소드를 지원한다.

- **클래스 메소드**는 클래스 객체에서 호출되는 메소드이다. 일반적으로 클래스의 새로운 인스턴스를 생성하거나 초기화 하기 위하여 사용된다. 또한 클래스 속성에 접근하거나 갱신하기 위해서도 사용될 수 있다.
- **인스턴스 메소드**는 클래스의 인스턴스에서 호출되는 메소드이다. 대부분의 연산들이 인스턴스에서 수행되기 때문에 클래스 메소드보다 더 자주 사용된다. 예를 들어 인스턴스 메소드는 인스턴스의 속성을 계산하거나 갱신하기 위해 작성될 수 있다. 이 메소드는 메소드가 정의된 클래스의 어떤 인스턴스에서도 호출될 수 있고, 메소드를 상속 받은 어떠한 서브클래스의 인스턴스에서도 호출 할 수 있다.

메소드에 대한 상속 법칙은 속성에 대한 상속 법칙과 비슷하다. 서브클래스는 슈퍼클래스(superclass)로부터 클래스와 인스턴스 메소드를 상속 받는다. 서브클래스는 슈퍼클래스로부터 클래스나 인스턴스 메소드의 정의의 이름만을 가지고 있다.

메소드 이름에 대한 충돌 해결 규칙은 속성 이름에 대한 충돌 해결 규칙과 같다. 속성과 메소드 상속 충돌에 대한 추가적인 정보는 [클래스 충돌 해결](#)을 참조한다.

메소드 호출

개요

메소드는 **CALL** 또는 **EVALUATE** 문의 사용에 의해서 수행되며, 메소드의 결과는 질의의 결과가 반환되는 것과 같은 방법으로 반환된다.

이러한 구문은 질의로부터 메소드를 호출할 때도 사용된다(**CALL** 또는 **EVALUATE** 키워드는 생략된다).

CALL 문

설명

CUBRID **CALL** 문은 데이터베이스에 정의된 메소드를 호출하기 위해 사용된다. 클래스 메소드, 인스턴스 메소드 모두 **CALL** 문으로 호출이 가능하다.

구문

```
CALL method_call [ ; ]

method_call :
• method name ( [ arg value [ { , arg value } ] ] ) ON call target [ to variable ]
• method name ( call target [ , arg value [ { , arg value } ] ] ) [ to variable ]

arg value :
• any CSQL expression

call target :
• an object-valued expression

to variable :
• INTO variable
• TO variable
```

- *method name*은 클래스에 정의된 메소드의 이름이거나, 큐브리드와 함께 제공되는 시스템 정의 메소드의 이름이다. 메소드는 하나 혹은 그 이상의 인수 값을 필요로 한다. 메소드에 인수가 없으면 빈 괄호를 사용해야 한다.
- *call target*은 클래스 이름, 변수, 또 다른 메소드 호출(객체를 반환하는)을 포함하는 객체 값을 나타내는 식(object-valued expression)을 사용할 수 있다. 만약 클래스 객체에서 동작하는 클래스 메소드를 호출하려면, *call target* 앞에 반드시 **CLASS** 키워드가 있어야 한다. 이러한 경우에 클래스 이름은 클래스 메소드가 정의된 클래스의 이름이어야 한다. 만약 인스턴스 메소드를 호출하려면, 인스턴스 객체를 나타내는 식을 지정해야 한다. 클래스 메소드나 인스턴스 메소드에 의해 반환되는 값은 선택적으로 *to variable*에 저장할 수 있다. 이 반환 변수의 값은 *call target*이나 *arg value* 파라미터처럼 **CALL** 문 내에 사용될 수 있다.
- 중첩된 메소드 호출은 다른 *method_call*이 메소드의 *call target*이거나 *arg value* 인수의 하나로 주어질 때 성립된다.

EVALUATE 문

설명

EVALUATE 문 또한 데이터베이스에 정의된 메소드를 호출하는데 사용된다.

EVALUATE 문에서 메소드 호출은 표현식의 용어(*term*)로 주어진다. 메소드의 결과가 상수 값이면, 다른 상수(또는 상수를 반환하는 메소드) 또한 표현식의 용어로 주어질 수 있다. 클래스 메소드와 인스턴스 메소드 모두 **EVALUATE** 문으로 호출할 수 있다.

구문

```
EVALUATE expression [ ; ]

expression:
```

```

• [ + | - ] term [ { + | - | * | / } term ]

term:
• method call

method call :
• method_name ( call_target [ , arg_value [ { , arg_value } _ ] ] ) [ to_variable ]
    method name ( [ arg_value [ { , arg_value } _ ] ] )
    ON call_target [ to_variable ]

arg_value :
• literal
• variable
• expression

call_target :
• CLASS class name
• variable
• expression
• method call

to_variable :
• INTO variable
• TO variable

```

EVALUATE 문에서 지명된 메소드를 위한 대상 인수는 *method_name* 뒤에 괄호 안에 기술된다. 대상은 리스트에서 첫 번째로 주어질 수 있으며, 메소드 인수들이 뒤따른다. 수행된 메소드가 클래스 메소드이면, **CLASS** 키워드가 반드시 리스트의 첫 항목으로써 목표 클래스 앞에 놓여야 한다. 괄호가 메소드 인수만 포함하고 있을 때 *call_target*은 반드시 **ON** 절에 주어진다.

EVALUATE 문 또한 한 메소드 호출이 또 다른 메소드의 대상이나 인수로 기술함으로써 중첩된 메소드 호출을 허용한다. 이러한 유형의 기술은 외부(outer) 메소드의 결과를 결정하기 위하여 내부(inner) 메소드의 결과를 사용한다.

분할

분할이란?

분할 기법(partitioning)은 하나의 테이블을 여러 독립적인 논리적 단위로 분할하는 기법이다. 분할 기법에서 사용하는 분할된 단위를 분할(partition)이라 한다. 분할은 주로 관리의 편의(manageability), 성능(performance), 가용성(availability)의 목적으로 사용한다. 분할을 적용함으로써 얻을 수 있는 효과는 다음과 같다.

- 대용량 테이블의 관리 향상
- 데이터 조회 시 접근 범위를 줄임으로써 성능 향상
- 디스크 I/O를 분산함으로써 성능 향상 및 물리적 부하 감소
- 여러 분할로 나눔으로써 전체 데이터의 훼손 가능성 감소 및 가용성 향상
- 스토리지 비용의 최적화

CUBRID는 영역 분할(Range Partitioning), 해시 분할(Hash Partitioning), 리스트 분할(List Partitioning)의 세 가지 분할을 제공한다.

한 테이블이 가질 수 있는 최대 분할 수는 1024이다. 테이블의 각 분할은 그 테이블의 서브 테이블로 생성된다. 분할 정의를 통해 생성된 서브 테이블은 사용자가 임의로 내용을 변경하거나 삭제할 수 없다. 서브 테이블의 이름은 'class_name_p_partition_name'의 형식으로 시스템 테이블에 등록된다.

데이터베이스 사용자는 db_class 뷰와 db_partition 뷰에서 분할의 정보를 확인할 수 있다. 또 다른 확인 방법은 CUBRID 매니저나 CSQL 인터프리터의 ;sc <테이블명> 명령을 사용하는 것이다.

영역 분할

영역 분할 정의

설명

영역 분할은 **PARTITION BY RANGE** 문을 이용하여 정의한다.

구문

```
CREATE TABLE (
...
)
PARTITION BY RANGE ( <partition expression> ) (
PARTITION <partition name> VALUES LESS THAN ( <range value> ),
PARTITION <partition_name> VALUES LESS THAN ( <range_value> ) ),
... )
)
```

- *partition_expression* : 분할 표현식을 지정한다. 표현식은 분할 대상이 되는 컬럼 명을 지정하거나 함수를 사용하여 지정할 수 있다. 사용 가능한 데이터 타입과 함수에 대한 자세한 설명은 [분할 표현식에 사용할 수 있는 데이터 타입](#)을 참조한다.
- *partition_name* : 분할 명을 지정한다.
- *range_value* : 분할의 기준이 되는 값을 지정한다.

예제 1

다음은 올림픽 참가국 정보를 담은 participant2 테이블을 생성하고 참가한 올림픽의 개최연도를 2000년도 전/후로 영역 분할하는 데이터를 삽입하는 예제이다. 데이터 삽입 시 88년, 96년 올림픽에 참가한 국가는 before_2000에, 나머지 국가는 before_2008에 저장된다.

```
CREATE TABLE participant2 (host year INT, nation CHAR(3), gold INT, silver INT, bronze INT)
PARTITION BY RANGE (host_year)
(PARTITION before_2000 VALUES LESS THAN (2000),
PARTITION before_2008 VALUES LESS THAN (2008) );

INSERT INTO participant2 VALUES (1988, 'NZL', 3, 2, 8);
INSERT INTO participant2 VALUES (1988, 'CAN', 3, 2, 5);
INSERT INTO participant2 VALUES (1996, 'KOR', 7, 15, 5);
INSERT INTO participant2 VALUES (2000, 'RUS', 32, 28, 28);
INSERT INTO participant2 VALUES (2004, 'JPN', 16, 9, 12);
```

예제 2

다음과 같이 영역 분할에서 분할 키 값이 NULL이면 첫 번째 분할에 저장된다.

```
INSERT INTO participant2 VALUES(NULL, 'AAA', 0, 0, 0);
```

주의 사항

- 한 테이블이 가질 수 있는 최대 분할 개수는 1024이다.
- 분할 키 값이 NULL이면, 첫 번째 분할에 저장된다(예제 2 참조).

영역 분할 재정의

설명

ALTER 문의 **REORGANIZE PARTITION** 절을 이용하여 분할을 재정의한다. 재정의의 통해 복수개의 분할을 1개에 결합할 수 있으며, 1개의 분할을 복수개로 분리할 수 있다.

구문

```
ALTER {TABLE | CLASS} <table_name>
REORGANIZEPARTITION
<alter partition name comma list>
INTO ( <partition definition comma list> )

partitiondefinition comma list:
PARTITION <partition_name> VALUES LESS THAN ( <range_value> ),...
```

- *table_name* : 재정의할 테이블의 이름을 지정한다.
- *alter partition name comma list* : 재정의할 분할을 지정한다. 복수개인 경우 쉼표(,) 구분한다.
- *partition definition comma list* : 재정의 내용을 정의한다. 복수개인 경우 쉼표(,)로 구분한다.

예제 1

다음은 participant2 테이블의 before_2000 분할을 before_1996과 before_2000으로 재분할하는 예제이다.

```
CREATE TABLE participant2 ( host_year INT, nation CHAR(3), gold INT, silver INT, bronze INT)
PARTITION BY RANGE (host_year)
( PARTITION before_2000 VALUES LESS THAN (2000),
  PARTITION before_2008 VALUES LESS THAN (2008) );

ALTER TABLE participant2 REORGANIZE PARTITION before_2000 INTO (
PARTITION before_1996 VALUES LESS THAN (1996),
PARTITION before_2000 VALUES LESS THAN (2000)
);
```

예제 2

다음은 예제 1에서 재정의했던 분할을 다시 before_2000 하나로 결합하는 예제이다.

```
ALTER TABLE participant2 REORGANIZE PARTITION before_1996, before_2000 INTO
(PARTITION before_2000 VALUES LESS THAN (2000) );
```

주의 사항

- 영역 및 리스트 분할 테이블을 재정의할 때, 새로운 분할 스키마에는 중복된 영역이나 값은 허용되지 않는다.
- **REORGANIZE PARTITION** 절을 사용해 테이블의 분할 종류를 변경할 수 없다. 예를 들어, 영역 분할을 해시 분할로 변경할 수 없으며, 그 반대도 마찬가지이다.
- 분할 추가 후 최대 분할의 개수는 1,024개를 넘지 못하며, 분할 삭제 후 최소 1개 이상의 분할이 남아 있어야 한다. 영역 분할 테이블은 인접한 분할만 재정의할 수 있다.

영역 분할 추가

설명

ALTER 구문의 **ADD PARTITION** 절을 이용하여 분할된 테이블에 분할을 추가한다.

구문

```
ALTER {TABLE | CLASS} <table_name>
ADD PARTITION <partition definitions comma list>
partition definition comma list:
PARTITION <partition_name> VALUES LESS THAN ( <range_value> ),...
```

- *table_name*: 분할을 추가할 테이블의 이름을 지정한다.
- *partition definition comma list*: 추가할 분할을 정의한다. 복수개의 경우 쉼표(,)로 구분한다.

예제

현재 participant2 테이블에는 2008년 이전 올림픽 정보에 관한 분할만 정의되어 있다. 다음은 2012년 올림픽 정보가 저장될 before_2012 분할과 2016년 올림픽 정보가 저장될 before_2016 분할을 추가하는 예제이다.

```
ALTER TABLE participant2 ADD PARTITION (
PARTITION before_2012 VALUES LESS THAN (2012),
PARTITION before_2016 VALUES LESS THAN MAXVALUE );
```

주의 사항

- 영역 분할을 추가할 때는 분할 기준 값이 기존의 분할보다 큰 값만 추가할 수 있다. 따라서, 위의 예제처럼 **MAXVALUE**로 최대값을 설정하면 더 이상 분할을 추가할 수 없다(분할 재정의의 통해서 **MAXVALUE**를 다른 값으로 변경하면 분할 추가 가능).
- 기존의 분할보다 작은 분할 기준 값을 추가하려면 분할 재정의의 이용한다([영역 분할 재정의](#) 참조).

영역 분할 삭제

설명

ALTER 구문의 **DROP PARTITION** 절을 이용하여 분할을 삭제한다.

구문

```
ALTER {TABLE | CLASS} <table_name>  
DROP PARTITION <partition_name>
```

- table_name*: 분할된 테이블의 이름을 지정한다.
- partition_name*: 삭제할 분할의 이름을 지정한다.

예제

다음은 participant2 테이블의 before_2000 분할을 삭제한다.

```
ALTER TABLE participant2 DROP PARTITION before_2000;
```

주의 사항

- 분할된 테이블을 삭제하면 해당 분할 내에 저장된 데이터도 모두 삭제된다.
- 데이터는 유지한 채 테이블의 분할을 변경하는 경우 **ALTER TABLE...REORGANIZE PARTITION** 문을 사용한다([영역 분할 재정의](#) 참조).
- 분할을 삭제할 경우 삭제된 행의 수를 반환하지 않는다. 테이블과 분할을 유지한 채로 데이터만 삭제하고 싶은 경우 **DELETE** 문을 수행한다.

해시 분할

해시 분할 정의

설명

해시 분할은 **PARTITION BY HASH** 문을 이용하여 정의한다.

구문

```
CREATE TABLE (  
...  
)  
( PARTITION BY HASH ( <partition_expression> )  
PARTITIONS ( <number_of_partitions> )  
)
```


- *partition_expression*: 분할 표현식을 지정한다. 표현식은 분할 대상이 되는 컬럼 이름이나 함수를 사용하여 지정할 수 있다.
- *number_of_partitions*: 원하는 분할의 수를 지정한다.

예제 1

다음은 국가 코드와 국가 이름의 정보를 담은 nation2 테이블을 생성하고 코드 값을 기준으로 4개의 해시 분할을 정의하는 예제이다. 해시 분할은 분할의 수만 지정하고 이름은 지정하지 않으므로 p0, p1과 같이 자동으로 이름이 부여된다.

```
CREATE TABLE nation2
( code CHAR(3),
  name VARCHAR(50) )
PARTITION BY HASH ( code) PARTITIONS 4;
```

예제 2

다음은 예제 1에서 생성한 해시 분할에 데이터를 삽입하는 예제이다. 해시 분할에 값을 입력하면 분할 키의 해시 값에 따라 저장될 분할이 결정된다. 해시 분할에서 분할키 값이 NULL이면 첫 번째 분할에 저장된다.

```
INSERT INTO nation2 VALUES ('KOR','Korea');
INSERT INTO nation2 VALUES ('USA','USA United States of America');
INSERT INTO nation2 VALUES ('FRA','France');
INSERT INTO nation2 VALUES ('DEN','Denmark');
INSERT INTO nation2 VALUES ('CHN','China');
INSERT INTO nation2 VALUES (NULL,'AAA');
```

주의 사항

- 한 테이블이 가질 수 있는 최대 분할 개수는 1024이다.

해시 분할 재정의

설명

ALTER 문의 **COALESCE PARTITION** 절을 이용하여 재정의할 수 있다. 해시 분할이 재정의되는 경우 인스턴스는 그대로 보존된다.

구문

```
ALTER {TABLE | CLASS} <table_name>
COALESCE PARTITION <unsigned integer>
```

- *table_name*: 재정의할 테이블의 이름을 지정한다.
- *unsigned integer*: 삭제하려는 분할의 개수를 지정한다.

예제

다음은 nation2 테이블의 분할의 개수를 4개에서 2개로 줄이는 예제이다.

```
ALTER TABLE nation2 COALESCE PARTITION 2;
```

주의 사항

- 분할의 개수를 감소시키는 재편성 결합만 가능하다.
- 분할의 수를 늘리고자 하는 경우에는 영역 분할에서와 같은 **ALTER TABLE ... ADD PARTITION** 구문을 이용한다(자세한 내용은 [영역 분할 추가](#) 참조).
- 분할 재정의 후에 최소 1개 이상의 분할이 남아 있어야 한다.

리스트 분할

리스트 분할 정의

설명

리스트 분할은 **PARTITION BY LIST** 문을 이용하여 정의한다.

구문

```
CREATE TABLE (  
...  
)  
PARTITION BY LIST ( <partition_expression> ) (  
PARTITION <partition_name> VALUES IN ( <partition_value_list> ),  
PARTITION <partition_name> VALUES IN ( <partition_value_list>  
,  
...  
);
```

- *partition_expression*: 분할 표현식을 지정한다. 표현식은 분할 대상이 되는 컬럼 명을 지정하거나 함수를 사용하여 지정할 수 있다. 사용 가능한 데이터 타입과 함수에 대한 자세한 내용은 [분할 표현식에 사용할 수 있는 데이터 타입](#)을 참조한다.
- *partition_name*: 분할 명을 지정한다.
- *partition_value_list*: 분할의 기준이 되는 값의 목록을 지정한다.

예제 1

다음은 선수의 이름과 종목 정보를 담고있는 athlete2 테이블을 생성하고 종목에 따른 리스트 분할을 정의하는 예제이다.

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )  
PARTITION BY LIST (event) (  
PARTITION event1 VALUES IN ('Swimming', 'Athletics' ),  
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),  
PARTITION event3 VALUES IN ('Football', 'Basketball',  
'Baseball')  
);
```

예제 2

다음은 예제 1에서 생성한 리스트 분할에 데이터를 삽입하는 예제이다. 마지막 질의와 같이 데이터 삽입 시 분할 표현식에서 기술하였던 리스트에 없는 값으로 삽입하는 경우에는 삽입이 이루어지지 않는다.

```
INSERT INTO athlete2 VALUES ('Hwang Young-Cho', 'Athletics');  
INSERT INTO athlete2 VALUES ('Lee Seung-Yuop', 'Baseball');  
INSERT INTO athlete2 VALUES ('Moon Dae-Sung', 'Taekwondo');
```

```
INSERT INTO athlete2 VALUES ('Cho In-Chul', 'Judo');
INSERT INTO athlete2 VALUES ('Hong Kil-Dong', 'Volleyball');
```

예제 3

다음은 분할키 값이 **NULL**인 경우에 삽입이 이루어지지 않고 에러가 발생함을 보여주는 예제이다.

NULL 값을 삽입 가능하도록 분할을 정의하려면 두 번째 코드와 같이 **NULL** 값을 리스트로 갖는 분할을 정의하면 된다.

```
INSERT INTO athlete2 VALUES ('Hong Kil-Dong', 'NULL');

CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo', 'Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball',
'Baseball', NULL)
);
```

주의 사항

- 한 테이블이 가질 수 있는 최대 분할 개수는 1024이다.

리스트 분할 재정의

설명

ALTER 문의 **REORGANIZE PARTITION** 절을 이용하여 재정의할 수 있다. 재정의의를 통해 복수의 분할을 1개에 결합할 수 있으며, 1개의 분할을 복수개로 분리할 수 있다.

구문

```
ALTER {TABLE | CLASS} <table_name>
REORGANIZEPARTITION
<alter partition name comma list>
INTO ( <partition definition comma list> )
partition definition comma list:
PARTITION <partition_name> VALUES IN ( <partition_value_list>),...
```

- *table_name*: 재정의할 테이블의 이름을 지정한다.
- *alter partition name comma list*: 재정의할 분할을 지정한다. 복수개의 경우 쉼표(,)로 구분한다.
- *partition definition comma list*: 재정의 내용을 정의한다. 복수개의 경우 쉼표(,)로 구분한다.

예제 1

다음은 종목에 따라 리스트 분할한 athlete2 테이블을 생성하고 분할 event2를 event2_1(유도), event2_2(태권도, 복싱)로 재정의하는 예제이다.

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo', 'Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')
);

ALTER TABLE athlete2 REORGANIZE PARTITION event2 INTO
(PARTITION event2_1 VALUES IN ('Judo'),
```

```
PARTITION event2_2 VALUES IN ( 'Taekwondo','Boxing'));
```

예제 2

다음은 예제 1에서 분할한 event2_1과 event2_2를 다시 event2 하나로 결합하는 예제이다.

```
ALTER TABLE athlete2 REORGANIZE PARTITION event2_1, event2_2 INTO  
(PARTITION event2 VALUES IN('Judo','Taekwondo','Boxing'));
```

리스트 분할 삭제

설명

ALTER 구문의 **DROP PARTITION** 절을 이용하여 분할을 삭제할 수 있다.

구문

```
ALTER {TABLE | CLASS} <table_name>  
DROP PARTITION <partition_name>
```

- *table_name* : 분할된 테이블의 이름을 지정한다.
- *partition_name* : 삭제할 분할의 이름을 지정한다.

예제

다음은 종목에 따라 리스트 분할한 athlete2 테이블을 생성하고 event3 분할을 삭제하는 예제이다.

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )  
PARTITION BY LIST (event) (  
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),  
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),  
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')  
);  
  
ALTER TABLE athlete2 DROP PARTITION event3;
```

분할 관리

분할에서 데이터 조회와 조작

설명

데이터를 조회할 때에는 분할 테이블뿐만 아니라 각 분할에 대해서도 **SELECT** 문을 이용하여 조회가 가능하다.

예제

다음은 종목에 따라 리스트 분할한 athlete2 테이블을 생성하고 데이터를 삽입한 뒤 event1 분할과 event2 분할을 조회하는 예제이다.

```
CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )  
PARTITION BY LIST (event) (  
PARTITION event1 VALUES IN ('Swimming', 'Athletics ' ),  
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),  
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')  
);
```

```

INSERT INTO athlete2 VALUES ('Hwang Young-Cho', 'Athletics');
INSERT INTO athlete2 VALUES ('Lee Seung-Yuop', 'Baseball');
INSERT INTO athlete2 VALUES ('Moon Dae-Sung', 'Taekwondo');
INSERT INTO athlete2 VALUES ('Cho In-Chul', 'Judo');
csql> select * from athlete2__p__event1;
csql> ;x
=== <Result of SELECT Command in Line 1> ===
      name                event
=====
      'Hwang Young-Cho'    'Athletics'

1 rows selected.
csql> select * from athlete2__p__event2;
csql> ;x
=== <Result of SELECT Command in Line 1> ===
      name                event
=====
      'Moon Dae-Sung'      'Taekwondo'
      'Cho In-Chul'       'Judo'

2 rows selected.

```

주의 사항

- 분할 테이블의 각 분할에 대해서 데이터 삽입, 갱신, 삭제 등 데이터 조작은 불가능하다.

분할 키 값의 변경에 의한 데이터 이동

설명

분할의 분할 키 값이 변경되면 변경된 인스턴스는 분할 표현식에 의해서 다른 분할로 이동할 수 있다.

예제

다음은 분할 키 값이 변경되어 인스턴스가 다른 분할로 이동하는 것을 보여주는 예제이다.

event1 분할에 저장되어 있는 황영조 선수의 종목 정보를 athletics에서 Football로 바꾸면 인스턴스가 event3 분할로 이동된다.

```

CREATE TABLE athlete2( name VARCHAR(40), event VARCHAR(30) )
PARTITION BY LIST (event) (
PARTITION event1 VALUES IN ('Swimming', 'Athletics' ),
PARTITION event2 VALUES IN ('Judo', 'Taekwondo','Boxing'),
PARTITION event3 VALUES IN ('Football', 'Basketball', 'Baseball')
);

INSERT INTO athlete2 VALUES ('Hwang Young-Cho', 'Athletics');
INSERT INTO athlete2 VALUES ('Lee Seung-Yuop', 'Baseball');
csql> SELECT * FROM athlete2__p__event1;
csql> ;x
=== <Result of SELECT Command in Line 1> ===
      name                event
=====
      'Hwang Young-Cho'    'Athletics'

1 rows selected.
csql> UPDATE athlete2 SET event = 'Football' WHERE name = 'Hwang Young-Cho';
csql> ;x
1 rows affected.
csql> SELECT * FROM athlete2__p__event3;
csql> ;x
=== <Result of SELECT Command in Line 1> ===
      name                event
=====

```

```
'Lee Seung-Yuop'      'Baseball'
'Hwang Young-Cho'     'Football'

2 rows selected.
```

주의 사항

- 분할 키 값의 변경에 의한 분할 간 데이터 이동은 내부적으로 삭제와 삽입을 수반하여 성능 저하의 원인이 될 수 있으므로 사용에 주의한다.

일반 테이블을 분할 테이블로 변경

설명

일반 테이블을 분할 테이블로 변경하려면 **ALTER TABLE** 문을 이용한다. **ALTER TABLE** 문을 이용하여 세 종류의 분할 모드로 변경 가능하다. 분할 테이블로 변경하면 기존 테이블에 있던 데이터는 분할 정의에 따라 각 분할로 이동 저장된다.

구문

```
ALTER {TABLE | CLASS} table_name
PARTITION BY {RANGE | HASH | LIST } ( <partition_expression> )
( PARTITION partition_name VALUES LESS THAN { MAXVALUE | ( <partition_value_option> ) }
| PARTITION partition_name VALUES IN ( <partition_value_option list> > ]
| PARTITION <UNSIGNED_INTEGER> )

<partition_expression>
expression
<partition_value_option>
literal_
```

- *table_name* : 변경하려는 테이블의 이름을 지정한다.
- *partition_expression* : 분할 표현식을 지정한다. 표현식은 분할 대상이 되는 컬럼 명을 지정하거나 함수를 사용하여 지정할 수 있다. 사용 가능한 데이터 타입과 함수에 대한 자세한 내용은 [분할 표현식에 사용할 수 있는 데이터 타입](#)을 참조한다.
- *partition_name* : 분할명을 지정한다.
- *partition_value_option* : 분할의 기준이 되는 값 또는 값의 목록을 지정한다.

예제

다음은 record 테이블을 영역, 리스트, 해시 분할로 각각 변경하는 예제이다.

```
ALTER TABLE record PARTITION BY RANGE (host year)
( PARTITION before_1996 VALUES LESS THAN (1996),
  PARTITION after_1996 VALUES LESS THAN MAXVALUE);

ALTER TABLE record PARTITION BY list (unit)
( PARTITION time_record VALUES IN ('Time'),
  PARTITION kg_record VALUES IN ('kg'),
  PARTITION meter_record VALUES IN ('Meter'),
  PARTITION score_record VALUES IN ('Score') );

ALTER TABLE record
PARTITION BY HASH (score) PARTITIONS 4;
```

주의 사항

- 분할 조건을 충족하지 않는 데이터가 존재하는 경우에는 분할이 정의되지 않는다.

분할 테이블을 일반 테이블로 변경

설명

기존에 정의된 분할 테이블을 일반 테이블로 변경하려면 **ALTER TABLE** 문을 이용한다.

구문

```
ALTER {TABLE | CLASS} <table_name>  
REMOVE PARTITIONING
```

- *table_name*: 변경하고자 하는 테이블의 이름을 지정한다.

예제

다음은 분할 테이블인 nation2를 일반 테이블로 변경하는 예제이다.

```
ALTER TABLE nation2 REMOVE PARTITIONING;
```

분할 프루닝

설명

분할 프루닝(partition pruning)은 검색 조건을 통해 데이터 검색 범위를 한정시키는 기능이다. 질의에서 필요한 데이터를 포함하고 있지 않은 분할은 검색 과정에서 제외시킨다. 이를 통해 디스크로부터 인출되는 데이터의 양과 처리 시간을 크게 줄이고 질의 성능 및 자원 사용률을 개선할 수 있다.

예제 1

다음은 참가한 올림픽의 개최연도에 따라 영역 분할하는 olympic2 테이블을 생성하고 2000년도 시드니 올림픽 이후의 올림픽에 참가한 국가를 조회하는 질의이다. **WHERE** 절에서 분할 키에 대하여 상수 값과 동등 비교 또는 범위를 비교하는 경우 분할 프루닝이 발생한다.

예제의 경우 2000보다 작은 연도 값을 가진 before_1996 분할은 접근하지 않는다.

```
CREATE TABLE olympic2  
( opening_date DATE, host_nation VARCHAR(40))  
PARTITION BY RANGE ( EXTRACT (YEAR FROM opening_date) )  
( PARTITION before_1996 VALUES LESS THAN (1996),  
  PARTITION before_MAX VALUES LESS THAN MAXVALUE );  
  
SELECT opening_date, host_nation FROM olympic2 WHERE EXTRACT ( YEAR FROM (opening_date))  
>= 2000;
```

예제 2

다음은 분할 프루닝이 되지 않는 경우에 사용자가 특정 분할을 지정하여 데이터를 조회함으로써 분할 프루닝의 효과를 얻는 방법을 보여주는 예제이다.

예제에서 첫 번째 질의는 비교 값이 분할 표현식과 같은 형식이 아니므로 분할 프루닝이 일어나지 않는다. 따라서 두 번째 질의와 같이 알맞은 분할을 지정하여 분할 프루닝이 발생하는 것과 같은 기능을 사용할 수 있다.

```
SELECT host_nation FROM olympic2 WHERE opening_date >= '2000 - 01 - 01';
SELECT host_nation FROM olympic2__p__before_max WHERE opening_date >= '2000 - 01 - 01';
```

예제 3

다음은 해시 분할 테이블인 manager 테이블에서 분할 프루닝이 발생하도록 검색 조건을 지정한 예제이다.

해시 분할의 경우 **WHERE** 절에서 분할 키에 대하여 상수 값과 동등 비교를 하는 경우에만 분할 프루닝이 발생한다.

```
CREATE TABLE manager (
code INT,
name VARCHAR(50))
PARTITION BY HASH ( code) PARTITIONS 4;
SELECT * FROM manager WHERE code = 10053;
```

주의 사항

- 분할 표현식과 비교되는 값은 표현식의 결과와 같은 형식이어야 한다.

분할 표현식에 사용할 수 있는 데이터 타입

설명

분할 키로 사용할 수 있는 컬럼의 데이터 타입과 사용할 수 없는 데이터 타입은 다음과 같다.

사용할 수 있는 데이터 타입	사용할 수 없는 데이터 타입
CHAR	FLOAT
VARCHAR	REAL
NCHAR	DOUBLE
VARNCHAR	BIT
INTEGER	BIT VARYING
SMALLINT	NUMERIC OR DECIMAL
DATE	MONETARY
TIME	SET
TIMESTAMP	LIST OR SEQUENCE
	MULTISET
	OBJECT

다음과 같은 연산자 함수를 분할키에 적용하는 분할 표현식에 사용할 수 있다.

- 숫자 관련 연산자 함수
+, -, *, /, MOD, STRCAT, FLOOR, CEIL, POWER, ROUND, ABS, TRUNC

- 문자열 관련 연산자 함수
POSITION, SUBSTRING, OCTEC_LENGTH, BIT_LENGTH, CHAR_LENGTH, LOWER, UPPER, TRIM, LTRIM, RTRIM, LPAD, RPAD, REPLACE, TRANSLATE
- 날짜 관련 연산자 함수
ADD_MONTH, LAST_DAY, MONTH_BETWEEN, SYS_DATE, SYS_TIME, SYS_TIMESTAMP, TO_DATE, TO_NUMBER, TO_TIME, TO_TIMESTAMP, TO_CHAR
- 기타 관련 연산자 함수
EXTRACT, CAST

분할 테이블을 이용하여 VIEW 생성

설명

분할 테이블의 각 분할을 이용하여 뷰를 정의할 수 있다. 이 때, 생성된 뷰를 이용하여 데이터를 조회할 수 있지만, 데이터 삽입, 삭제, 갱신은 할 수 없다.

예제

다음은 참가연도에 따라 영역 분할된 participant2 테이블을 생성하고 participant2__p__before_2000 분할을 이용하여 뷰를 생성, 조회하는 예제이다.

```
CREATE TABLE participant2 (host_year INT, nation CHAR(3), gold INT, silver INT, bronze INT)
PARTITION BY RANGE (host_year)
( PARTITION before_2000 VALUES LESS THAN (2000),
  PARTITION before_2008 VALUES LESS THAN (2008) );

INSERT INTO participant2 VALUES (1988, 'NZL', 3, 2, 8);
INSERT INTO participant2 VALUES (1988, 'CAN', 3, 2, 5);
INSERT INTO participant2 VALUES (1996, 'KOR', 7, 15, 5);
INSERT INTO participant2 VALUES (2000, 'RUS', 32, 28, 28);
INSERT INTO participant2 VALUES (2004, 'JPN', 16, 9, 12);

CREATE VIEW v_2000 AS
SELECT * FROM participant2__p__before_2000
WHERE host_year = 1988;
csql> SELECT * FROM v_2000;
csql> ;x
=== <Result of SELECT Command in Line 1> ===
=====
host_year  nation      gold      silver     bronze
=====
1988      'NZL'        3         2         8
1988      'CAN'        3         2         5
=====
2 rows selected.
```

분할 테이블의 통계 정보 갱신

데이터베이스의 통계 정보를 갱신하는 방법에는 **cubrid optimizedb** 유틸리티를 이용하는 방법과 **UPDATE STATISTICS ON CLASSES**라는 SQL 문을 이용하는 방법이 있다. 여기에 분할된 테이블에 대해서는 **ANALYZE PARTITION** 구문이 추가로 지원된다.

ANALYZE PARTITION 구문 예제는 다음과 같다.

```
ALTER TABLE t1 ANALYZE PARTITION p3;
```

클래스 상속

개요

설명

CUBRID 데이터베이스에 있는 클래스들은 클래스 계층 구조를 가질 수 있으며, 계층 구조를 통해 속성과 메소드를 상속할 수 있다.

예를 들면, Employee 클래스로부터 속성과 메소드를 상속하는 Manager 클래스를 생성할 수 있다. 이때 Manager 클래스는 Employee 클래스의 서브클래스라고 하고, Employee 클래스는 Manager 클래스의 슈퍼클래스라고 한다. 상속을 이용하면 이미 존재하는 클래스의 구조를 재사용하므로 간단하게 클래스를 생성할 수 있다.

CUBRID는 다중 상속을 허용하므로, 하나의 클래스는 두 개 이상의 클래스로부터 속성과 메소드를 상속할 수 있다. 그러나 다중 상속을 이용하면 메소드나 속성, 클래스를 삭제하거나 추가할 때 충돌이 발생할 수 있다.

클래스 충돌은 클래스와 슈퍼클래스, 또는 두 개 이상의 슈퍼클래스에 같은 이름의 속성이나 메소드가 존재할 때 발생한다. 예를 들어 클래스가 두 개 이상의 클래스로부터 이름과 도메인이 같은 속성을 상속할 가능성이 있다면, 어떤 클래스의 속성을 상속할 것인지 지정해야 한다. 이때 지정된 슈퍼클래스가 삭제되면, 다른 슈퍼클래스로부터 이름과 도메인이 같은 속성을 상속하도록 지정해야 한다. 일반적으로는 데이터베이스 시스템이 자동으로 이와 같은 문제를 해결하지만, 시스템과 다른 방법으로 해결하고 싶다면 상속 구문을 통해 직접 지정할 수 있다.

클래스가 두 개 이상의 클래스로부터 속성을 상속할 때 속성의 이름은 같지만 도메인은 다른 경우, 서브클래스는 좀 더 상세한 도메인을 가지는 속성을 자동으로 상속한다. 예를 들어, 이름이 같고 도메인이 클래스인 속성을 두 개의 슈퍼클래스가 가지고 있다면, 클래스 계층 구조에서 더 하위에 존재하는 클래스의 속성을 상속한다. 이때 해당 속성의 도메인이 각각 스트링, 정수와 같이 시스템이 제공하는 기본 타입이라면, 상속은 불가능하다.

상속 시 발생하는 충돌과 이에 대한 해결 방법은 [클래스 충돌 해결](#)에서 다룬다.

주의 사항

상속 시 주의 사항은 다음과 같다.

- 클래스 이름은 데이터베이스 내에서 고유해야 한다. 존재하지 않는 클래스로부터 상속하는 클래스를 생성하면 에러가 발생한다.
- 한 클래스 내에서 메소드 이름과 속성의 이름은 고유해야 한다. 이름은 공백을 포함할 수 없으며 CUBRID의 예약어와 중복될 수 없다. 알파벳, '.', '#', '%' 등은 클래스 이름에 포함될 수 있으나 첫 글자가

'.' 가 될 수는 없다. 클래스 이름은 영문 255자를 넘을 수 없다. 클래스 이름은 대/소문자를 구분하지 않고 소문자로 변환되어 시스템에 저장된다.

참고 상속 구문에서 수퍼클래스의 이름을 알기 쉽게 명시하기 위해서 수퍼클래스 이름 앞에 사용자의 이름을 붙일 수 있다.

클래스 속성과 클래스 메소드

클래스 객체나 클래스에 저장된 모든 인스턴스의 공통적인 특징을 저장하기 위한 클래스 속성을 생성할 수 있다. 클래스 메소드는 클래스 객체에 대한 연산을 위해 생성된다. 클래스 속성이나 메소드를 생성하기 위해서는 속성이나 메소드 이름 앞에 **CLASS**라는 키워드를 사용한다. 클래스 속성은 인스턴스와 관련이 있다기 보다는 클래스 자체와 관련이 있기 때문에 클래스 속성의 값은 하나의 값만 존재한다. 예를 들어, 클래스 속성은 클래스 메소드에 의해 결정되는 평균값을 저장하거나, 클래스가 생성된 timestamp 값을 저장할 수 있다. 클래스 메소드는 클래스 객체 자체를 대상으로 메소드가 수행된다. 클래스 메소드는 클래스에 저장된 인스턴스 값에 대한 집합 값을 계산하는 용도로 사용될 수 있다.

서브클래스가 수퍼클래스를 상속할 때, 각 클래스는 클래스 속성을 위한 별도의 저장 공간을 가지므로 두 클래스의 클래스 속성은 서로 다른 값을 가질 수 있다. 따라서 수퍼클래스의 클래스 속성에 대한 값 변경은 서브클래스에 영향을 주지 않는다.

클래스 속성의 이름은 동일한 클래스 내의 인스턴스 속성의 이름과 같을 수 있다. 마찬가지로, 클래스 메소드의 이름도 동일한 클래스 내의 인스턴스 메소드의 이름과 같을 수 있다.

상속을 위한 순서 규칙

상속이 적용된 경우 다음 규칙들이 적용된다. 클래스라는 용어는 데이터베이스 내에서 클래스와 가상 클래스간 상속 개념을 서술할 때 일반적으로 사용된다.

- 수퍼클래스가 없는 객체의 경우, 속성 정의 순서는 **CREATE** 구문에서 속성을 정의한 순서와 동일하다(이 특징은 ANSI 표준이다).
- 수퍼클래스가 하나인 경우, 수퍼클래스의 속성들 다음에 지역적으로 생성된 속성이 위치한다. 수퍼클래스로부터 상속받은 속성들 간의 순서는 수퍼클래스 정의 시 정해진 순서를 따른다. 다중 상속인 경우, 클래스 정의 시 지정된 수퍼클래스의 순서에 따라 수퍼클래스의 속성 순서가 결정된다.
- 두 개 이상의 수퍼클래스가 동일한 클래스로부터 상속된 클래스라면, 두 수퍼클래스에 동시에 존재하는 속성은 한 번만 서브클래스에 상속된다. 이 때, 충돌이 발생하면 서브클래스는 첫 번째 수퍼클래스의 속성을 상속한다.
- 두 개 이상의 수퍼클래스들 사이에 이름 충돌이 발생할 경우, 이름 충돌을 해결하기 위해 **INHERIT** 구문을 사용하여 수퍼클래스의 속성 중 원하는 것만 상속 받을 수 있다.
- 수퍼클래스의 속성의 이름이 **INHERIT** 구문의 별칭 기능을 통해 변경된 경우, 이름이 변경된 속성의 위치는 전과 동일하게 유지된다.

INHERIT 절

설명

한 클래스의 서브클래스로 클래스를 생성하면 자동으로 클래스 계층 구조의 상위 클래스들의 모든 속성과 메소드를 상속받는다. 상속 시 발생할 수 있는 이름 충돌은 시스템에 의해 자동으로 처리되거나 사용자가 직접 해결할 수 있다. 사용자가 이름 충돌을 해결하고자 한다면 **CREATE CLASS** 구문에 **INHERIT** 구문을 추가하여 해결할 수 있다:

구문

```
CREATE CLASS
.
.
.
INHERIT resolution [ { , resolution } _ ]
resolution:
{ column_name | method_name } OF superclass_name [ AS alias ]
```

INHERIT 구문의 *attr_mthd_name* 위치에 상속하고 싶은 슈퍼클래스의 속성이나 메소드 이름을 지정한다.

AS 절을 사용하여 새로운 이름으로 상속받을 수도 있으므로, 다중 상속 구문에서 이름 충돌이 발생할 경우에서 충돌을 해결할 수 있다.

ADD SUPERCLASS 절

설명

클래스의 상속은 클래스에 슈퍼클래스를 추가하여 확장할 수 있다. 이미 존재하는 클래스에 슈퍼클래스를 추가하여 두 클래스 사이에 관계를 생성한다. 슈퍼클래스를 추가한다는 것이 새로운 클래스를 추가한다는 것을 의미하지는 않는다.

구문

```
ALTER CLASS
.
.
.
ADD SUPERCLASS [ user_name.]class_name [ { , [ user_name.]class_name } _ ]
[ INHERIT resolution [ { , resolution } _ ] ] [ ; ]
resolution:
{ column_name | method_name } OF superclass_name [ AS alias ]
```

슈퍼클래스를 추가할 클래스의 이름을 첫 번째 *class_name*에 지정한다. 위 구문을 사용하여 슈퍼클래스의 속성과 메소드를 상속할 수 있다.

새로운 슈퍼클래스를 추가할 경우 이름 충돌이 발생할 수 있다. 데이터베이스 시스템의 의해서 이름 충돌이 자동으로 해결될 수 없는 경우, **INHERIT** 구문을 사용하여 슈퍼클래스에서 상속받을 속성이나 메소드를 지정할 수 있다. 충돌이 발생한 속성이나 메소드를 모두 상속 받기 위해서는 별칭을 사용할 수 있다.

슈퍼클래스에서 발생하는 이름 충돌에 대한 자세한 설명은 [클래스 충돌 해결](#)을 참조한다.

예제

demodb에 포함되어 있는 event 클래스를 상속하여 female_event 클래스를 생성한다면 다음과 같은 클래스 생성 문장이 수행된다.

```
CREATE CLASS female_event UNDER event;
```

DROP SUPERCLASS 절

설명

클래스로부터 수퍼클래스를 삭제하는 것은 두 클래스 사이의 관계를 제거하는 것이다. 클래스에서 수퍼클래스를 삭제하면, 해당 클래스뿐만 아니라 그 클래스의 모든 서브클래스의 상속 관계 수정을 의미한다.

구문

```
ALTER CLASS
.
.
.
DROP SUPERCLASS class_name [ { , class_name }_ ]
[ INHERIT resolution [ { , resolution }_ ] ] [ ; ]

resolution:
{ column_name | method_name } OF superclass_name [ AS alias ]
```

첫 번째 *class_name*에는 수정할 클래스의 이름을 지정하고 두 번째 *class_name*에는 삭제할 수퍼클래스의 이름을 지정한다. 수퍼클래스의 삭제에 의해 이름 충돌이 발생할 경우, 해결 방법은 [클래스 충돌 해결](#)을 참조한다.

예제 1

다음은 female_event 클래스가 event 클래스를 상속받은 예이다.

```
CREATE CLASS female_event UNDER event;
```

예제 2

다음 ALTER 구문은 female_event 클래스에서 수퍼클래스 event를 삭제하는 예이다. female_event 클래스가 event 클래스로부터 상속받은 모든 속성은 더 이상 존재하지 않는다.

```
ALTER CLASS female_event
DROP SUPERCLASS event;
```

클래스 충돌 해결

개요

데이터베이스의 스키마를 변경하면 상속 관련 클래스들 사이의 속성이나 메소드에서 충돌이 발생할 수 있다. 충돌하면 대부분, CUBRID에서 자동으로 해결되지만 그렇지 않은 경우에는 사용자가 직접 충돌을 해결해야 한다. 따라서 스키마를 변경하기 전에, 충돌이 발생할 가능성을 면밀히 조사해야 한다.

두 가지 형태의 충돌이 데이터베이스 스키마를 손상시킬 수 있다. 하나는 서브클래스의 스키마가 변경되어 서브클래스와 충돌이 발생하는 경우이고 또 다른 하나는 슈퍼클래스가 변경되어 서브클래스와 충돌이 발생하는 것이다. 다음은 클래스들간 충돌을 유발하는 연산들이다:

- 속성 추가
- 속성 삭제
- 슈퍼클래스의 추가
- 슈퍼클래스의 삭제
- 클래스 삭제

위의 연산들로 인해 서브클래스와 충돌이 발생할 경우, CUBRID는 충돌이 발생한 서브클래스에 대해 기본 해결 방법을 적용한다. 따라서 데이터베이스 스키마는 항상 일관된 상태를 유지한다.

해결 지시자

설명

데이터베이스 스키마를 변경하면, 기존 클래스나 속성 간의 충돌이나 상속 충돌이 발생할 수 있다. 시스템이 자동으로 충돌을 해결하지 못하거나 시스템의 해결 방법이 마음에 들지 않으면 **ALTER** 구문의 **INHERIT** 절을 사용하여 충돌을 해결하는 방법을 제시할 수 있다(흔히 해결 지시자 라고 한다).

시스템이 자동적으로 충돌을 해결할 때는 상속이 존재한다면 기본적으로 이전의 상속을 유지한다. 스키마 변경으로 인해 이전의 해결 방법이 무효화된다면 시스템은 또 다른 해결 방법을 임의로 선택할 것이다. 따라서 시스템이 충돌을 해결하는 방법을 항상 예측할 수는 없으므로 가급적이면 스키마 설계 단계에서 속성이나 메소드의 과도한 재사용을 피해야 한다.

다음에서 충돌과 관련하여 논의하고 있는 사항은 속성과 메소드에 공통적으로 적용된다.

구문

```
ALTER [ class type ] class name alter clause
[ INHERIT resolution [ {, resolution }_ ] ] [ ; ]

resolution:
{ column_name | method_name } OF superclass_name [ AS alias ]
```

수퍼클래스 충돌

수퍼클래스 추가

ALTER CLASS 구문에서 **INHERIT** 절은 선택 사항이지만 클래스의 변경에 의해 충돌이 발생할 경우에는 반드시 사용해야 하는 문장이다. **INHERIT** 절 다음에 하나 이상의 해결방법을 명시할 수 있다.

*superclass_name*에는 충돌이 발생했을 때 새로 상속받을 속성이나 메소드를 가지는 수퍼클래스의 이름을 명시하고, *attr_mthd_name*에는 상속받을 속성이나 메소드의 이름을 명시한다. 상속받을 속성이나 메소드의 이름을 변경할 필요가 있는 경우에는 **AS** 절을 이용하여 별칭을 지정할 수 있다.

다음 예는 demodb에 있는 olympic db 중 event 클래스와 stadium 클래스를 상속받아서 soccer_stadium 클래스를 만든다. event 클래스와 stadium 클래스는 모두 name, code 속성을 가지고 있기 때문에

INHERIT을 사용하여 상속받을 속성을 지정해야 한다.

```
CREATE CLASS soccer_stadium UNDER event, stadium
INHERIT name OF stadium, code OF stadium;
```

두 수퍼클래스 event, stadium가 name이라는 속성을 가지고 있고, soccer_stadium 클래스가 두 속성을 모두 상속받으려면, stadium의 name은 그대로 상속 받고 event 클래스의 name은 **INHERIT**의 **alias** 절을 사용하여 이름을 변경하여 상속 받을 수 있다.

아래 예는 stadium 클래스의 name은 그대로 name으로 상속받고, event 클래스의 name은 purpose이라는 별명으로 상속받는다.

```
ALTER CLASS soccer_stadium
INHERIT name OF event AS purpose;
```

수퍼클래스 삭제

INHERIT을 사용하여 명시적으로 속성이나 메소드를 상속한 수퍼클래스를 삭제하면 서브클래스에서 다시 이름 충돌이 발생할 수 있다. 이 경우에는 삭제할 때 명시적으로 상속받을 속성이나 메소드를 지정해야 한다.

다음 예는 demodb의 game, participant, stadium 클래스를 상속받아서 seoul_1988_soccer 클래스를 만들고, 그 중 participant 클래스를 수퍼클래스에서 제거한다. participant 클래스에서 nation_code와 host_year를 명시적으로 상속받았기 때문에, 수퍼클래스에서 제거하기 전에 nation_code와 host_year의 이름 충돌을 해결해야 한다. 하지만, host_year는 game 클래스에만 존재하므로 명시적으로 지정할 필요는 없다.

```
CREATE CLASS seoul_1988_soccer UNDER game, participant, stadium
INHERIT nation_code OF participant, host_year OF participant;
ALTER CLASS seoul_1988_soccer
DROP SUPERCLASS participant
INHERIT nation_code OF stadium;
```

호환되는 도메인

두 개 이상의 수퍼클래스 사이에서 속성의 충돌이 발생할 때, 모든 속성이 호환되는 도메인을 가지는 경우에만 충돌을 해결하는 구문이 불가능하다.

예들 들어, 정수 타입의 phone이라는 속성을 가지는 수퍼클래스를 상속받은 클래스에는 스트링 타입의 phone 속성을 가지는 또 다른 수퍼클래스를 추가할 수 없다. 두 수퍼클래스의 phone 속성의 타입이 모두 스트링이거나 정수라면 **INHERIT** 구문을 이용하여 충돌을 해결하면서 수퍼클래스를 추가할 수 있다.

이름은 같지만 도메인이 다른 속성을 상속할 때 도메인 호환성이 점검된다. 이 경우, 클래스 상속 계층 구조의 하위 클래스를 도메인으로 갖는 속성이 자동으로 상속된다. 상속받을 속성들의 도메인이 호환 가능할 때, 상속 관계가 만들어지는 클래스에서 충돌이 해결되어야 한다.

서브클래스 충돌

클래스의 변경 사항은 모든 서브클래스에 자동으로 전파된다. 변화된 내용으로 인해 서브클래스에 문제가 발생한다면, CUBRID가 문제되는 서브클래스 충돌(subclass conflict)을 처리하고 시스템이 자동으로 충돌을 해결했다는 경고 메시지를 보여준다.

수퍼클래스의 추가, 속성과 메소드의 생성, 삭제로 인해 서브클래스 충돌이 발생할 수 있다. 클래스의 변경 사항은 모든 서브클래스에 영향을 미친다. 변경된 사항이 자동 전파되는 특징으로 인해 정상적인 변경도 하위 서브클래스들에 부작용을 유발할 수 있다.

속성과 메소드의 추가

서브클래스 충돌의 가장 단순한 형태는 속성을 추가할 때 발생한다. 한 수퍼클래스에 추가된 속성이 또 다른 수퍼클래스에서 이미 상속 받고 있는 속성의 이름과 동일하다면 서브클래스 충돌이 발생할 것이다. 이러한 경우 CUBRID는 이 문제를 자동으로 해결할 것이다. 즉, 추가된 속성은 동일한 이름의 속성을 이미 상속하고 있는 모든 서브클래스에 상속되지 않는다.

다음은 event 클래스에 속성을 추가하는 예이다. soccer_stadium 클래스는 수퍼클래스로 event와 stadium 클래스를 가지며, stadium 클래스에는 nation_code 속성이 이미 존재한다. 따라서 event 클래스에 nation_code 속성을 추가하면 soccer_stadium 클래스에서는 nation_code 속성과 관련하여 충돌이 발생하지만, CUBRID가 이 충돌을 자동으로 해결한다.

```
ALTER CLASS event
ADD ATTRIBUTE nation_code CHAR(3);
```

만약 event가 soccer_stadium의 수퍼클래스에서 제거되면, stadium 클래스의 cost 속성이 자동으로 상속될 것이다.

속성과 메소드의 삭제

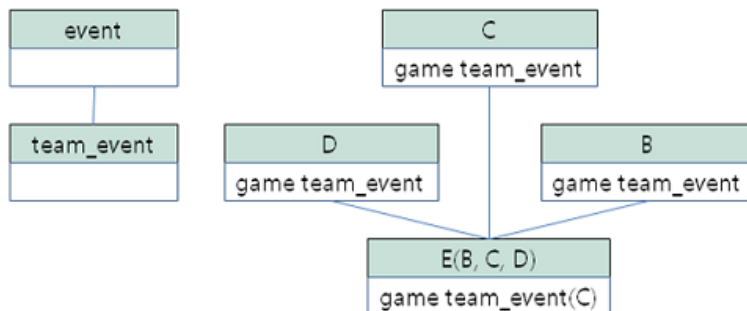
속성이 삭제되면, **INHERIT** 구문을 사용하여 그 속성을 상속받도록 한 문장의 효력 역시 사라진다. 속성이 삭제됨으로써 충돌이 발생한다면 시스템은 새로운 상속 계층 구조를 결정할 것이다. 만약, 시스템이 결정한 상속 계층 구조가 마음에 들지 않으면 **ALTER** 구문의 **INHERIT** 절을 사용하여 사용자가 계층 구조를 정할 수도 있다. 아래의 경우가 이러한 충돌에 해당할 것이다.

세 개의 서로 다른 수퍼클래스로부터 속성을 상속 받는 서브클래스가 있다고 가정하자. 모든 수퍼클래스에서 이름 충돌이 발생하였고, 이 문제를 해결하기 위해 명시적으로 상속된 속성이 삭제되었다면 나머지 두 개의 속성 중 하나가 자동으로 상속될 것이다.

다음은 서브클래스 충돌의 예이다. 클래스 B, C, D는 클래스 E의 수퍼클래스고 세 개의 수퍼클래스는 이름이 team이고 도메인이 team_event인 속성을 가진다. 클래스 E는 다음과 같이 C 클래스의 place 속성을 상속받으며 생성되었다:

```
create class E under B, C, D
inherit place of C;
```

이 경우의 상속 계층 구조는 다음과 같다:



클래스 C를 수퍼클래스에서 삭제하기로 결정했다고 가정하자. 이 삭제는 상속 계층 구조의 변경을 요구할 것이다. 나머지 B, D 클래스의 game 속성의 도메인이 동일 레벨이므로 시스템은 둘 중 하나를 임의로 선택하여 상속할 것이다. 시스템의 임의의 선택을 원하지 않으면 클래스 변경 시에 **INHERIT** 구문을 사용하여 상속받을 클래스를 지정할 수 있다:

```
ALTER CLASS E
INHERIT game OF D;

ALTER CLASS C
DROP game;
```

참고 한 수퍼클래스의 game 속성의 도메인이 event이고, 또 다른 수퍼클래스의 속성이 team_event인 경우, team_event가 event 에 비해 더 상세하므로(상속 계층 구조상 더 하위에 존재하므로) team_event를 도메인으로 가지는 속성이 상속될 것이다. 이 경우 사용자가 강제로 event를 도메인으로 가지는 속성을 상속할 수는 없다. event 클래스는 team_event보다 상속 계층 구조의 상위에 존재하기 때문이다.

스키마 불변성

데이터베이스 스키마 불변성은 항상(스키마 변경 전/후) 스키마가 지켜야 하는 스키마의 특징이다, 클래스 계층 불변성, 이름 불변성, 상속 불변성, 일관성의 불변성 등 네 가지 유형의 불변성이 존재한다.

- **클래스 계층 불변성**은 하나의 루트를 가지며 연결된 클래스들이 방향성을 갖는 비순환 그래프(DAG: directed acyclic graph)인 클래스 계층 구조를 정의한다. 즉, 루트를 제외한 모든 클래스는 하나 이상의

수퍼클래스를 가지고 자기 자신이 수퍼클래스가 될 수 없다. DAG의 루트는 object라는 시스템 정의 클래스이다.

- **이름 불변성**이란 클래스 계층 구조상의 모든 클래스는 고유한 이름을 가져야 하고, 클래스 내의 모든 속성 역시 고유한 이름을 가져야 함을 의미한다. 즉, 동일한 이름의 클래스를 생성하거나 한 클래스에서 동일한 이름의 속성, 메소드를 생성하는 것은 규칙에 어긋나므로 거부된다.

이름 불변성은 이름변경 한정어(qualifier)에 의해 재정의된다. 이름변경 한정어는 속성 또는 메소드의 이름이 변경될 수 있도록 한다.

- **상속 불변성**은 한 클래스는 모든 수퍼클래스의 모든 속성들과 메소드들을 상속해야 한다는 것이다. 이 불변성은 출처 한정어, 충돌 한정어, 도메인 한정어 등 세 개의 한정어로 구분될 수 있다. 상속 이후, 상속된 속성들과 메소드들은 이름이 변경될 수 있다. 기본 값 또는 공유 값 속성의 경우에, 기본 값과 공유 값은 수정될 수 있다. 상속 불변성은 이러한 변경들이 속성들과 메소드들을 상속한 모든 클래스에 전파될 것이라는 것을 의미한다.
 - **출처 한정어**는, 클래스 S라는 클래스를 상속한 클래스들을 클래스 C가 다시 상속받을 경우, 클래스 S로부터 각각의 클래스에 상속된 속성(메소드)들은 오직 하나씩만 클래스 C에 상속될 수 있다는 것을 의미한다. 다시 말하면, 만일 한 속성(메소드)이 클래스 S에 먼저 정의되었고, 다른 클래스들에 의해 상속되었다면, 그 속성(메소드)이 여러 개의 서브클래스에 존재하지만 실질적으로는 한 속성(메소드)인 것이다. 따라서, 한 클래스가 출처가 같은 속성(메소드)를 가지는 클래스들로부터 다중 상속 받는 경우, 오직 한 속성(메소드)의 모습만을 상속한다.
 - **충돌 한정어**란, 출처는 다르지만 동일한 이름을 가지는 속성(메소드)을 가지는 두 개 이상의 클래스를 클래스 C가 상속한다면, 클래스 C는 하나 이상의 클래스를 모두 상속받을 수 있다는 것이다. 동일한 이름의 속성(메소드)를 상속받으려면 이름 불변성을 위반하므로 이름 변경이 필요하다.
 - **도메인 한정어**는 상속된 속성의 도메인이 그 도메인의 서브클래스로 변환될 수도 있음을 의미한다.
- **일치 불변성**은 데이터베이스 스키마는 스키마를 변경하는 순간을 제외하고 항상 스키마 불변성과 모든 규칙들([스키마 변경 규칙](#))을 준수해야 한다는 것이다.

스키마 변경 규칙

스키마 불변성에서 항상 유지되어야 하는 스키마의 특성들에 대해 언급하였다. 스키마를 변경하는 방법은 몇 가지가 존재하며 이 방법들은 스키마 불변성을 유지해야 한다. 예를 들어, 수퍼클래스를 하나만 가지는 클래스에서 그 수퍼클래스와의 관계를 제거한다고 가정하자. 수퍼클래스와의 관계가 삭제되면 그 클래스는 object 클래스의 직속 서브클래스가 되거나 만약 사용자가 그 클래스는 적어도 하나의 수퍼클래스를 가져야 한다고 명시했다면 그 삭제는 거부될 것이다. 이러한 선택은 임의적인 측면이 있지만, 스키마를 변경하는 방법 중 하나를 선택하기 위한 몇 가지 규칙을 가지는 것은 사용자나 데이터베이스 설계자에게 분명 유용할 것이다.

충돌 해결 규칙(conflict-resolution rules), 도메인 변경 규칙(domain-change rule), 클래스 계층 규칙(class-hierarchy rule)의 세 가지 형태 규칙이 적용된다.

일곱 개의 충돌 해결 규칙은 상속 불변성을 강화한다. 대부분의 스키마 변경 규칙은 이름 충돌 때문에 필요하다. 도메인 변경 규칙은 상속 불변성의 도메인 해결을 강화한다. 클래스 계층 규칙은 클래스 계층 불변성을 강화한다.

충돌 해결 규칙

- **규칙 1 :** 클래스 C의 속성(메소드) 이름이 수퍼클래스 S의 속성 이름과 충돌이 발생한다면(이름이 같다면), 클래스 C의 속성이 사용된다. S의 속성은 상속되지 않는다.

어떤 클래스가 하나 이상의 수퍼클래스를 가지는 경우, 속성들이 의미적으로 같은지, 어떤 속성을 상속 받을 것인지를 결정하기 위해 각 수퍼클래스가 가지는 속성(메소드)들의 세가지 측면이 고려되어야 한다. 속성(메소드)의 세 가지 측면은 이름, 도메인, 출처이다. 아래 표는 세 가지 측면에서 두 수퍼클래스에서 발생할 수 있는 여덟 가지 조합이다. 사례 1의 경우(두 개의 서로 다른 수퍼클래스의 속성이 이름, 도메인, 출처가 모두 같은 경우), 두 속성은 동일하므로 서브클래스는 둘 중 하나만 상속받아야 한다. 사례 8의 경우(두 개의 서로 다른 수퍼클래스의 속성이 이름, 도메인, 출처가 모두 다른 경우), 두 속성은 완전히 다른 속성이므로 모두 상속받아야 한다.

사례	이름	도메인	출처
1	같음	같음	같음
2	같음	같음	다름
3	같음	다름	같음
4	같음	다름	다름
5	다름	같음	같음
6	다름	같음	다름
7	다름	다름	같음
8	다름	다름	다름

8개의 사례 중 5개(1, 5, 6, 7, 8)는 명확한 의미를 가지고 있다. 상속 불변성은 이러한 경우의 충돌을 해결하기 위한 가이드 라인이다. 나머지 사례(2, 3, 4)의 경우, 충돌을 자동으로 해결하는 것은 매우 어렵다. 규칙 2, 규칙 3이 이러한 충돌의 해결 방안이 될 수 있다.

- **규칙 2 :** 두 개 이상의 수퍼클래스가 출처는 다르지만 같은 이름과 도메인의 속성(메소드)을 가질 때, 사용자가 충돌 해결 구문을 사용할 경우 하나 이상의 속성(메소드)을 상속할 수 있다. 충돌 해결 구문을 사용하지 않는다면 시스템은 임의의 어느 한 속성을 선택하여 상속할 것이다.

이 규칙은 위 표의 사례 2 형태의 충돌을 해결하기 위한 가이드 라인이다.

- **규칙 3 :** 두 개 이상의 수퍼클래스가 출처와 도메인은 다르지만 이름이 같은 속성(메소드)을 가질 때, 더 상세한 도메인(상속 계층 구조의 하위에 있는)을 가지는 속성(메소드)이 상속될 것이다. 도메인들 사이에 상속 관계가 없으면 스키마 변경은 허용되지 않는다.

이 규칙은 사례 3, 4 형태의 충돌을 해결하기 위한 가이드 라인이다. 규칙 3과 규칙 4가 충돌하는 경우, 규칙 3이 규칙 4보다 우선한다.

- **규칙 4** : 사용자는 사례 3, 4의 경우를 제외한 어떠한 변경도 가능하다. 뿐만 아니라, 서브클래스에 대한 충돌 해결이 슈퍼클래스에 대한 변경을 초래할 수 없다.

규칙 4의 철학은 "상속은 서브클래스가 슈퍼클래스로부터 부여 받은 권리로 서브클래스의 변경이 슈퍼클래스에 영향을 줄 수 없다"라는 것이다. 규칙 4는 클래스 C와 슈퍼클래스들 사이에 발생하는 충돌을 해결하기 위해 슈퍼클래스의 포함된 속성(메소드)의 이름을 변경할 수 없다는 것을 의미한다. 규칙 4의 예외는 스키마 변경이 사례 3, 4의 충돌을 유발하는 경우이다.

- 예를 들어, 클래스 A가 클래스 B의 슈퍼클래스고, 클래스 B가 타입이 **DATE**인 `playing_date`라는 속성을 가진다고 가정하자. 클래스 A에 **STRING** 타입의 `playing_date`라는 이름의 속성을 추가하면, 클래스 B의 `playing_date` 속성과 충돌이 발생할 것이다. 이것이 사례 4의 경우다. 이 충돌을 해결하는 정확한 방법은 사용자가 클래스 B가 클래스 A의 `playing_date` 속성을 상속하도록 명시하는 것이다. 메소드가 속성을 참조한다면, 클래스 B의 사용자는 올바른 `playing_date` 속성을 참조하도록 메소드를 적절히 변경할 필요가 있다. 클래스 A의 스키마 변경이 허용되지 않는 이유는 클래스 B의 사용자가 스키마 변경으로 인해 발행하는 충돌을 해결하기 위해 명시적인 구문을 기술하지 않으면, 스키마가 일관되지 않은 상태가 되기 때문이다.



- **규칙 5** : 슈퍼클래스의 스키마를 변경함으로써 충돌이 발생하면, 그 변경이 규칙들을 위반하지 않는 한 원래의 해결 방법이 유지된다. 그러나 스키마 변경이 원래의 해결 방법을 무효화한다면 시스템은 다른 해결 방법을 적용할 것이다.

규칙 5는 충돌이 없는 클래스에 충돌을 유발하거나, 이전의 충돌을 해결하는 방법을 무효화하는 상황을 책임지는 규칙이다.

이러한 경우는 슈퍼클래스에 속성(메소드)이 추가되거나 슈퍼클래스로부터 상속받은 속성(메소드)이 삭제될 때, 속성(메소드)의 이름 또는 도메인이 변경되거나, 슈퍼클래스가 삭제되는 상황이다. 규칙 5는 규칙 4의 철학과 일치한다. 즉, 사용자는 그 클래스를 상속한 서브클래스가 상속받은 속성(메소드)에 어떠한 영향을 미칠지 신경 쓰지 않고 자유롭게 클래스를 변경할 수 있다.

클래스 C의 슈퍼클래스의 스키마 변경할 때, 이전에 다른 클래스와 충돌이 발생하여 그 클래스의 속성을 상속하기로 결정했다면 클래스 C의 속성(메소드) 손실을 초래할 수 있다. 이 경우, 이전에 충돌했던 속성(메소드) 중 하나를 대신 상속 받아야 한다.

슈퍼클래스의 스키마 변경은 속성(메소드)과 클래스 C의 (지역적으로 선언되거나 상속받은) 속성(메소드)의 충돌을 일으킬 수 있다. 이 경우, 시스템은 규칙 2나 규칙 3을 적용하여 충돌을 자동으로 해결시키고 사용자에게 알릴 수도 있다.

수퍼클래스와의 관계를 추가하거나 삭제함으로써 새로운 충돌이 발생하는 상황은 규칙 5를 적용할 수 없다. 클래스에 대한 수퍼클래스 추가/삭제는 클래스 내에서 제어되어야 한다. 즉, 사용자가 명시적인 해결 방법을 제시해야 한다.

- **규칙 6** : 속성이나 메소드의 변경은 충돌이 발생하지 않는 서브클래스들에게만 전파된다.

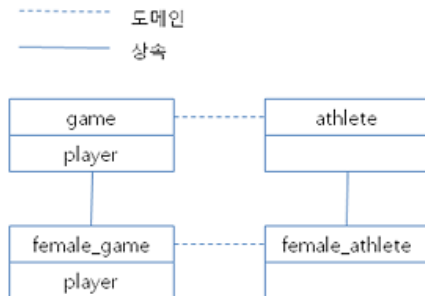
이 규칙은 규칙 5와 상속 불변성의 적용을 제한한다. 규칙 2, 규칙 3을 적용하여 충돌을 탐지하고 해결할 수 있다.

- **규칙 7** : 클래스 R의 속성이 클래스 C를 도메인으로 사용해도 클래스 C를 삭제할 수 있다. 이 경우, 클래스 C를 도메인으로 사용하는 속성의 도메인이 object로 변경 될 수 있다.

도메인 변경 규칙

- **규칙 8** : 클래스 C의 한 속성의 도메인이 D에서 D의 수퍼클래스로 변경되었다면 새로운 도메인은 클래스 C가 속성을 상속받은 수퍼클래스의 대응하는 도메인보다 더 일반적이지 않다. 다음 예는 이 규칙의 원리를 설명한다.

데이터베이스에 player라는 속성을 가지는 game 클래스와 game을 상속한 female_game 클래스가 존재한다고 가정하자. game의 player 속성의 도메인은 athlete 클래스이지만 female_game의 player 속성의 도메인은 athlete의 서브클래스인 female_athlete 클래스로 변경되었다. 다음 그림이 이러한 관계를 보여주고 있다. 그러나 female_game의 player 속성의 도메인은 female_athlete의 수퍼클래스인 athlete로 다시 변경될 수 있다.



클래스 계층 규칙

- **규칙 9** : 수퍼클래스가 없는 클래스는 object의 직속 서브클래스가 된다. 클래스 계층 규칙은 수퍼클래스가 없는 클래스의 특성을 정의한다. 수퍼클래스 없이 클래스를 생성한다면 object를 수퍼클래스가 갖게 된다. 만약 클래스 C의 고유한 수퍼클래스인 S를 삭제하면 클래스 C는 object의 직속 서브클래스가 된다.

멀티미디어 데이터 관리

glo 클래스

개요

사이즈가 큰 객체를 다루기 위해서는 **glo** (Generalized Large Object) 클래스를 사용한다. **glo** 클래스는 CUBRID에서 멀티미디어 지원의 토대가 된다. 사용자 정의 클래스의 속성을 **glo** 클래스(또는 그 하위 클래스)로 정의하면, 다른 유형의 비구조적 데이터를 저장할 수 있다. 비트맵, 디지털화된 오디오 데이터, 대용량 문서 데이터, 2진 데이터 등이 **glo** 클래스(또는 그 하위 클래스)의 인스턴스의 예이다.

glo 클래스의 인스턴스는 두 가지 형식으로 분류할 수 있다. 첫 번째 형식은 CUBRID를 통해서만 데이터베이스에 저장, 접근할 수 있는 LO (Large Object)이다. 두 번째 형식은 호스트 파일 시스템에 저장하는 FBO (File Based Object)이다. 대부분의 **glo** 인스턴스는 LO인데, 이는 CUBRID가 데이터를 관리하기에 더 편리하기 때문이다. 그러나 데이터베이스가 아닌 애플리케이션에서 데이터를 사용할 때에는, 호스트 파일 시스템이 FBO 데이터 스토리지를 관리하도록 하면 데이터를 압축하지 않아도 되는 등의 이점이 있다.

클래스에 정의된 속성

glo 클래스에 있는 모든 속성은 CUBRID에 의해 미리 정의되어 있다. 각 속성과 관련된 값은 하나 또는 그 이상의 **glo** 메소드를 호출할 때 정의된다. 따라서, 멀티미디어 데이터는 사용자 정의 클래스에 일반적인 데이터를 입력하는 방법으로 입력할 수 없고, 전용 **glo** 메소드를 통해 관련된 **glo** 인스턴스와 연결된다.

glo 클래스에 정의되어 있는 속성은 다음과 같다.

glo 속성	속성 타입	정의
unit_size	integer	glo 데이터의 유닛당 비트 수를 나타낸다. 일반적으로 8 이 사용된다. 데이터의 유닛당 허용되는 최대 비트 수는 2,147,483,647 이다.
header_size	integer	glo 데이터로부터 떨어져 있는 헤더(0 위치에서 시작된다)를 구성하는 유닛의 크기를 나타낸다. 디폴트 값은 0 이다.
holder_obj	glo_holder	내부 클래스로 일반 사용자는 접근할 수 없다.

다음은 **glo** 인스턴스와 그 속성에 관련된 값의 예를 그림으로 나타낸 것이다. 이 인스턴스의 **unit_size** 속성 값은 디폴트 값이다. 그러나 **header_size**의 값은 **glo** 메소드에 의해 결정되었다.

unit_size	8
header_size	28
	header_data
	glo data

클래스에 정의된 메소드

설명

glo 클래스는 CUBRID의 **CALL METHOD** 문으로 호출하여 사용한다. **glo** 클래스에서는 클래스 메소드와 인스턴스 메소드가 운영된다.

glo 클래스에 정의된 두 메소드는 클래스의 인스턴스보다는 클래스 객체를 운영하는 데에 사용된다.

glo 인스턴스 메소드는 **glo** 데이터를 조작하기 위한 접근, 갱신 기능을 제공한다. 메소드의 인수(데이터 버퍼 등)는 CSQL 프로세서에서 명시할 수 없기 때문에 대부분의 인스턴스 메소드는 Java나 C와 같은 프로그래밍 언어 인터페이스에서 사용한다.

glo 클래스 메소드

new_fbo 클래스 메소드

이 메소드는 **glo** 인스턴스의 FBO (File Based Object) 타입을 생성한다. 메소드의 인수 값은 단일 문자열의 파일 경로이다. **new_fbo()** 메소드가 성공적으로 수행되면, 요청한 클래스의 새로운 인스턴스가 반환된다. FBO를 생성하기 위한 **new_fbo()** 메소드의 문법은 다음과 같다.

```
CALL new_fbo(:pathname)
ON CLASS glo INTO :write_object;
```

new_fbo() 클래스 메소드의 파일 경로명으로 환경 변수를 사용할 수도 있다. 따라서 서로 다른 파일 시스템을 사용하는 클라이언트가 동일한 FBO에 접속하는 것도 가능하다.

예를 들어, 다음과 같이 FBO 파일 경로를 주고, 어떤 사용자는 FBO_ROOT의 값을 **/usr/local**로 설정하고 다른 사용자는 **/net/edssel/usr/local**로 설정할 수 있다.

```
csql>CALL new_fbo('$FBO_ROOT/test/fbo1.txt')
ON CLASS glo INTO :fbo;
```

파일 경로는 CUBRID 환경 변수인 **CUBRID_FBO_PREFIX**를 수정하여 사용할 수도 있다.

CUBRID_FBO_PREFIX 변수에 값을 설정하면 파일에 접근할 때 모든 FBO 파일 경로명 앞에 붙는다. 이미 파일 경로명에 환경 변수 참조가 포함되어 있으면, 변수의 값이 앞에 붙지 않는다.

예를 들어, 다음의 경우 FBO 파일에 접근하는 경로명은 **"h:/remote1/usr/local/test/fbo1.txt"**이다.


```
%setenv CUBRID FBO_PREFIX "h:/remote1"
csql>CALL new_fbo('/usr/local/test/fbo1.txt')
        ON CLASS glo INTO :fbo;
```

NFS가 루트 파일 시스템이나 특정 서버 디렉토리에 마운트되고, 파일 경로명이 같다면, UNIX와 PC 클라이언트 모두 FBO의 내용에 접근할 수 있다. 위의 예에서, UNIX 시스템에서는 **CUBRID_FBO_PREFIX** 변수에 값이 없고, PC에서는 변수에 NFS 볼륨 식별자 "h:/remote1"를 명시해야 한다.

new_lo 클래스 메소드

이 메소드는 **glo** 인스턴스의 LO (Large Object)를 생성한다. 함수 인수는 없다. **new_lo()** 메소드는 원하는 클래스에 새로운 인스턴스를 반환한다.

```
csql>CALL new_lo() ON CLASS glo INTO :labeled_object;
```

new_lo_import() 클래스 메소드

이 메소드는 새로운 LO 인스턴스를 생성하거나 외부 파일로부터 그 내용을 초기화한다. 메소드 인수가 **NULL**이면 비어있는 LO를 생성하고, 메소드 인수가 파일명을 포함하는 문자열이면 새로운 인스턴스 생성 후 해당 파일의 내용을 복사하게 된다.

```
csql>CALL new_lo_import('./import data.txt')
        ON CLASS glo INTO :labeled_object;
```

glo 인스턴스 메소드

initialize_data() 메소드

이 메소드는 **glo** 클래스의 새로운 인스턴스를 생성할 때마다 **new_lo()** / **new_fbo()** 클래스 메소드에 의해 자동으로 호출된다. **glo** 클래스에서 기본 값으로 속성을 설정한다. **glo** 클래스로부터 상속받은 클래스는 다른 기본 속성을 설정하거나, 다른 연산을 수행하도록 **initialize_data()** 메소드를 재정의할 수 있다. 이 메소드는 일반적으로 사용자에게 의해 호출되지 않으므로 인수가 없으면 무시된다.

data_pos() 메소드

이 메소드는 **glo** 인스턴스 메소드에 의해 사용되는 **glo** 데이터 구역의 시작 위치를 반환한다. 다음은 **glo** 인스턴스로부터 현재 위치를 반환한다.

```
csql>CALL data_pos() ON :labeled_object INTO :location;
```

data_seek() 메소드

이 메소드는 첫 번째 인수 값을 현재 위치로 설정한다. **data_seek()** 메소드가 성공적으로 수행되면 새로운 위치가 반환된다. 다음은 **data_seek()** 메소드를 사용하여 **glo** 데이터 구역에서 현재 위치를 42로 설정하는 예이다.

```
csql>CALL data_seek(42) ON :labeled_object;
```

read_data() 메소드

이 메소드는 **glo**로부터 사용자 데이터 버퍼로 데이터를 전송한다. 데이터는 **glo** 데이터 내의 현재 위치부터 시작하여 읽게 된다. **read_data()** 메소드의 인수는 두 개이다. 첫 번째 인수는 상수로 **glo**로부터 읽어 들일 유닛의 개수를 명시하고, 두 번째 인수는 문자열로 데이터를 저장할 버퍼를 나타낸다. 메소드가 성공적으로 실행되면, 읽어 들인 실제 유닛의 개수와 **glo**로부터 읽어 들인 데이터를 반환한다. 현재 위치는 자동적으로 갱신된다. 다음은 **read_data()** 메소드를 사용한 C API 예이다. **glo** 데이터의 현재 위치로부터 1024개의 유닛을 읽어 들이고, 문자 배열 버퍼에 데이터를 전송한다.

```
...
DB INT32 rc, length = 1024;
DB_OBJECT *read_object;
char buffer[1024];
DB VALUE val1, val2, val3;
...
DB MAKE INTEGER(&val1, length);
DB MAKE VARCHAR(&val2, DB MAX VARCHAR PRECISION, buffer, length);
rc = db_send(read object, GLO METHOD READ, &val3, &val1, &val2);
...
```

참고 **read_data()** 메소드는 리턴 값이 반드시 문자 버퍼에 저장되어야 하기 때문에 CSQL에서 사용되는 것이 지원되지 않는다. 따라서 JDBC나 C API를 통해서만 사용이 가능하다.

write_data() 메소드

이 메소드는 사용자 데이터 버퍼로부터 **glo**로 데이터를 전송한다. 쓰기는 데이터가 존재하는 경우 **glo** 데이터의 현재 위치로부터 덮어쓰기를 한다. **write_data()** 메소드의 인수는 두 개이다. 첫 번째 인수는 **glo**에 저장할 유닛의 수(정수)이고 두 번째 인수는 저장될 데이터를 포함하고 있는 버퍼(문자열)이다. 실제 저장된 유닛의 개수를 반환하고, **glo** 데이터 내의 현재 위치는 자동으로 갱신된다. 다음은 **write_data()** 메소드를 이용한 예이다. LO를 만들고 데이터를 저장한다.

```
csql>CALL new lo() ON CLASS glo INTO :wr_lo
csql>CALL write_data(8,'raw data') TO :wr_lo;
```

insert_data() 메소드

이 메소드는 사용자 데이터 버퍼로부터 **glo** 내로 데이터를 삽입한다. 데이터는 현재 위치에 의해 정의된 위치를 시작으로 하여 삽입된다. **insert_data()** 메소드의 인수는 두 개이다. 첫 번째 인수는 **glo**로 삽입될 데이터의 유닛에 상응하는 정수이고, 두 번째 인수는 삽입될 데이터를 포함하고 있는 버퍼이다. 현재 위치는 자동으로 갱신되며, 다음 연산이 시작될 위치를 나타내게 된다. **insert_data()**가 성공적으로 수행되면 **glo**로 입력된 데이터의 양이 반환된다.

```
csql>CALL insert_data(3,'end') ON :wr_lo;
```

delete_data() 메소드

이 메소드는 현재 위치에서 시작하는 **glo** 데이터를 삭제한다. 메소드는 첫 번째 인수에 명시된 수만큼의 유닛을 제거한다. 삭제가 성공하면 제거된 유닛의 수가 반환된다. 삭제할 유닛의 수가 **glo** 데이터의 끝부분을 넘어가면 시스템은 절단(truncate)을 수행하고, 실제로 제거된 유닛의 수를 반환한다. 데이터가

삭제될 때 현재 위치는 변하지 않는다. **data_size()** 호출 후에 **delete_data()**를 호출하면 할당된 바이트 수가 출력된다. 다수의 삭제와 삽입 후에는 **glo**의 조각 모으기를 위해 **compress_data()** 메소드를 호출해야 한다.

```
csql>CALL delete_data(3) ON :wr_lo;
```

truncate_data() 메소드

이 메소드는 현재 위치에서부터 **glo** 데이터의 끝까지 모든 데이터를 제거한다. **truncate_data()** 메소드는 인수가 없다. 성공적으로 절단이 수행되면 삭제된 유닛의 수가 반환된다.

```
csql> call truncate_data() on :wr_lo into :num_deleted
```

append_data() 메소드

이 메소드는 **glo** 데이터의 끝 부분에 데이터를 추가한다. 메소드의 인수는 두 개이다. 첫 번째 인수는 추가할 유닛의 수를 명시한다. 두 번째 인수는 추가될 데이터를 가지고 있는 버퍼이다. 추가 연산이 성공적으로 수행되면 추가된 유닛의 수가 반환된다. 현재 위치는 추가된 **glo** 데이터의 끝으로 자동 갱신된다.

```
csql>call append_data( 14, 'appended_data' ) on :wr_lo
```

data_size() 메소드

이 메소드는 **glo**가 가지고 있는 모든 유닛의 개수를 반환한다. 갱신 메소드를 이용하여 **glo**에 데이터를 추가하거나 삭제할 수 있기 때문에 데이터의 크기는 변할 수 있다. 메소드의 사용 예는 다음과 같으며, 유닛의 총 개수는 **total_units** 변수에 저장되어 반환된다

```
csql>call data_size () on :wr_lo
```

compress_data() 메소드

이 메소드는 좀 더 효과적인 공간 사용을 위해 LO 데이터를 재구성한다. **compress_data()** 메소드는 다수의 **delete()** 또는 **insert()** 메소드 수행 후에 호출할 것을 권장한다. 메소드가 성공적으로 수행되면 0이 반환되고, 실패하면 0보다 작은 값이 반환된다.

```
csql>CALL compress_data() ON :wr_lo;
```

다수의 삭제와 삽입 후에는 **compress_data()** 를 수행하여 **glo** 조각 모으기를 해야 한다.

destroy_data() 메소드

이 메소드는 **glo** 데이터를 제거한다. 메소드는 **size** 속성, 현재 위치, **header_size** 속성을 모두 0으로 설정한 후 완료된다. 메소드가 성공적으로 수행되면 0을 반환하고, 그렇지 않으면 -1을 반환한다.

```
csql>CALL destroy_data() ON :wr_lo;
```

copy_from() 메소드

이 메소드는 데이터 원본(파일 또는 다른 **glo** 인스턴스)에서 **glo** 내로 데이터를 복사한다. **copy_from()** 메소드의 인수로는 파일 경로명을 가지는 문자열이나, 다른 **glo** 인스턴스가 사용될 수 있다. 메소드가 성공적으로 수행되면 기존의 **glo** 인스턴스에 덮어쓰기를 하고, 새로운 **glo**의 크기를 반환한다.

```
csql>CALL new lo() ON CLASS glo INTO :lob;
csql>CALL copy from('test.txt')
        ON :lob INTO :size;
```

참고 **glo**를 더 작은 데이터를 포함하는 새로운 파일로 덮어쓸 때, **glo**의 마지막에 원치 않는 데이터를 삭제하기 위해 **truncate_data()**를 호출하지 않는다면, 그 결과로 **glo** 객체는 원래 크기를 계속 유지하게 된다.

copy_to() 메소드

이 메소드는 **glo** 데이터로부터 파일이나 다른 **glo**로 데이터를 복사한다. **copy_to()** 메소드는 파일 경로명을 나타내는 문자열이나, 다른 **glo** 인스턴스를 인수로 갖는다. 메소드가 성공적으로 수행되면 복사된 데이터의 크기가 반환된다.

```
csql>CALL copy_to('extract.txt') ON :lob INTO :size_value;
```

glo_pathname() 메소드

이 메소드는 **glo** 인스턴스가 FBO일 경우에 경로명을 반환한다. **glo** 인스턴스가 LO이면 **NULL**을 반환한다. 이 메소드는 인수를 갖지 않는다. 다음 예에서 **glo**가 FBO이면 파일 경로명을, LO이면 **NULL**을 **file_name** 변수에 저장하여 반환한다.

```
csql> CALL glo_pathname() ON :fbo INTO :file_name;
```

get_error() 메소드

에러 발생시 에러 원인에 따른 값을 반환한다. 다음은 **get_error()**가 반환하는 에러 값이다.

```
INVALID STRING INPUT ARGUMENT -2      /* bad input argument */
INVALID INTEGER INPUT ARGUMENT -3      /* bad input argument */
INVALID_STRING_OR_OBJ_ARGUMENT -4      /* bad input argument */
INVALID_OBJECT_INPUT_ARGUMENT -5      /* bad input argument */
UNABLE_TO_FIND_GLO_STRUCTURE -6       /* internal error */
COULD_NOT_ACQUIRE_WRITE_LOCK -7      /* object already locked*/
ERROR_DURING_TRUNCATION -8            /* truncation error */
ERROR_DURING_DELETE -9                /* delete error */
ERROR_DURING_INSERT -10               /* insert error */
ERROR_DURING_WRITE -11                /* write error */
ERROR_DURING_READ -12                 /* read error */
ERROR_DURING_SEEK -13                 /* seek error */
ERROR_DURING_APPEND -14                /* append error */
ERROR_DURING_MIGRATE -15               /* migrate error */
COPY_TO_ERROR -16                      /* copy to error */
COPY_FROM_ERROR -17                   /* copy from error */
COULD_NOT_ALLOCATE_SEARCH_BUFFERS -18  /* search error */
COULD_NOT_COMPILE_REGULAR_EXPRESSION -19 /* reg exp error */
COULD_NOT_RESET_WORKING_BUFFER -20    /* search error */
```

get_error() 메소드의 예는 다음과 같다.

```
csql>CALL get_error() ON :fbo INTO :error_value;
```

set_error() 메소드

사용자에 의해 **glo** 클래스 계층이 확장된 경우 **set_error()** 메소드를 이용하여 **get_error()** 메소드에서 반환하는 에러 값을 추가할 수 있다.

```
csql> CALL set_error(:user_method_error) ON :fbo;
```

동시성

동시성 규칙은 다른 CUBRID 클래스 인스턴스에 적용되는 것과 같이 **glo** 인스턴스에도 동일하게 적용된다. **glo** 인스턴스에서 읽기 잠금을 공유함으로써 동시에 여러 사용자가 **glo** 데이터를 읽을 수 있다. 해당 **glo** 인스턴스에 다른 사용자가 쓰기 잠금을 가지고 있지 않으면, 읽기 잠금을 부여할 수 있다. 특정 인스턴스에 읽기 잠금이나 쓰기 잠금이 없다면, 트랜잭션이 수행되는 동안 **glo** 데이터에 대해 한 사용자에게 의해서만 갱신에 대한 쓰기 잠금이 허용된다.

glo에 대한 읽기와 쓰기 잠금은 모든 **glo**에 적용된다. **glo** 데이터의 부분 잠금은 지원하지 않는다.

FBO에는 한가지 특별한 동시성 조건이 적용된다. CUBRID 데이터베이스 시스템은 동일한 파일을 참조하는 **glo** 인스턴스를 생성하는 것을 허용한다.

모든 **glo** 인스턴스가 같은 경로명을 사용한다면, 같은 동시성 법칙이 적용된다. 그러나 절대 경로명을 사용하는 **glo** 인스턴스와 상대 경로명을 사용하는 **glo** 인스턴스가 같은 파일을 참조한다면, 두 인스턴스에 대해 일관적인 갱신 접근을 보장할 수 없다. 마찬가지로, 절대 경로명을 사용하는 **glo** 인스턴스와 연결(link)를 사용하는 **glo** 인스턴스가 같은 파일을 참조하는 경우에도 일관적인 갱신 접근을 보장할 수 없다.

CUBRID는 데이터베이스를 통해 FBO에 간접적으로 접근하기 위해서 동시성 제어를 지원한다. 그러나 외부 사용자나, 응용 프로그램에 의한 FBO의 직접적인 접근(또는 갱신)은 막을 수 없다.

복구

glo 복구 규칙은 다른 클래스 인스턴스처럼 강제적이다. 트랜잭션이 롤백되면, **glo**(LO와 FBO)의 어떤 갱신도 롤백된다. 트랜잭션이 커밋되었을 때 모든 갱신 **glo**는 영구적으로 저장된다.

만약 **ROLLBACK WORK**나 **COMMIT WORK** 문장을 수행 중에 데이터베이스 충돌이 발생하면, 데이터베이스가 재시작할 때 강제로 **glo**에 대한 복구 절차가 수행된다. LO에 대한 복구 절차는 재시작 시점에 내부적으로 데이터베이스 시스템에 의해 이루어진다. FBO 갱신에 대한 복구 절차는, 시스템에 충돌이 발생했을 때 FBO를 갱신한 사용자가 다시 데이터베이스에 접속했을 때 완료된다. 복구가 완료될 때까지 **glo**에 대한 읽기, 갱신 잠금은 데이터베이스가 소유하게 된다.

FBO 복구를 지원하기 위해 데이터베이스는 갱신되는 FBO에 대해 임시 파일(temporary shadow file)을 사용한다. 이는 FBO를 갱신하려는 사용자는 반드시 FBO가 존재하는 디렉토리에 갱신 권한이 있어야 함을 의미한다(또한 FBO 데이터의 복사본을 만들 수 있는 충분한 공간도 있어야 한다). FBO 파일이 위치한 디렉토리에 대해 쓰기 권한이 없는 경우 FBO 인스턴스에 대한 갱신은 실패하게 된다. 마찬가지로 **glo** 클래스나 서브클래스에 대해 갱신에 대한 데이터베이스 인가가 없으면 갱신 작업은 실패하게 된다.

CUBRID 시스템 카탈로그

개요

시스템 카탈로그 가상 클래스(virtual class)를 사용하면 다양한 스키마 정보를 SQL 문을 이용하여 쉽게 얻어낼 수 있다.

예를 들어 다음과 같은 스키마 정보들을 카탈로그 가상 클래스를 이용하여 얻을 수 있다.

```
-- 클래스 'db_user'를 참조하는 클래스들
SELECT class name
FROM db attribute
WHERE domain class name = 'db user';

-- 현재 사용자가 접근 가능한 클래스 개수
SELECT COUNT(*)
FROM db_class;

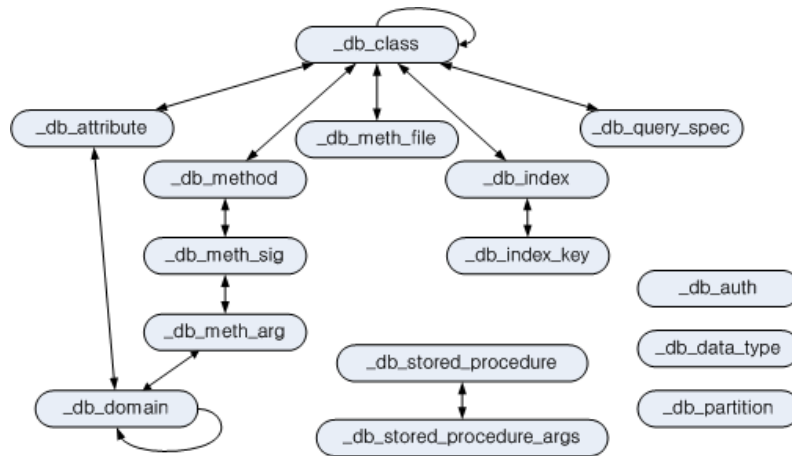
-- 클래스 'db_user'의 속성
SELECT attr_name, data_type
FROM db attribute
WHERE class_name = 'db_user';

-- 클래스 'glo'의 속성 중 상위 클래스로부터 상속받은 속성
SELECT attr name
FROM db attribute
WHERE class name = 'glo' AND
class_name <> from_class_name;
```

시스템 카탈로그 클래스

시스템 카탈로그 클래스

카탈로그 가상 클래스를 정의하기 위해 우선 카탈로그 클래스를 정의한다. 아래 그림은 추가되는 카탈로그 클래스들과 그 관계를 나타낸다. 화살표는 참조 관계이며, '_'로 시작하는 클래스는 카탈로그 클래스이다.



추가한 카탈로그 클래스들은 데이터베이스 안의 모든 클래스, 속성(attribute), 메소드 등에 대한 정보를 표현한다. 카탈로그 클래스들은 클래스 합성 계층(class composition hierarchy)으로 구성되어 있는데, 상호 참조가 가능하도록 카탈로그 클래스 인스턴스들의 OID를 가지도록 구성되어 있다.

_db_class

클래스에 대한 정보를 표현하며 class_name에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
class_of	object	클래스 객체로서 시스템 내부에 저장된 클래스에 대한 메타 정보 객체를 의미한다.
inst_attr_count	INTEGER	인스턴스 속성의 개수
shared_attr_count	INTEGER	공유 속성의 개수
inst_meth_count	INTEGER	인스턴스 메소드의 개수
class_meth_count	INTEGER	클래스 메소드의 개수
class_attr_count	INTEGER	클래스 속성의 개수
is_system_class	INTEGER	사용자가 정의한 클래스이면 0, 시스템 클래스이면 1
class_type	INTEGER	클래스이면 0, 가상 클래스이면 1
owner	db_user	클래스 소유자
class_name	VARCHAR(255)	클래스명
sub_classes	SEQUENCE OF _db_class	한 단계 하위 클래스
super_classes	SEQUENCE OF _db_class	한 단계 상위 클래스

inst_attrs	SEQUENCE OF _db_attribute	인스턴스 속성
shared_attrs	SEQUENCE OF _db_attribute	공유 속성
class_attrs	SEQUENCE OF _db_attribute	클래스 속성
inst_meths	SEQUENCE OF _db_method	인스턴스 메소드
class_meths	SEQUENCE OF _db_method	클래스 메소드
meth_files	SEQUENCE OF _db_methfile	메소드에 대한 함수가 위치한 파일 경로
query_specs	SEQUENCE OF _db_queryspec	가상 클래스인 경우 그 SQL 정의문
indexes	SEQUENCE OF _db_index	클래스에 생성된 인덱스

예제

다음 예제에서는 사용자 'PUBLIC'이 소유하고 있는 클래스의 모든 하위 클래스를 검색한다(결과로 보이는 하위 클래스 female_event는 [ADD SUPERCLASS 절](#)의 예제를 참조한다).

```
csql> select class_name, sequence(select class_name
csql>                                from _db_class s
csql>                                where s in c.sub_classes)
csql> from _db_class c
csql> where c.owner.name = 'PUBLIC' and
csql>        c.sub_classes is NOT NULL;
csql> ;x

=== <Result of SELECT Command in Line 1> ===

  class name          sequence((select class name from _db_class s where s in
c.sub_classes))
=====
  'event'             {'female_event'}
1 rows selected.
```

참고 시스템 카탈로그 클래스에 대한 모든 예제는 csql 프로그램에서 작성되었는데, 여기에서는 **--no-auto-commit** (auto-commit 모드 비활성화), **-u** (사용자 dba를 명시) 옵션을 사용하였다.

```
% csql --no-auto-commit -u dba demodb
```

_db_attribute

속성에 대한 정보를 표현하며 class_of, attr_name에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
class_of	_db_class	속성이 속한 클래스
attr_type	INTEGER	속성이 정의된 타입. 인스턴스 속성이면 0, 클래스 속성이면 1, 공유 속성이면 2
data_type	INTEGER	속성의 데이터 타입. 아래의 'CUBRID 가 지원하는 데이터 타입' 표에서 명시하는 value 중

		하나.
def_order	INTEGER	속성이 클래스에 정의된 순서로 0 부터 시작함. 상속받은 속성이면 그 상위 클래스에서 정의된 순서임. 예를 들어, 클래스 y 가 클래스 x 로부터 속성 a 를 상속받고 a 는 x 에서 첫 번째로 정의되었을 때 def_order 는 0 이 됨.
from_class_of	_db_class	상속받은 속성이면 그 속성이 정의되어 있는 상위 클래스가 설정되며, 상속받지 않은 것이면 NULL
from_attr_name	VARCHAR(255)	상속받은 속성이며, 이름 충돌(name conflict) 이 발생하여 이를 해결하기 위해 그 속성명이 바뀐 경우, 상위 클래스에 정의된 원래 이름이 설정됨. 그 이외에는 모두 NULL
attr_name	VARCHAR (255)	속성명
default_value	VARCHAR (255)	기본 값으로서 그 데이터 타입에 관계없이 모두 문자열로 저장. 기본 값이 없으면 NULL , 기본 값이 NULL 이면 'NULL'로 표현됨. 데이터 타입이 객체 타입이면 'volume id page id slot id', 집합 타입이면 '{element 1, element 2, ...}'로 표현됨.
domains	SEQUENCE OF _db_domain	데이터 타입에 대한 도메인 정보
is_nullable	INTEGER	not null 제약이 설정되어 있으면 0, 그렇지 않으면 1

CUBRID가 지원하는 데이터 타입

값	의미	값	의미
1	INTEGER	13	MONETARY
2	FLOAT	18	SHORT
3	DOUBLE	20	OID
4	STRING	22	NUMERIC
5	OBJECT	23	BIT
6	SET	24	VARBIT
7	MULTISET	25	CHAR
8	SEQUENCE	26	NCHAR
9	ELO	27	VARNCHAR

10	TIME	31	BIGINT
11	TIMESTAMP	32	DATETIME
12	DATE		

CUBRID가 지원하는 문자 세트

값	의미
0	US English – ASCII encoding
3	Latin 1 – ISO 8859 encoding
4	KSC 5601 1990 – EUC encoding

예제

다음 예제에서는 사용자 'PUBLIC'이 소유하고 있는 클래스 중에서 사용자 클래스(from_class_of.is_system_class = 0)인 것을 검색한다.

```
csql> select class of.class name, attr name
csql> from _db_attribute
csql> where class of.owner.name = 'PUBLIC' and
csql>         from class of.is_system_class = 0
csql> order by 1, def order;
csql> ;xrun

class of.class name  attr name
=====
'female_event'      'code'
'female_event'      'sports'
'female_event'      'name'
'female_event'      'gender'
'female_event'      'players'
```

_db_domain

도메인에 대한 정보이며 object_of에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
object_of	object	도메인을 참조하는 속성, 메소드 인자 또는 도메인
data_type	INTEGER	도메인의 데이터 타입(_db_attribute 의 'CUBRID가 지원하는 데이터 타입' 표의 '값' 중 하나)
prec	INTEGER	데이터 타입에 대한 전체 자릿수(precision). 전체 자릿수가 명시되지 않은 경우 0이 설정됨
scale	INTEGER	데이터 타입에 대한 소수점 이하의 자릿수(scale). 소수점 이하의 자릿수가 명시되지 않은 경우 0이 설정됨

class_of	_db_class	데이터 타입이 객체 타입인 경우 그 도메인 클래스. 객체 타입이 아닌 경우 NULL 이 설정됨.
code_set	INTEGER	문자 스트링 타입인 경우, 문자 세트 (_db_attribute 의 'CUBRID' 가 지원하는 문자 세트' 표의 '값' 중 하나). 문자 스트링 타입이 아닌 경우 0.
set_domains	SEQUENCE OF _db_domain	집합형 데이터 타입인 경우, 그 집합을 구성하는 원소의 데이터 타입에 대한 도메인 정보. 집합형 데이터 타입이 아닌 경우 NULL 이 설정됨

_db_method

메소드에 대한 정보이며 class_of, meth_name에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
class_of	_db_class	메소드가 속한 클래스
meth_type	INTEGER	메소드가 클래스에 정의된 타입. 인스턴스 메소드이면 0, 클래스 메소드이면 1
from_class_of	_db_class	메소드가 상속된 것이면 그 메소드가 정의되어 있는 상위 클래스가 설정되며 그렇지 않으면 NULL
from_meth_name	VARCHAR(255)	상속받은 메소드이며 이름 충돌이 발생하여 이를 해결하기 위해 그 메소드명이 바뀐 경우, 상위 클래스에 정의된 원래 이름이 설정됨. 그 이외에는 모두 NULL
meth_name	VARCHAR(255)	메소드 이름
signatures	SEQUENCE OF _db_meth_sig	메소드 호출시 수행하는 C 함수에 대한 구성 정보

예제

다음 예제에서는 사용자 'DBA' 가 소유하고 있는 클래스 중에서 클래스 메소드가 있는 것(c.class_meth_count > 0)의 클래스 메소드를 검색한다.

```
csql> select class_name, sequence(select meth_name
csql>                                from db.method m
csql>                                where m in c.class_meths)
csql> from _db class c
csql> where c.owner.name = 'DBA' and
csql>         c.class_meth_count > 0
csql> order by 1;
csql> ;xrun
```

```

=== <Result of SELECT Command in Line 1> ===

class name          sequence((select meth name from db method m where m in c.class
meths))
=====
'db authorization'  {'check authorization'}
'db authorizations' {'add_user', 'drop_user', 'find_user', 'print_authorizations',
'info', 'change owner', 'change trigger owner', 'get owner'}
'db root'           {'add_user', 'drop_user', 'find_user', 'print_authorizations',
'info', 'change owner', 'change trigger owner', 'get owner', 'change sp owner'}
'db_user'           {'add_user', 'drop_user', 'find_user', 'login'}
'glo'               {'new', 'new_lo', 'new_lo_import', 'new_fbo'}
'glo_holder'       {'new'}

```

_db_meth_sig

메소드에 대한 C 함수의 구성 정보이며 meth_of에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
meth_of	_db_method	함수 정보에 대한 메소드
arg_count	INTEGER	함수의 입력인자 개수
func_name	VARCHAR(255)	함수명
return_value	SEQUENCE OF _db_meth_arg	함수의 리턴 값
arguments	SEQUENCE OF _db_meth_arg	함수의 입력인자

_db_meth_arg

메소드 인자에 대한 정보이며 meth_sig_of에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
meth_sig_of	_db_meth_sig	인자가 속한 함수 정보
data_type	INTEGER	인자의 데이터 타입(_db_attribute 의 'CUBRID 가 지원하는 데이터 타입' 표의 '값' 중 하나)
index_of	INTEGER	함수정의에 인자가 나열된 순서. 리턴 값이면 0, 입력인자이면 1 부터 시작함.
domains	SEQUENCE OF _db_domain	인자의 도메인

_db_meth_file

메소드에 대한 함수가 정의된 파일 정보이며 class_of에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
class_of	_db_class	메소드 파일 정보가 속한 클래스
from_class_of	_db_class	파일 정보가 상속된 것이면 그 파일 정보가 정의되어 있는 상위 클래스가 설정되며, 그렇지 않으면 NULL
path_name	VARCHAR(255)	메소드가 위치한 파일의 경로

_db_query_spec

가상 클래스의 SQL 정의문이며 class_of에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
class_of	_db_class	가상 클래스에 대한 클래스 정보
spec	VARCHAR(4096)	가상 클래스에 대한 SQL 정의문

_db_index

인덱스에 대한 정보이며 class_of에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
class_of	_db_class	인덱스가 속한 클래스
index_name	varchar(255)	인덱스명
is_unique	INTEGER	고유 인덱스(unique index)이면 1, 그렇지 않으면 0
key_count	INTEGER	키를 구성하는 속성의 개수
key_attrs	SEQUENCE OF _db_index_key	키를 구성하는 속성들
is_reverse	INTEGER	역 인덱스(reverse index)이면 1, 그렇지 않으면 0
is_primary_key	INTEGER	기본 키이면 1, 그렇지 않으면 0
is_foreign_key	INTEGER	외래 키이면 1, 그렇지 않으면 0

예제

다음 예제에서는 클래스에 속하는 인덱스명을 검색한다.

```
csql> select class of.class name, index name
csql> from _db_index
csql> order by 1;
csql> ;xrun
```

```
=== <Result of SELECT Command in Line 1> ===
```

```

class of.class name    index name
=====
' db attribute'        'i__db attribute class of attr name'
' db auth'             'i__db auth grantee'
' db class'            'i__db class class name'
' db domain'           'i__db domain object of'
' db index'            'i__db index class of'
' db_index_key'        'i__db_index_key_index_of'
' db meth arg'         'i__db meth arg meth sig of'
' db meth file'        'i__db meth file class of'
' db meth sig'         'i__db meth sig meth of'
' db method'           'i__db method class of meth name'
' db partition'        'i__db partition class of pname'
' db_query_spec'       'i__db_query_spec class of'
' db stored procedure' 'u__db stored procedure sp name'
' db stored procedure_args' 'i__db stored procedure_args_sp_name'
' athlete'            'pk athlete code'
' db serial'           'pk db serial name'
' db user'             'i__db user name'
' event'               'pk event code'
' game'                'pk game host year event code athlete code'
' game'                'fk game_event code'
' game'                'fk game athlete code'
' history'             'pk_history_event_code_athlete'
' nation'              'pk nation code'
' olympic'             'pk olympic host year'
' participant'         'pk_participant_host_year_nation_code'
' participant'         'fk participant host year'
' participant'         'fk participant_nation_code'
' record'              'pk_record_host_year_event_code_athlete_code_medal'
' stadium'             'pk_stadium_code'

```

_db_index_key

인덱스에 대한 키 정보이며 index_of에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
index_of	_db_index	키 속성이 속하는 인덱스
key_attr_name	VARCHAR(255)	키를 구성하는 속성명
key_order	INTEGER	키에서 속성이 위치한 순서로 0 부터 시작함
asc_desc	INTEGER	속성 값의 순서가 내림차순이면 1, 그렇지 않으면 0
key_prefix_length	INTEGER	키로 사용할 prefix 의 길이

예제

다음 예제에서는 클래스에 속하는 인덱스명을 검색한다.

```

csql> select class of.class name, sequence(select key attr name
csql>                                     from _db_index_key k
csql>                                     where k in i.key attrs)
csql> from _db_index i
csql> where key count >= 2;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

class of.class name    sequence((select key_attr name from _db_index key k where k in
i.key attrs))
=====
' _db_partition'      {'class_of', 'pname'}

```

```

' db method'      {'class of', 'meth name'}
' db attribute'   {'class of', 'attr name'}
' participant'    {'host year', 'nation code'}
' game'           {'host year', 'event code', 'athlete code'}
' record'         {'host year', 'event code', 'athlete code', 'medal'}
' history'        {'event_code', 'athlete'}

```

_db_auth

클래스에 대한 사용자 권한 정보를 나타내며, grantee에 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
grantor	db_user	권한 부여자
grantee	db_user	권한 받은자
class_of	_db_class	권한부여 대상인 클래스 객체
auth_type	VARCHAR(7)	부여된 권한 타입 이름
is_grantable	INTEGER	권한 받은 클래스에 대해 다른 사용자에게 권한을 부여할 수 있으면 1, 아니면 0

CUBRID가 지원하는 권한 타입은 다음과 같다.

- **SELECT**
- **INSERT**
- **UPDATE**
- **DELETE**
- **ALTER**
- **INDEX**
- **EXECUTE**

예제

다음 예제에서는 클래스 'glo'에 정의되어 있는 권한 정보를 검색한다.

```

csql> select grantor.name, grantee.name, auth type
csql> from _db_auth
csql> where class of.class name = 'glo';
csql> ;xrun

```

=== <Result of SELECT Command in Line 1> ===

grantor.name	grantee.name	auth_type
'DBA'	'PUBLIC'	'SELECT'
'DBA'	'PUBLIC'	'INSERT'
'DBA'	'PUBLIC'	'UPDATE'
'DBA'	'PUBLIC'	'DELETE'
'DBA'	'PUBLIC'	'EXECUTE'

5 rows selected.

_db_data_type

CUBRID가 지원하는 데이터 타입([_db_attribute](#)의 'CUBRID가 지원하는 데이터 타입' 표 참조)을 나타낸다.

속성명	데이터 타입	설명
type_id	INTEGER	데이터 타입 식별자. 'CUBRID 가 지원하는 데이터 타입' 표의 '값'에 해당함
type_name	VARCHAR(9)	데이터 타입 이름. 'CUBRID 가 지원하는 데이터 타입' 표의 '의미'에 해당함

예제

다음 예제에서는 클래스 'event'의 속성과 각 타입명을 검색한다.

```
csql> select a.attr name, t.type name
csql> from db attribute a join db data type t
csql> on a.data_type = t.type_id
csql> where class of.class name = 'event'
csql> order by a.def_order;
csql> ;xrun
```

=== <Result of SELECT Command in Line 1> ===

attr name	type name
'code'	'INTEGER'
'sports'	'STRING'
'name'	'STRING'
'gender'	'CHAR'
'players'	'INTEGER'

_db_partition

분할에 대한 정보이며 class_of, pname에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
class_of	_db_class	Parent class 의 OID
pname	VARCHAR(255)	Parent - NULL
ptype	INTEGER	0 - HASH 1 - RANGE 2 - LIST
pexpr	VARCHAR(255)	Parent only
pvalues	SEQUENCE OF	Parent - 컬럼명, Hash size RANGE - MIN/MAX value - 무한의 MIN/MAX 는 NULL 로 저장 LIST - value list

_db_stored_procedure

Java 저장 함수에 대한 정보이며 sp_name에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
sp_name	VARCHAR(255)	SP 이름
sp_type	INTEGER	SP 종류 (function or procedure)
return_type	INTEGER	리턴 값 타입
arg_count	INTEGER	매개변수 개수
args	SEQUENCE OF _db_stored_procedure_args	매개변수 리스트
lang	INTEGER	구현 언어(현재로서는 Java)
target	VARCHAR(4096)	실행될 Java 메소드 이름
owner	db_user	소유자

_db_stored_procedure_args

Java 저장 함수 인자에 대한 정보이며 sp_name에 대한 인덱스가 생성되어 있다.

속성명	데이터 타입	설명
sp_name	VARCHAR(255)	SP 이름
index_of	INTEGER	매개변수 순서
arg_name	VARCHAR(255)	매개변수 이름
data_type	INTEGER	매개변수 데이터 타입
mode	INTEGER	모드 (IN, OUT, INOUT)

db_user

속성명	데이터 타입	설명
name	VARCHAR(1073741823)	사용자명
id	INTEGER	사용자 식별자
password	db_password	사용자 패스워드로 사용자에게 보여지지는 않는다.
direct_groups	SET OF db_user	사용자가 직접적으로 속한 그룹
groups	SET OF db_user	사용자가 직,간접적으로 속한 그룹

authorization	db_authorization	사용자가 가지고 있는 권한 정보
triggers	SEQUENCE OF object	사용자의 action 에 의해 발생하는 트리거들

메소드 이름

- **set_password()**
- **set_password_encoded()**
- **add_member()**
- **drop_member()**
- **print_authorizations()**
- **add_user()**
- **drop_user()**
- **find_user()**
- **login()**

db_authorization

속성명	데이터 타입	설명
owner	db_user	사용자 정보
grants	SEQUENCE OF object	{사용자가 권한 받은 객체, 객체의 권한 부여자, 권한 종류}의 sequence

메소드 이름

- **check_authorization**(varchar(255), integer)

db_trigger

속성명	데이터 타입	설명
owner	db_user	트리거 소유자
name	VARCHAR(1073741823)	트리거명
status	INTEGER	INACTIVE 이면 1, ACTIVE 이면 2. 기본 값은 2
priority	DOUBLE	트리거간의 수행순서에 대한 우선순위. 기본 값은 0
event	INTEGER	UPDATE 는 0, UPDATE STATEMENT 는 1, DELETE 는 2, DELETE STATEMENT 는 3, INSERT 는 4, INSERT STATEMENT 는 5, COMMIT 는 8, ROLLBACK 은 9 로 설정

target_class	object	트리거 대상(target)인 클래스에 대한 클래스 객체
target_attribute	VARCHAR(1073741823)	트리거 대상 속성명. 대상 속성이 명시되지 않으면 NULL 을 설정
target_class_attribute	INTEGER	대상 속성에 대해, 인스턴스 속성이면 0, 클래스 속성이면 1. 기본 값은 0
condition_type	INTEGER	INSERT, UPDATE, DELETE, CALL, EVALUATE 중 하나이면 1, REJECT 이면 2, INVALIDATE_TRANSACTION 이면 3, PRINT 이면 4
condition	VARCHAR(1073741823)	IF 문에 명시된 action 발생 조건
condition_time	INTEGER	BEFORE 는 1, AFTER 는 2, DEFERRED 는 3 으로 설정
action_type	INTEGER	INSERT, UPDATE, DELETE, CALL, EVALUATE 중 하나이면 1, REJECT 이면 2, INVALIDATE_TRANSACTION 이면 3, PRINT 이면 4
action_definition	VARCHAR(1073741823)	triggering 되는 수행문
action_time	INTEGER	BEFORE 는 1, AFTER 는 2, DEFERRED 는 3 으로 설정

db_ha_apply_info

시스템 카탈로그 가상 클래스

시스템 카탈로그 가상 클래스

일반 사용자는 자신이 권한을 가진 클래스에 대해서만 그 클래스와 관련된 정보들을 시스템 카탈로그 가상 클래스들을 통해 볼 수 있다.

이 절에서는 각 시스템 카탈로그 가상 클래스들이 어떤 정보를 표현하는지와 가상 클래스 정의문에 대해 설명한다.

DB_CLASS

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 클래스에 대한 정보를 보여준다.

속성명	데이터 타입	설명
class_name	VARCHAR (255)	클래스명

메모 [OH1]: 관리자 안내서 > CUBRID HA > HA 관련 시스템 카탈로그> db_ha_apply_info 를 가리키도록 하고, 제목을 삽입

owner_name	VARCHAR (255)	클래스 소유자명
class_type	VARCHAR (6)	클래스이면 'CLASS', 가상 클래스이면 'VCLASS'
is_system_class	VARCHAR (3)	시스템 클래스이면 'YES', 아니면 'NO'
partitioned	VARCHAR (3)	분할 그룹 클래스이면 'YES', 아니면 'NO'
is_reuse_oid_class	<u>VARCHAR (3)</u>	<u>REUSE_OID 클래스이면 'YES', 아니면 'NO'</u>

정의

```
CREATE VCLASS db_class (class_name, owner_name, class_type, is_system_class, partitioned,
is_reuse_oid_class)
AS
SELECT c.class_name, CAST(c.owner_name AS VARCHAR(255)),
CASE WHEN c.class_type = 0 THEN 'CLASS' WHEN 1 THEN 'VCLASS' WHEN 2 THEN 'PROXY'
ELSE 'UNKNOWN' END,
CASE WHEN MOD(c.is_system_class, 2) = 1 THEN 'YES' ELSE 'NO' END,
CASE WHEN c.sub_classes IS NULL THEN 'NO' ELSE NVL((SELECT 'YES' FROM db_partition
p WHERE p.class_of = c and p.pname IS NULL), 'NO') END
FROM db_class c
WHERE (CURRENT_USER = 'DBA' OR
(c.owner.name) SUBSETEQ (
SELECT SET(CURRENT_USER) + COALESCE(SUM(SET(t.g.name)), SET{})
FROM db_user u, TABLE(groups) AS t(g)
WHERE u.name = CURRENT_USER ) OR
(c) SUBSETEQ (
SELECT SUM(SET{au.class_of})
FROM db_auth au
WHERE {au.grantee.name} SUBSETEQ (
SELECT SET(CURRENT_USER) + COALESCE(SUM(SET(t.g.name)), SET{})
FROM db_user u, TABLE(groups) AS t(g)
WHERE u.name = CURRENT_USER ) AND
au.auth_type = 'SELECT')));
SELECT c.class_name, CAST(c.owner_name AS VARCHAR(255)),
CASE c.class_type WHEN 0 THEN 'CLASS' WHEN 1 THEN 'VCLASS' ELSE 'UNKNOWN' END,
CASE WHEN MOD(c.is_system_class, 2) = 1 THEN 'YES' ELSE 'NO' END,
CASE WHEN c.sub_classes IS NULL THEN 'NO' ELSE NVL((SELECT 'YES' FROM db_partition
p WHERE p.class_of = c and p.pname IS NULL), 'NO') END,
CASE WHEN MOD(c.is_system_class / 8, 2) = 1 THEN 'YES' ELSE 'NO' END
FROM db_class c
WHERE CURRENT_USER = 'DBA' OR
(c.owner.name) SUBSETEQ (
SELECT SET(CURRENT_USER) + COALESCE(SUM(SET(t.g.name)), SET{})
FROM db_user u, TABLE(groups) AS t(g)
WHERE u.name = CURRENT_USER ) OR
(c) SUBSETEQ (
SELECT SUM(SET{au.class_of})
FROM db_auth au
WHERE {au.grantee.name} SUBSETEQ (
SELECT SET(CURRENT_USER) + COALESCE(SUM(SET(t.g.name)), SET{})
FROM db_user u, TABLE(groups) AS t(g)
WHERE u.name = CURRENT_USER ) AND au.auth_type = 'SELECT');
```

예제

다음 예제에서는 현재 사용자가 소유하고 있는 클래스를 검색한다.

```
csql> select class_name
csql> from db_class
csql> where owner_name = CURRENT_USER;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

class_name
=====
```

```
'stadium'
'code'
'nation'
'event'
'athlete'
'participant'
'olympic'
'game'
'record'
'history'
'female_event'
```

참고 시스템 카탈로그 가상 클래스에 대한 모든 예제는 csql 프로그램에서 작성되었는데, 위의 예제는 사용자 옵션을 생략하였으며(생략시 기본 사용자는 **PUBLIC**), 그 외 별도의 언급이 없는 경우 **--no-auto-commit** (auto-commit 모드 비활성화), **-u** (사용자 **dba**를 명시) 옵션을 사용하였다.

```
% csql --no-auto-commit -u dba demodb
```

다음 예제에서는 현재 사용자가 접근할 수 있는 가상 클래스를 검색한다.

```
csql> select class_name
csql> from db class
csql> where class_type = 'VCLASS';
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

class name
=====
'db stored procedure args'
'db stored procedure'
'db_partition'
'db_trig'
'db_auth'
'db_index key'
'db_index'
'db_meth_file'
'db_meth_arg_setdomain elm'
'db_meth_arg'
'db_method'
'db_attr_setdomain elm'
'db_attribute'
'db_vclass'
'db_direct_super_class'
'db_class'
```

다음 예제에서는 현재 사용자가 접근할 수 있는 시스템 클래스를 검색한다. (사용자는 **PUBLIC**)

```
csql> select class_name
csql> from db class
csql> where is_system_class = 'YES' and
csql>         class_type = 'CLASS'
csql> order by 1;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

class name
=====
'db_authorization'
'db_authorizations'
'db_root'
'db_serial'
'db_user'
'glo'
```

DB_DIRECT_SUPER_CLASS

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 클래스에 대해 상위 클래스가 존재하면 그 클래스명을 보여준다.

속성명	데이터 타입	설명
class_name	VARCHAR (255)	클래스명
super_class_name	VARCHAR (255)	한 단계 상위 클래스명

정의

```
CREATE VCLASS db_direct_super_class (class_name, super_class_name)
AS
SELECT c.class_name, s.class_name
FROM _db_class c, TABLE(c.super_classes) AS t(s)
WHERE (CURRENT_USER = 'DBA' OR
      {c.owner.name} subseteq (
        SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
        from db_user u, table(groups) as t(g)
        where u.name = CURRENT_USER ) OR
      {c} subseteq (
        SELECT sum(set{au.class_of})
        FROM db_auth au
        WHERE {au.grantee.name} subseteq (
          SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
          from db_user u, table(groups) as t(g)
          where u.name = CURRENT_USER ) AND
          au.auth_type = 'SELECT'));
```

예제

- 다음 예제에서는 클래스 'female_event'의 상위 클래스를 검색한다. ([ADD SUPERCLASS 절 참조](#))

```
csql> select super_class_name
csql> from db_direct_super_class
csql> where class_name = 'female event';
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

  super_class_name
=====
  'event'
```

- 다음 예제에서는 현재 사용자가 소유하고 있는 클래스의 상위 클래스를 검색한다. (사용자는 **PUBLIC**)

```
csql> select c.class_name, s.super_class_name
csql> from db_class c, db_direct_super_class s
csql> where c.class_name = s.class_name and
csql>         c.owner_name = user
csql> order by 1;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

  class_name      super_class_name
=====
  'female event'  'event'
.
```

DB_VCLASS

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 가상 클래스들에 대해 그 SQL 정의문을 보여준다.

속성명	데이터 타입	설명
vclass_name	VARCHAR (255)	가상 클래스명
vclass_def	VARCHAR 4096)	가상 클래스의 SQL 정의문

정의

```
CREATE VCLASS db vclass (vclass name, vclass def)
AS
SELECT q.class of.class name, q.spec
FROM db query spec q
WHERE CURRENT USER = 'DBA' OR
      {q.class of.owner.name} subseteq (
        SELECT set{CURRENT USER} + coalesce(sum(set{t.g.name}), set{})
        from db user u, table(groups) as t(g)
        where u.name = CURRENT USER ) OR
      {q.class of} subseteq (
SELECT sum(set{au.class of})
FROM db_auth au
WHERE {au.grantee.name} subseteq (
  SELECT set{CURRENT USER} + coalesce(sum(set{t.g.name}), set{})
  from db user u, table(groups) as t(g)
  where u.name = CURRENT_USER ) AND
  au.auth_type = 'SELECT');
```

예제

다음 예제에서는 가상 클래스 'db_class'의 SQL 정의문을 검색한다.

```
csql> select vclass def
csql> from db vclass
csql> where vclass name = 'db class';
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

'SELECT c.class name, CAST(c.owner.name AS VARCHAR(255)), CASE c.class type WHEN 0 THEN
'CLASS' WHEN 1 THEN 'VCLASS' WHEN 2 THEN 'PROXY' ELSE 'UNKNOWN' END, CASE WHEN
MOD(c.is system class, 2) = 1 THEN 'YES' ELSE 'NO' END, CASE WHEN c.sub classes IS NULL
THEN 'NO' ELSE NVL((SELECT 'YES' FROM db partition p WHERE p.class of = c and p.pname IS
NULL), 'NO') END FROM db class c WHERE CURRENT USER = 'DBA' OR {c.owner.name} SUBSETEQ
( SELECT SET{CURRENT USER} + COALESCE(SUM(SET{t.g.name}), SET{}) FROM db user u,
TABLE(groups) AS t(g) WHERE u.name = CURRENT USER) OR {c} SUBSETEQ ( SELECT
SUM(SET{au.class of}) FROM db_auth au WHERE {au.grantee.name} SUBSETEQ ( SELECT
SET{CURRENT USER} + COALESCE(SUM(SET{t.g.name}), SET{}) FROM db_user u, TABLE(groups) AS
t(g) WHERE u.name = CURRENT_USER) AND au.auth_type = 'SELECT')
```

DB_ATTRIBUTE

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 클래스에 대해 그 속성 정보를 보여준다.

속성명	데이터 타입	설명
attr_name	VARCHAR (255)	속성명
class_name	VARCHAR (255)	속성이 속한 클래스명
attr_type	VARCHAR (8)	인스턴스 속성이면 'INSTANCE', 클래스 속성이면 'CLASS', 공유 속성이면 'SHARED'
def_order	INTEGER	클래스에서 속성이 정의된 순서로 0 부터 시작함.

		상속받은 속성이면 그 상위 클래스에서 정의된 순서임.
from_class_name	VARCHAR (255)	상속받은 속성이면 그 속성이 정의되어 있는 상위 클래스명이 설정되며, 그렇지 않으면 NULL
from_attr_name	VARCHAR (255)	상속받은 속성이며, 이름 충돌이 발생하여 이를 해결하기 위해 그 속성명이 바뀐 경우, 상위 클래스에 정의된 원래 이름임. 그 이외에는 모두 NULL
data_type	VARCHAR (9)	속성의 데이터 타입(db_attribute 의 'CUBRID'가 지원하는 데이터 타입' 표의 '의미' 중 하나)
prec	INTEGER	데이터 타입의 전체 자릿수. 전체 자릿수가 명시되지 않은 경우 0임
scale	INTEGER	데이터 타입의 소수점 이하의 자릿수. 소수점 이하의 자릿수가 명시되지 않은 경우 0임
code_set	INTEGER	스트링 타입인 경우, 문자 세트(db_attribute 의 'CUBRID'가 지원하는 문자 세트' 표의 '값' 중 하나). 스트링 타입이 아닌 경우 0.
domain_class_name	VARCHAR (255)	데이터 타입이 객체 타입인 경우 그 도메인 클래스명. 객체 타입이 아닌 경우 NULL
default_value	VARCHAR (255)	기본 값으로 그 데이터 타입에 관계없이 모두 문자열로 저장. 기본 값이 없으면 NULL , 기본 값이 NULL 이면 'NULL'로 표현됨. 데이터 타입이 객체 타입이면 'volume id page id slot id', 집합 타입이면 '{element 1, element 2, ...}'로 표현됨.
is_nullable	VARCHAR (3)	not null 제약이 설정되어 있으면 'NO', 그렇지 않으면 'YES'

정의

```
CREATE VCLASS db_attribute (
attr name, class name, attr type, def order, from class name, from attr name, data type,
prec, scale, code set, domain class name, default value, is nullable)
AS
SELECT a.attr name, c.class name,
CASE WHEN a.attr_type = 0 THEN 'INSTANCE'
      WHEN a.attr_type = 1 THEN 'CLASS'
      ELSE 'SHARED' END,
a.def_order, a.from_class_of.class_name, a.from_attr_name, t.type_name,
d.prec, d.scale, d.code set, d.class_of.class_name, a.default_value,
CASE WHEN a.is_nullable = 0 THEN 'YES' ELSE 'NO' END
FROM db_class c, db_attribute a, db_domain d, db_data_type t
WHERE a.class_of = c AND d.object_of = a AND d.data_type = t.type_id AND
(CURRENT_USER = 'DBA' OR
 {c.owner.name} subseteq (
SELECT set(CURRENT_USER) + coalesce(sum(set(t.g.name)), set{})
from db_user u, table(groups) as t(g)
where u.name = CURRENT_USER ) OR
 {c} subseteq (
SELECT sum(set{au.class_of})
```



```

FROM db auth au
WHERE {au.grantee.name} subseteq (
    SELECT set{CURRENT USER} + coalesce(sum(set{t.g.name}), set{})
    from db user u, table(groups) as t(g)
    where u.name = CURRENT USER ) AND
    au.auth_type = 'SELECT');

```

예제

- 다음 예제에서는 클래스 'event'의 속성과 각 데이터 타입을 검색한다.

```

csql> select attr_name, data_type, domain_class_name
csql> from db attribute
csql> where class_name = 'event'
csql> order by def_order;
csql> ;xrun

```

=== <Result of SELECT Command in Line 1> ===

attr name	data type	domain class name
'code'	'INTEGER'	NULL
'sports'	'STRING'	NULL
'name'	'STRING'	NULL
'gender'	'CHAR'	NULL
'players'	'INTEGER'	NULL

- 다음 예제에서는 클래스 'female_event'와 그 상위 클래스의 속성을 검색한다.

```

csql> select attr_name, from_class_name
csql> from db attribute
csql> where class_name = 'female event'
csql> order by def_order;
csql> ;xrun

```

=== <Result of SELECT Command in Line 1> ===

attr_name	from_class_name
'code'	'event'
'sports'	'event'
'name'	'event'
'gender'	'event'
'players'	'event'

- 다음 예제에서는 현재 사용자가 소유하고 있는 클래스 중에서 속성명이 'name'과 유사한 클래스를 검색한다. (사용자는 **PUBLIC**)

```

csql> select a.class_name, a.attr_name
csql> from db class c join db attribute a
csql> on c.class_name = a.class_name
csql> where c.owner_name = CURRENT USER and
csql> attr_name like '%name%'
csql> order by 1;
csql> ;xrun

```

=== <Result of SELECT Command in Line 1> ===

class_name	attr_name
'athlete'	'name'
'code'	'f_name'
'code'	's_name'
'event'	'name'
'female event'	'name'
'nation'	'name'
'stadium'	'name'

DB_ATTR_SETDOMAIN_ELM

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 클래스의 속성 중에서 그 데이터 타입이 집합 타입 (set, multiset, sequence)인 경우, 그 집합의 원소에 대한 데이터 타입을 보여준다.

속성명	데이터 타입	설명
attr_name	VARCHAR(255)	속성명
class_name	VARCHAR (255)	속성이 속한 클래스명
attr_type	VARCHAR (8)	인스턴스 속성이면 'INSTANCE', 클래스 속성이면 'CLASS', 공유 속성이면 'SHARED'
data_type	VARCHAR (9)	원소의 데이터 타입
Prec	INTEGER	원소의 데이터 타입에 대한 전체 자릿수
scale	INTEGER	원소의 데이터 타입에 대한 소수점 이하의 자릿수
code_set	INTEGER	원소의 데이터 타입이 문자 타입인 경우 그 문자집합
domain_class_name	VARCHAR (255)	원소의 데이터 타입이 객체 타입인 경우 그 도메인 클래스명

정의

```
CREATE VCLASS db_attr_setdomain elm (
attr name, class name, attr type,data type, prec, scale, code set, domain class name)
AS
SELECT a.attr name, c.class name,
CASE WHEN a.attr type = 0 THEN 'INSTANCE'
      WHEN a.attr_type = 1 THEN 'CLASS'
      ELSE 'SHARED' END,
et.type_name, e.prec, e.scale, e.code_set, e.class_of.class_name
FROM _db_class c, _db_attribute a, _db_domain d,
TABLE(d.set_domains) AS t(e), _db_data_type et
WHERE a.class of = c AND d.object of = a AND e.data type = et.type_id AND
(CURRENT USER = 'DBA' OR
 {c.owner.name} subseteq (
SELECT set{CURRENT USER} + coalesce(sum(set{t.g.name})), set{}}
from db_user u, table(groups) as t(g)
where u.name = CURRENT USER ) OR
 {c} subseteq (
SELECT sum(set{au.class of})
FROM db_auth au
WHERE {au.grantee.name} subseteq (
SELECT set{CURRENT USER} + coalesce(sum(set{t.g.name})), set{}}
from db_user u, table(groups) as t(g)
where u.name = CURRENT USER ) AND
au.auth_type = 'SELECT'));
```

가령 클래스 D의 속성 set_attr 이 SET(A, B, C) 타입이면 다음 세 개의 레코드가 존재하게 된다.

Attr_name	Class_name	Attr_type	Data_type	Prec	Scale	Code_set	Domain_class_name
'set_attr'	'D'	'INSTANCE'	'SET'	0	0	0	'A'
'set_attr'	'D'	'INSTANCE'	'SET'	0	0	0	'B'

'set_attr'	'D'	'INSTANCE'	'SET'	0	0	0	'C'
------------	-----	------------	-------	---	---	---	-----

예제

다음 예제에서는 클래스 'city'의 집합 타입의 속성과 각 데이터 타입을 검색한다. ([포함 연산자](#)에 정의한 city 테이블을 생성)

```
csql> select attr_name, attr_type, data_type, domain_class_name
csql> from db_attr setdomain elm
csql> where class_name = 'city';
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

attr_name      attr_type      data_type      domain_class_name
=====
'sports'       'INSTANCE'     'STRING'       NULL
```

DB_METHOD

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 클래스에 대해 그 메소드 정보를 보여준다.

속성명	데이터 타입	설명
meth_name	VARCHAR (255)	메소드명
class_name	VARCHAR (255)	메소드가 속한 클래스명
meth_type	VARCHAR (8)	인스턴스 메소드이면 'INSTANCE', 클래스 메소드이면 'CLASS'
from_class_name	VARCHAR (255)	상속받은 메소드이면 그 메소드가 정의되어 있는 상위 클래스명이 설정되며 그렇지 않으면 NULL
from_meth_name	VARCHAR (255)	상속받은 메소드이며, 이름 충돌이 발생하여 이를 해결하기 위해 그 메소드명이 바뀐 경우, 상위 클래스에 정의된 원래 이름이 설정됨. 그 이외에는 모두 NULL
func_name	VARCHAR (255)	메소드에 대한 C 함수명

정의

```
CREATE VCLASS db_method (
meth_name, class_name, meth_type, from_class_name, from_meth_name, func_name)
AS

SELECT m.meth name, m.class of.class name,
CASE WHEN m.meth_type = 0 THEN 'INSTANCE' ELSE 'CLASS' END,
m.from class of.class name, m.from meth name, s.func name
FROM db_method m, db_meth_sig s
WHERE s.meth_of = m AND
(CURRENT_USER = 'DBA' OR
 {m.class_of.owner.name} subseteq (
SELECT set(CURRENT_USER) + coalesce(sum(set{t.g.name}), set{})
from db_user u, table(groups) as t(g)
where u.name = CURRENT_USER ) OR
 {m.class_of} subseteq (
```

```
SELECT sum(set{au.class of})
      FROM db auth au
      WHERE {au.grantee.name} subseteq (
        SELECT set{CURRENT USER} + coalesce(sum(set{t.g.name})), set{}}
      from db user u, table(groups) as t(g)
      where u.name = CURRENT USER ) AND
      au.auth_type = 'SELECT');
```

예제

다음 예제에서는 클래스 'glo'의 메소드를 검색한다.

```
csql> select meth name, meth_type, func_name
csql> from db method
csql> where class name = 'glo'
csql> order by meth_type, meth_name;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

meth name          meth type          func name
=====
'new'              'CLASS'            'esm Glo create'
'new_fbo'          'CLASS'            'esm Glo create_fbo'
'new_lo'           'CLASS'            'esm Glo create_lo'
'new_lo import'    'CLASS'            'esm Glo import_lo'
'append data'      'INSTANCE'         'esm Glo append'
'binary search'    'INSTANCE'         'esm Glo binary search'
'compress data'    'INSTANCE'         'esm Glo compress'
'copy from'        'INSTANCE'         'esm Glo copy from'
'copy to'          'INSTANCE'         'esm Glo copy to'
'data pos'         'INSTANCE'         'esm Glo position'
'data_seek'        'INSTANCE'         'esm Glo seek'
'data size'        'INSTANCE'         'esm Glo size'
'delete_data'      'INSTANCE'         'esm Glo delete'
. . .
```

DB_METH_ARG

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 클래스의 메소드에 대해 그 입출력 인자 정보를 보여준다.

속성명	데이터 타입	설명
meth_name	VARCHAR (255)	메소드명
class_name	VARCHAR (255)	메소드가 속한 클래스명
meth_type	VARCHAR (8)	인스턴스 메소드이면 'INSTANCE', 클래스 메소드이면 'CLASS'
index_of	INTEGER	인자가 함수 정의에 나열된 순서. 리턴 값이면 0, 입력인자이면 1 부터 시작함
data_type	VARCHAR (9)	인자의 데이터 타입
prec	INTEGER	인자의 전체 자릿수
scale	INTEGER	인자의 소수점 이하의 자릿수
code_set	INTEGER	인자의 데이터 타입이 문자 타입인 경우 그 문자집합

domain_class_name	VARCHAR (255)	인자의 데이터 타입이 객체 타입인 경우 도메인 클래스명
-------------------	---------------	--------------------------------

정의

```
CREATE VCLASS db meth arg (
meth_name, class_name, meth_type,
index of, data_type, prec, scale, code_set, domain_class_name)
AS
SELECT s.meth of.meth name, s.meth of.class of.class name,
CASE WHEN s.meth_of.meth_type = 0 THEN 'INSTANCE' ELSE 'CLASS' END,
a.index of, t.type name, d.prec, d.scale, d.code set,
d.class of.class name
FROM db meth sig s, db meth arg a, db domain d, db data type t
WHERE a.meth sig of = s AND d.object of = a AND d.data type = t.type id AND
(CURRENT_USER = 'DBA' OR
{s.meth of.class of.owner.name} subseteq (
SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name})), set{}}
from db user u, table(groups) as t(g)
where u.name = CURRENT_USER ) OR
{s.meth of.class of} subseteq (
SELECT sum(set{au.class_of})
FROM db auth au
WHERE {au.grantee.name} subseteq (
SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name})), set{}}
from db_user u, table(groups) as t(g)
where u.name = CURRENT_USER ) AND
au.auth_type = 'SELECT'));
```

예제

다음 예제에서는 클래스 'glo'의 메소드 'append_data'의 입력 인자를 검색한다.

```
csql> select meth name, data type, prec
csql> from db_meth_arg
csql> where class name = 'glo' and
csql>      meth name = 'append_data' and
csql>      index of > 0
csql> order by index of;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

meth name          data type          prec
=====
'append_data'      'INTEGER'          0
'append_data'      'STRING'           1073741823
```

DB_METH_ARG_SETDOMAIN_ELM

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 클래스의 메소드에 대해 그 입/출력 인자의 데이터 타입이 집합 타입이면 그 집합의 원소에 대한 데이터 타입을 보여준다.

속성명	데이터 타입	설명
meth_name	VARCHAR(255)	메소드명
class_name	VARCHAR(255)	메소드가 속한 클래스명
meth_type	VARCHAR (8)	인스턴스 메소드이면 'INSTANCE', 클래스 메소드이면 'CLASS'

index_of	INTEGER	인자가 함수 정의에 나열된 순서. 리턴 값이면 0, 입력인자이면 1 부터 시작함
data_type	VARCHAR(9)	원소의 데이터 타입
prec	INTEGER	원소의 전체 자릿수
scale	INTEGER	원소의 소수점 이하의 자릿수
code_set	INTEGER	원소의 데이터 타입이 문자 타입인 경우 그 문자집합
domain_class_name	VARCHAR(255)	원소의 데이터 타입이 객체 타입인 경우 도메인 클래스명.

정의

```
CREATE VCLASS db_meth_arg_setdomain_elm(
meth name, class name, meth type,
index_of, data_type, prec, scale, code_set, domain_class_name)
AS
SELECT s.meth_of.meth_name, s.meth_of.class_of.class_name,
CASE WHEN s.meth_of.meth_type = 0 THEN 'INSTANCE' ELSE 'CLASS' END,
a.index_of, et.type_name, e.prec, e.scale, e.code_set,
e.class_of.class_name
FROM db_meth_sig s, db_meth_arg a, db_domain d,
TABLE(d.set_domains) AS t(e), db_data_type et
WHERE a.meth_sig_of = s AND d.object_of = a AND e.data_type = et.type_id AND
(CURRENT_USER = 'DBA' OR
{s.meth_of.class_of.owner.name} subseteq (
SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name})), set{}
from db_user u, table(groups) as t(g)
where u.name = CURRENT_USER ) OR
{s.meth_of.class_of} subseteq (
SELECT sum(set{au.class_of})
FROM db_auth au
WHERE {au.grantee.name} subseteq (
SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name})), set{}
from db_user u, table(groups) as t(g)
where u.name = CURRENT_USER ) AND
au.auth_type = 'SELECT'));
```

DB_METH_FILE

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 클래스에 대해 그 메소드가 정의된 파일 정보를 보여준다.

속성명	데이터 타입	설명
class_name	VARCHAR(255)	메소드 파일이 속한 클래스명
path_name	VARCHAR(255)	C 함수가 정의된 파일의 경로
from_class_name	VARCHAR(255)	상속받은 메소드이면 그 메소드 파일이 정의되어 있는 상위 클래스명이 설정. 그렇지 않으면 NULL

정의

```
CREATE VCLASS db meth file (class name, path name, from class name)
AS
SELECT f.class of.class name, f.path name, f.from class of.class name
FROM _db meth file f
WHERE (CURRENT_USER = 'DBA' OR
      {f.class of.owner.name} subseteq (
        SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name})), set{}
        from db_user u, table(groups) as t(g)
        where u.name = CURRENT_USER ) OR
      {f.class of} subseteq (
SELECT sum(set{au.class of})
FROM _db_auth au
WHERE {au.grantee.name} subseteq (
  SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name})), set{}
  from db_user u, table(groups) as t(g)
  where u.name = CURRENT_USER ) AND
  au.auth_type = 'SELECT'));
```

DB_INDEX

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 클래스에 대해 생성된 인덱스에 대한 정보를 보여준다.

속성명	데이터 타입	설명
index_name	VARCHAR(255)	인덱스명
is_unique	VARCHAR(3)	고유 인덱스이면 'YES', 그렇지 않으면 'NO'
is_reverse	VARCHAR(3)	역 인덱스(reverse indexd)이면 'YES', 그렇지 않으면 'NO'
class_name	VARCHAR(255)	인덱스가 속한 클래스명
key_count	INTEGER	키를 구성하는 속성의 개수
is_primary_key	VARCHAR(3)	기본 키이면 'YES', 그렇지 않으면 'NO'
is_foreign_key	VARCHAR(3)	외래 키이면 'YES', 그렇지 않으면 'NO'

정의

```
CREATE VCLASS db index (index name, is unique, is reverse, class name, key count,
is primary key, is foreign key)
AS
SELECT i.index name, CASE WHEN i.is unique = 0 THEN 'NO' ELSE 'YES' END,
CASE WHEN i.is reverse = 0 THEN 'NO' ELSE 'YES' END, i.class of.class name, i.key count,
CASE WHEN i.is primary key = 0 THEN 'NO' ELSE 'YES' END, CASE WHEN i.is foreign key = 0
THEN 'NO' ELSE 'YES' END
FROM db index i
WHERE (CURRENT_USER = 'DBA' OR
      {i.class of.owner.name} subseteq (
        SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name})), set{}
        from db_user u, table(groups) as t(g)
        where u.name = CURRENT_USER ) OR
      {i.class of} subseteq (
SELECT sum(set{au.class of})
FROM _db_auth au
WHERE {au.grantee.name} subseteq (
  SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name})), set{}
  from db_user u, table(groups) as t(g)
```

```
where u.name = CURRENT USER ) AND
      au.auth_type = 'SELECT'));
```

예제

다음 예제에서는 클래스의 인덱스 정보를 검색한다.

```
csql> select class_name, index_name, is_unique
csql> from db index
csql> order by 1;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

  class name          index name          is unique
=====
'athlete'            'pk athlete code'    'YES'
'city'               'pk city_city_name'  'YES'
'db serial'          'pk db serial name'  'YES'
'db user'            'i db user name'     'NO'
'event'              'pk event code'      'YES'
'female_event'       'pk event code'      'YES'
'game'               'pk game host year event code athlete code' 'YES'
'game'               'fk game_event_code' 'NO'
'game'               'fk game_athlete code' 'NO'
'history'            'pk history event code athlete' 'YES'
'nation'             'pk nation code'     'YES'
'olympic'            'pk olympic host year' 'YES'
'participant'        'pk participant host year nation code' 'YES'
'participant'        'fk participant host year' 'NO'
'participant'        'fk participant nation code' 'NO'
'record'             'pk record host year event code athlete code medal' 'YES'
'stadium'            'pk stadium_code'    'YES'
...
```

DB_INDEX_KEY

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 클래스에 대해 생성된 인덱스에 대한 키 정보를 보여준다.

속성명	데이터 타입	설명
index_name	VARCHAR(255)	인덱스명
class_name	VARCHAR(255)	인덱스가 속한 클래스명
key_attr_name	VARCHAR(255)	키를 구성하는 속성의 이름
key_order	INTEGER	키에서 속성이 위치한 순서. 0 부터 시작함
asc_desc	VARCHAR(4)	속성 값의 순서가 내림차순이면 'DESC', 그렇지 않으면 'ASC'
key_prefix_length	INTEGER	키로 사용할 prefix 의 길이

정의

```
CREATE VCLASS db index key (index name, class name, key attr name, key order,
key prefix length)
AS
SELECT k.index of.index name, k.index of.class of.class name, k.key attr name, k.key order
CASE k.asc_desc
WHEN 0 THEN 'ASC'
```



```

WHEN 1 THEN 'DESC' ELSE 'UNKN' END,
k.key prefix length
FROM db index key k
WHERE (CURRENT USER = 'DBA' OR
{k.index of.class of.owner.name}
subsetq (
SELECT set{CURRENT USER} + coalesce(sum(set{t.g.name})), set{}}
from db user u, table(groups) as t(g)
where u.name = CURRENT_USER ) OR {k.index of.class of}
subsetq (
SELECT sum(set{au.class_of})
FROM db auth au
WHERE {au.grantee.name} subsetq (
SELECT set{CURRENT USER} + coalesce(sum(set{t.g.name})), set{}}
from db user u, table(groups) as t(g)
where u.name = CURRENT_USER ) AND
au.auth_type = 'SELECT'));

```

예제

다음 예제에서는 클래스의 인덱스 키 정보를 검색한다.

```

csql> select class name, key attr name, index name
csql> from db index key
csql> order by class name, key order;
csql> ;xrun

=== <Result of SELECT Command in Line 1> ===

'athlete'          'code'             'pk athlete code'
'city'             'city name'        'pk city city name'
'db_serial'        'name'             'pk db serial name'
'db_user'          'name'             'i_db_user_name'
'event'            'code'             'pk event code'
'female_event'     'code'             'pk event code'
'game'             'host year'        'pk game host year event code athlete code'
'game'             'event_code'       'fk_game_event_code'
'game'             'athlete_code'     'fk_game_athlete_code'
...

```

DB_AUTH

데이터베이스 내에서 현재 사용자가 권한을 가지는 클래스에 대한 권한 정보를 보여준다.

속성명	데이터 타입	설명
grantor_name	VARCHAR(255)	권한을 부여한 사용자명
grantee_name	VARCHAR(255)	권한을 받은 사용자명
class_name	VARCHAR(255)	권한부여 대상인 클래스명
auth_type	VARCHAR(7)	부여된 권한 타입명
is_grantable	VARCHAR(3)	권한 받은 클래스에 대해 다른 사용자에게 권한을 부여할 수 있으면 'YES', 아니면 'NO'

정의

```

CREATE VCLASS db_auth (grantor_name, grantee_name, class_name, auth_type, is_grantable )
AS
SELECT CAST(a.grantor.name AS VARCHAR(255)),
CAST(a.grantee.name AS VARCHAR(255)),
a.class_of.class_name, a.auth_type,
CASE WHEN a.is_grantable = 0 THEN 'NO' ELSE 'YES' END

```

```

FROM db_auth a
WHERE (CURRENT_USER = 'DBA' OR
      {a.class_of.owner.name} subseteq (
        SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name})), set{}
        from db_user u, table(groups) as t(g)
        where u.name = CURRENT_USER ) OR
      {a.class_of} subseteq (
SELECT sum(set{au.class_of})
      FROM db_auth au
      WHERE {au.grantee.name} subseteq (
        SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name})), set{}
        from db_user u, table(groups) as t(g)
        where u.name = CURRENT_USER ) AND
        au.auth_type = 'SELECT'));

```

예제

다음 예제에서는 이름이 'db_a'로 시작되는 클래스의 권한 정보를 검색한다.

```

csql> select class_name, auth_type, grantor_name
csql> from db_auth
csql> where class_name like 'db_a%'
csql> order by 1;
csql> ;xrun

```

=== <Result of SELECT Command in Line 1> ===

class name	auth type	grantor name
'db_attr_setdomain_elm'	'SELECT'	'DBA'
'db_attribute'	'SELECT'	'DBA'
'db_auth'	'SELECT'	'DBA'
'db_authorization'	'EXECUTE'	'DBA'
'db_authorization'	'SELECT'	'DBA'
'db_authorizations'	'EXECUTE'	'DBA'
'db_authorizations'	'SELECT'	'DBA'

DB_TRIG

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 클래스나 그 소속 속성을 대상(target)으로 하는 트리거 정보를 보여준다.

속성명	데이터 타입	설명
trigger_name	VARCHAR(255)	트리거명
target_class_name	VARCHAR(255)	대상이 되는 클래스
target_attr_name	VARCHAR(255)	대상이 되는 속성으로서 트리거에 명시되지 않으면 NULL
target_attr_type	VARCHAR(8)	대상이 속성으로 명시될 경우, 인스턴스 속성이면 'INSTANCE', 클래스 속성이면 'CLASS'.
action_type	INTEGER	INSERT, UPDATE, DELETE, CALL, EVALUATE 중 하나이면 1, REJECT 이면 2, INVALIDATE_TRANSACTION 이면 3, PRINT 이면 4
action_time	INTEGER	BEFORE 는 1, AFTER 는 2, DEFERRED 는 3 으로 설정

예제

- 데이터베이스 내에서 현재 사용자가 접근 권한을 가진 클래스나 그 소속 속성을 대상(target)으로 하는 트리거 정보를 보여준다.

```
CREATE VCLASS db trig (
trigger name, target class name, target attr name, target attr type, action type,
action_time)
AS
SELECT CAST(t.name AS VARCHAR(255)), c.class name,
       CAST(t.target attribute AS VARCHAR(255)),
       CASE WHEN t.target class attribute = 0 THEN 'INSTANCE' ELSE 'CLASS' END,
       t.action_type, t.action_time
FROM db_class c, db_trigger t
WHERE t.target_class = c.class_of AND
      (CURRENT_USER = 'DBA' OR
       {c.owner.name} subseteq (
         SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
         from db user u, table(groups) as t(g)
         where u.name = CURRENT_USER ) OR
       {c} subseteq (
SELECT sum(set{au.class_of})
FROM db_auth au
WHERE {au.grantee.name} subseteq (
  SELECT set{CURRENT_USER} + coalesce(sum(set{t.g.name}), set{})
  from db user u, table(groups) as t(g)
  where u.name = CURRENT_USER ) AND
  au.auth_type = 'SELECT'));
```

- 다음 예제에서는 클래스의 트리거 정보를 검색한다.

```
csql> select target class name, trigger name
csql> from db_trig
csql> order by 1;
csql> ;xrun
```

=== <Result of SELECT Command in Line 1> ===

```
target class name      trigger name
=====
'glo'                  'glo_delete_contents'
```

DB_PARTITION

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 분할 클래스에 대한 정보를 보여준다.

속성명	데이터 타입	설명
class_name	VARCHAR(255)	클래스명
partition_name	VARCHAR(255)	파티션명
partition_class_name	VARCHAR(255)	파티션 클래스 명
partition_type	VARCHAR(32)	파티션 타입 (HASH, RANGE, LIST)
partition_expr	VARCHAR(255)	파티션 표현식
partition_values	SEQUENCE OF	RANGE – MIN/MAX value - 무한의 MIN/MAX 는 NULL LIST - value list

정의

```
CREATE VCLASS db partition
(sp_name, sp_type, return_type, arg_count, lang, target, owner)
AS
SELECT p.class_of.class_name AS class_name, p.pname AS partition_name,
       p.class_of.class_name || ' p ' || p.pname AS partition_class_name,
       CASE WHEN p.ptype = 0 THEN 'HASH'
            WHEN p.ptype = 1 THEN 'RANGE'
            ELSE 'LIST' END AS partition_type,
       TRIM(SUBSTRING( pi.pexpr FROM 8 FOR (POSITION(' FROM ' IN pi.pexpr)-8))) AS
       partition_expression,
       p.pvalues AS partition_values
FROM   db partition p,
       ( select * from   db partition sp
where sp.class_of =  p.class_of AND sp.pname is null) pi
WHERE p.pname is not null AND
      ( CURRENT_USER = 'DBA'
      OR
        {p.class_of.owner.name} SUBSETEQ
        ( SELECT SET(CURRENT_USER) + COALESCE(SUM(SET{t.g.name}), SET{})
          FROM db user u, TABLE(groups) AS t(g)
          WHERE u.name = CURRENT_USER
        )
      OR
        {p.class_of} SUBSETEQ
        ( SELECT SUM(SET{au.class_of})
          FROM   db auth au
          WHERE {au.grantee.name} SUBSETEQ
                ( SELECT SET(CURRENT_USER) + COALESCE(SUM(SET{t.g.name}), SET{})
                  FROM db user u, TABLE(groups) AS t(g)
                  WHERE u.name = CURRENT_USER) AND
                  au.auth_type = 'SELECT'
                )
        )
      )
)
```

예제

다음 예제에서는 participant2 클래스의 현재 구성된 분할 정보를 조회한다. ([영역 분할 정의](#)의 예제 참조)

```
csql> select * from db partition where class name = 'participant2';
csql> ;x

=== <Result of SELECT Command in Line 2> ===

  class_name      partition_name      partition_class_name      partition_type
  partition_expr   partition_values
=====
'participant2'    'before 2000'      'participant2_p_before 2000' 'RANGE'
'host_year'       {NULL, 2000}
'participant2'    'before 2008'      'participant2_p_before 2008' 'RANGE'
'host_year'       {2000, 2008}
```

DB_STORED_PROCEDURE

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 Java 저장 함수에 대한 정보를 보여준다.

속성명	데이터 타입	설명
sp_name	VARCHAR(255)	SP 이름
sp_type	VARCHAR(16)	SP 종류 (function or procedure)
return_type	VARCHAR(16)	리턴 값 타입

arg_count	INTEGER	매개변수 개수
lang	VARCHAR(16)	구현 언어(현재로서는 JAVA)
target	VARCHAR(4096)	실행될 Java 메소드 이름
owner	VARCHAR(256)	소유자

정의

```
CREATE VCLASS db_stored_procedure
(sp name, sp type, return type, arg count, lang, target, owner)
AS
SELECT sp.sp_name,
       CASE sp.sp_type WHEN 1 THEN 'PROCEDURE'
                       ELSE 'FUNCTION' END,
       CASE WHEN sp.return_type = 0 THEN 'void'
             WHEN sp.return_type = 28 THEN 'CURSOR'
       ELSE ( SELECT dt.type name
               FROM _db_data_type dt
               WHERE sp.return_type = dt.type_id) END,
       sp.arg_count,
       CASE sp.lang WHEN 1 THEN 'JAVA'
       ELSE '' END, sp.target, sp.owner.name
FROM _db_stored_procedure sp
```

예제

다음 예제에서는 현재 사용자가 소유하고 있는 Java 저장 함수를 조회한다.

```
csql> select sp name, target from db_stored_procedure
csql> where sp type = 'FUNCTION'
csql> and owner = CURRENT USER
csql> ;x

=== <Result of SELECT Command in Line 3> ===

  sp_name                      target
=====
'hello'                        'SpCubrid.HelloCubrid() return java.lang.String'
'sp_int'                       'SpCubrid.SpInt(int) return int'
```

DB_STORED_PROCEDURE_ARGS

데이터베이스 내에서 현재 사용자가 접근 권한을 가진 Java 저장 함수의 인자에 대한 정보를 보여준다.

속성명	데이터 타입	설명
sp_name	VARCHAR(255)	SP 이름
index_of	INTEGER	매개변수 순서
arg_name	VARCHAR(256)	매개변수 이름
data_type	VARCHAR(16)	매개변수 데이터 타입
mode	VARCHAR(6)	모드 (IN, OUT, INOUT)

정의

```
CREATE VCLASS db_stored_procedure_args (sp name, index of, arg name, data_type, mode)
AS
```

```
SELECT sp.sp_name, sp.index_of, sp.arg_name,
       CASE sp.data_type WHEN 28 THEN 'CURSOR'
       ELSE ( SELECT dt.type_name FROM db.data_type dt
              WHERE sp.data_type = dt.type_id) END,
       CASE WHEN sp.mode = 1 THEN 'IN' WHEN sp.mode = 2 THEN 'OUT'
       ELSE 'INOUT' END
FROM db.stored_procedure_args sp
ORDER BY sp.sp_name, sp.index_of ;
```

예제

다음 예제에서는 'phone_info' Java 저장 프로시저의 인수 정보를 순서대로 조회한다.

```
csql> select index_of, arg_name, data_type, mode
csql> from db.stored_procedure_args
csql> where sp_name = 'phone_info'
csql> order by index_of
csql> ;x
```

=== <Result of SELECT Command in Line 3> ===

index_of	arg_name	data_type	mode
0	'name'	'STRING'	'IN'
1	'phoneno'	'STRING'	'IN'

카탈로그 클래스/가상 클래스 사용 권한

카탈로그 클래스들은 **dba** 소유로 생성된다. 그러나, **dba**가 **SELECT** 연산만 수행할 수 있을 뿐이며, **UPDATE/DELETE** 등의 연산을 수행할 경우에는 authorization failure 에러가 발생한다. 일반 사용자는 시스템 카탈로그 클래스에 대해서 질의를 수행할 수 없다.

카탈로그 가상 클래스는 **dba** 소유로 생성되지만 모든 사용자가 카탈로그 가상 클래스에 대해 **SELECT** 문을 수행할 수 있다. 물론 카탈로그 가상 클래스에 대한 **UPDATE/DELETE** 연산은 수행이 불가능하다.

카탈로그 클래스/가상 클래스에 대한 갱신은 사용자가 클래스/속성/인덱스/사용자/권한을 생성/변경/삭제하는 DDL 문을 수행할 경우 시스템에 의해 자동으로 수행된다.

카탈로그 정보의 일관성

카탈로그 정보는 카탈로그 클래스/가상 클래스의 인스턴스에 의해 표현된다. **READ UNCOMMITTED INSTANCES** (TRAN_REP_CLASS_UNCOMMIT_INSTANCE 혹은 TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE) 격리 수준에서 이러한 인스턴스를 접근할 경우 정확하지 않은(변경 중인) 값을 읽을 수도 있다. 따라서, 정확한 카탈로그 정보를 얻기 위해서는 **READ COMMITTED INSTANCES** 격리 수준 이상에서 카탈로그 클래스/가상 클래스에 대해 **SELECT** 질의를 사용해야 한다.

카탈로그에 대한 질의

클래스, 가상 클래스, 속성, 트리거, 메소드, 인덱스명 등과 같은 식별자(identifier)는 모두 소문자로 변경되어 시스템 카탈로그에 저장된다. 따라서 시스템 카탈로그 클래스에 대해 대상 식별자를 검색하려면 소문자를 사용해야 한다.

```

CREATE TABLE Foo(name varchar(255));
SELECT class name, partitioned FROM db class WHERE class name = 'Foo';

=== <Result of SELECT Command in Line 1> ===

There are no results.

0 rows selected.

SELECT class name, partitioned FROM db class WHERE class name = 'foo';

=== <Result of SELECT Command in Line 1> ===

  class name  partitioned
=====
'foo'        'NO'
1 rows selected.

```


관리자 안내서

관리자 안내서는 데이터베이스 관리자(**DBA**)가 CUBRID 시스템을 사용하는데 필요한 작업 방법을 설명한다. 데이터베이스 생성 및 삭제, 볼륨 추가와 같은 데이터베이스 관리 작업, 데이터베이스를 다른 곳으로 이동하거나 시스템 버전에 맞춰서 변경하는 마이그레이션 작업, 장애 대비를 위한 데이터베이스의 백업 및 복구 작업 등에 대한 내용을 포함한다.

그리고 CUBRID 서버, 브로커 및 매니저 서버 등의 다양한 프로세스들을 구동하고 정지하는 CUBRID 유틸리티의 사용법을 설명한다.

이 장에서 설명하는 주요 내용은 다음과 같다.

- CUBRID 유틸리티
- CUBRID 제어
- 데이터베이스 관리
- 데이터베이스 마이그레이션
- 데이터베이스 백업 및 복구
- 데이터베이스 복제

CUBRID 유틸리티

CUBRID 유틸리티는 CUBRID 서비스를 통합 관리할 수 있는 기능을 제공하며, CUBRID 서비스 프로세스를 관리하는 서비스 관리 유틸리티와, 데이터베이스를 관리하는 데이터베이스 관리 유틸리티로 구분된다.

서비스 관리 유틸리티는 다음과 같다.

- 서비스 유틸리티: 마스터 프로세스를 구동 및 관리한다.
 - cubrid service
- 서버 유틸리티: 서버 프로세스를 구동 및 관리한다.
 - cubrid server
- 브로커 유틸리티: 브로커 프로세스 및 응용서버(CAS) 프로세스를 구동 및 관리한다.
 - cubrid broker
- 매니저 유틸리티: 매니저 서버 프로세스를 구동 및 관리한다.
 - cubrid manager
- 복제 유틸리티: 복제 프로세스를 구동 및 관리한다.
 - cubrid repl_server
 - cubrid repl_agent

자세한 설명은 [CUBRID 서비스](#)를 참조한다.

데이터베이스 관리 유틸리티는 다음과 같다.

- 데이터베이스 생성/볼륨 추가/삭제 유틸리티
 - cubrid createdb
 - cubrid addvoldb
 - cubrid deletedb
- 데이터베이스 이름 변경/호스트 변경/복사/설치 유틸리티
 - cubrid renamedb
 - cubrid alterdbhost
 - cubrid copydb
 - cubrid installdb
- 데이터베이스 공간 확인/공간 정리 유틸리티
 - cubrid spacedb
 - cubrid compactdb
- 데이터베이스 질의 계획 확인/최적화 유틸리티
 - cubrid plandump
 - cubrid optimizedb

- cubrid statdump
- 데이터베이스 잠금 확인/트랜잭션 제거/일관성 확인 유틸리티
 - cubrid lockdb
 - cubrid killtran
 - cubrid checkdb
- 데이터베이스 진단/로그 재생성 유틸리티
 - cubrid diagdb
 - cubrid emergency_patchlog
 - cubrid paramdump
- 데이터베이스 적재 유틸리티
 - cubrid loaddb
 - cubrid unloaddb
- 데이터베이스 백업/복구 유틸리티
 - cubrid backupdb
 - cubrid restoredb
- HA 유틸리티
 - cubrid changemode
 - cubrid copylogdb
 - cubrid applylogdb

자세한 설명은 [데이터베이스 관리](#)를 참조한다.

프롬프트에서 cubrid를 입력하면 다음과 같은 구문 안내문이 출력된다.

```
% cubrid

cubrid utility, version R3.0
usage: cubrid <utility-name> [args]
Type 'cubrid <utility-name>' for help on a specific utility.

Available service's utilities:
  service
  server
  broker
  manager
  repl_server
  repl_agent
  heartbeat

Available administrator's utilities:
  addvoldb
  alterdbhost
  backupdb
  checkdb
  compactdb
  copydb
  createdb
  deletedb
  diagdb
  emergency_patchlog
  installdb
  killtran
  loaddb
```

```
lockdb  
optimizedb  
plandump  
renamedb  
restoredb  
spacedb  
unloaddb  
paramdump  
statdump  
changemode  
copylogdb  
applylogdb
```

cubrid is a tool for DBMS.
For additional information, see <http://www.cubrid.com>

CUBRID 제어

CUBRID 유틸리티 사용법(구문)

CUBRID 유틸리티의 사용법(구문)은 다음과 같다.

CUBRID 서비스 제어

CUBRID 설정 파일에 등록된 서비스를 제어하기 위한 **cubrid** 유틸리티 구문은 다음과 같다. *command*에 올 수 있는 명령어는 서비스 구동을 위한 **start**, 종료를 위한 **stop**, 재시작을 위한 **restart**, 상태 확인을 위한 **status** 중 하나이며, 추가로 입력해야 할 옵션이나 인수는 없다.

```
cubrid service command
command : { start | stop | restart | status }
```

데이터베이스 서버 제어

데이터베이스 서버 프로세스를 제어하기 위한 **cubrid** 유틸리티 구문은 다음과 같다. *command*에 올 수 있는 명령어는 서버 프로세스 구동을 위한 **start**, 종료를 위한 **stop**, 재시작을 위한 **restart**, 상태 확인을 위한 **status**가 있으며, **status**를 제외한 명령어에는 데이터베이스 이름이 인수로 지정되어야 한다.

```
cubrid server command [<database_name>]
command : { start | stop | restart | status }
```

브로커 제어

CUBRID 브로커 프로세스를 제어하기 위한 **cubrid** 유틸리티 구문은 다음과 같다. *command*로 올 수 있는 명령어는 브로커 프로세스 구동을 위한 **start**, 종료를 위한 **stop**, 재시작을 위한 **restart**, 상태 확인을 위한 **status**가 있으며, 특정 브로커 구동을 위한 **on**, 종료를 위한 **off**가 있다.

```
cubrid broker command
command : { start | stop | restart | status [<broker_name>] | on <broker_name> | off
<broker_name> }
```

CUBRID 매니저 서버 제어

CUBRID 매니저를 사용하기 위해서는 데이터베이스 서버가 실행된 곳에 매니저 서버가 실행되어야 한다.

CUBRID 매니저 프로세스를 제어하기 위한 **cubrid** 유틸리티 구문은 다음과 같다. *command*로 올 수 있는 명령어는 매니저 서버 프로세스 구동을 위한 **start**, 종료를 위한 **stop**, 상태 확인을 위한 **status**가 있다.

```
cubrid manager command
command : { start | stop | status }
```

복제 프로세스 제어

CUBRID 복제 프로세스를 제어하기 위한 **cubrid** 유틸리티 구문은 다음과 같다.

```
cubrid repl_server command
command : { start <master_database_name> <server_network_port> [-a <max_agents>] [-e
<error_file_name>] | stop <master_database_name> | status }
```

```
cubrid repl_agent command
command : { start <dist database name> [<dba password>] | stop <dist database name> |
status }
```

CUBRID 서비스

서비스 등록

사용자는 임의로 데이터베이스 서버, CUBRID 브로커, CUBRID 매니저 중 하나 이상을 데이터베이스 환경 설정 파일(cubrid.conf)에 CUBRID 서비스로 등록할 수 있다. 사용자가 별도로 서비스를 등록하지 않으면, 기본적으로 마스터 프로세스만 등록된다. CUBRID 서비스에 등록되어 있으면 **cubrid service** 유틸리티를 사용해서 한번에 관련된 프로세스들을 모두 구동, 정지하거나 상태를 알아볼 수 있어 편리하다. 다음은 데이터베이스 환경 설정 파일에서 데이터베이스 서버와 브로커를 서비스로 등록하고, CUBRID 서비스 구동과 함께 **demodb**와 **testdb**라는 데이터베이스를 자동으로 시작하도록 설정한 예이다.

```
%vi cubrid.conf
#
# Copyright (C) 2008 Search Solution Corporation. All rights reserved by Search Solution.
#
# $Id$
#
# cubrid.conf
#....

[service]

# The list of processes to be started automatically by 'cubrid service start' command
# Any combinations are available with server, broker and manager.
service=server,broker

# The list of database servers in all by 'cubrid service start' command.
# This property is effective only when the above 'service' property contains 'server'
keyword.
server=demodb,testdb
```

서비스 구동 및 종료

서비스 구동

Linux 환경에서는 CUBRID 설치 후 CUBRID 서비스 구동을 위해 아래와 같이 입력한다. 데이터베이스 환경 설정 파일에서 서비스를 등록하지 않으면 기본적으로 마스터 프로세스만 구동된다.

Windows 환경에서는 시스템 권한을 가진 사용자로 로그인한 경우에만 아래의 명령이 정상 수행된다. 관리자 또는 일반 사용자는 CUBRID 매니저 설치 후 작업 표시줄에 생성되는 CUBRID 서비스 트레이 아이콘을 클릭하여 CUBRID Server를 구동 또는 정지할 수 있다.

```
% cubrid service start
@ cubrid master start
++ cubrid master start: success
```

이미 마스터 프로세스가 구동 중이라면 다음과 같은 메시지가 표시된다.

```
% cubrid service start
@ cubrid master start
++ cubrid master is running.
```

마스터 프로세스의 구동에 실패한 경우라면 다음과 같은 메시지가 표시된다. 다음은 데이터베이스 환경 설정 파일(cubrid.conf)에 설정된 cubrid_port_id 파라미터 값이 충돌하여 구동에 실패한 예이다.

```
% cubrid service start
@ cubrid master start
cub master: '/tmp/CUBRID1523' file for UNIX domain socket exist.... Operation not
permitted
++ cubrid master start: fail
```

[서비스 등록](#)에 설명된 대로 서비스를 등록한 후, 서비스를 구동하기 위해 다음과 같이 입력한다. 마스터 프로세스, 데이터베이스 서버 프로세스, 브로커 및 등록된 demodb, testdb가 한번에 구동됨을 확인할 수 있다.

```
% cubrid service start
@ cubrid master start
++ cubrid master start: success
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0.....

++ cubrid server start: success
@ cubrid server start: testdb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0.....

++ cubrid server start: success
@ cubrid broker start
++ cubrid broker start: success
```

서비스 종료

CUBRID 서비스를 종료하려면 다음과 같이 입력한다. 사용자에 의해 등록된 서비스가 없는 경우, 마스터 프로세스만 종료된다.

```
% cubrid service stop
@ cubrid master stop
++ cubrid master stop: success
```

등록된 CUBRID 서비스를 종료하려면 다음과 같이 입력한다. demodb, testdb는 물론, 서버 프로세스, 브로커 프로세스, 마스터 프로세스가 모두 종료됨을 확인할 수 있다.

```
% cubrid service stop
@ cubrid server stop: demodb
Server demodb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid server stop: testdb
Server testdb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid broker stop
++ cubrid broker stop: success
@ cubrid master stop
++ cubrid master stop: success
```

서비스 재구동

CUBRID 서비스를 재구동하려면 다음과 같이 입력한다. 사용자에 의해 등록된 서비스가 없는 경우, 마스터 프로세스만 종료 후 재구동된다.

```
% cubrid service restart
@ cubrid master stop
++ cubrid master stop: success
@ cubrid master start
++ cubrid master start: success
```

등록된 CUBRID 서비스를 다음과 같이 입력한다. demodb, testdb는 물론, 서버 프로세스, 브로커 프로세스, 마스터 프로세스가 모두 종료된 후 재구동되는 것을 확인할 수 있다.

```
% cubrid service restart
@ cubrid server stop: demodb
Server demodb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid server stop: testdb
Server testdb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid broker stop
++ cubrid broker stop: success
@ cubrid master stop
++ cubrid master stop: success
@ cubrid master start
++ cubrid master start: success
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0.....

++ cubrid server start: success
@ cubrid server start: testdb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0.....

++ cubrid server start: success
@ cubrid broker start
++ cubrid broker start: success
```

서비스 상태 관리

등록된 마스터 프로세스, 데이터베이스 서버 및 복제 서버의 상태를 확인하기 위하여 다음과 같이 입력한다.

```
% $ cubrid service status
@ cubrid master status
++ cubrid master is running.
@ cubrid server status

Server testdb (rel 8.2, pid 31059)
Server demodb (rel 8.2, pid 30950)

@ cubrid broker status
% query editor - cub cas [15464,40000]
/homel/cubrid22/CUBRID/log/broker//query_editor.access
/homel/cubrid22/CUBRID/log/broker//query_editor.err
JOB QUEUE:0, AUTO ADD APPL SERVER:ON, SQL LOG MODE:ALL:100000
LONG TRANSACTION TIME:60.00, LONG QUERY TIME:60.00, SESSION TIMEOUT:300
KEEP CONNECTION:AUTO, ACCESS MODE:RW
-----
ID   PID   QPS   LQS PSIZE STATUS
-----
1 15465    0    0 48032 IDLE
2 15466    0    0 48036 IDLE
3 15467    0    0 48036 IDLE
4 15468    0    0 48036 IDLE
5 15469    0    0 48032 IDLE
@ cubrid manager server status
```



```
++ cubrid manager server is not running.  
@ cubrid replication status
```

만약, 마스터 프로세스가 중지된 상태라면, 다음과 같은 메시지가 출력된다.

```
% cubrid service status  
@ cubrid master status  
++ cubrid master is not running.
```

데이터베이스 서버

데이터베이스 서버 구동 및 종료

데이터베이스 서버 구동

demodb 서버를 구동하기 위하여 다음과 같이 입력한다.

```
% cubrid server start demodb  
@ cubrid server start: demodb  
  
This may take a long time depending on the amount of recovery works to do.  
  
CUBRID 2008 R3.0  
  
++ cubrid server start: success
```

마스터 프로세스가 중지된 상태에서 demodb 서버를 시작하면 다음과 같이 자동으로 마스터 프로세스를 구동한 후 지정된 데이터베이스 서버를 구동한다.

```
% cubrid server start demodb  
@ cubrid master start  
++ cubrid master start: success  
@ cubrid server start: demodb  
  
This may take a long time depending on the amount of recovery works to do.  
  
CUBRID 2008 R3.0  
  
++ cubrid server start: success
```

이미 demodb 서버가 구동 중인 상태라면 다음과 같은 메시지가 출력된다.

```
% cubrid server start demodb  
@ cubrid server start: demodb  
++ cubrid server 'demodb' is running.
```

데이터베이스 서버 종료

demodb 서버 구동을 종료하기 위하여 다음과 같이 입력한다.

```
% cubrid server stop demodb  
@ cubrid server stop: demodb  
Server demodb notified of shutdown.  
This may take several minutes. Please wait.  
++ cubrid server stop: success
```

이미 demodb 서버가 종료된 상태라면, 다음과 같은 메시지가 출력된다.

```
% cubrid server stop demodb  
@ cubrid server stop: demodb  
++ cubrid server 'demodb' is not running.
```

데이터베이스 서버 재구동

demodb 서버를 재구동하기 위하여 다음과 같이 입력한다. 이미 구동 중인 demodb 서버를 중지시킨 후 재구동하는 것을 알 수 있다.

```
% cubrid server restart demodb
@ cubrid server stop: demodb
Server demodb notified of shutdown.
This may take several minutes. Please wait.
++ cubrid server stop: success
@ cubrid server start: demodb

This may take a long time depending on the amount of recovery works to do.

CUBRID 2008 R3.0

++ cubrid server start: success
```

데이터베이스 상태 확인

데이터베이스 서버의 상태를 확인하기 위하여 다음과 같이 입력한다. 구동 중인 모든 데이터베이스 서버의 이름이 표시된다.

```
% cubrid server status
@ cubrid server status
Server testdb (rel 8.2, pid 24465)
Server demodb (rel 8.2, pid 24342)
```

마스터 프로세스가 중지된 상태라면, 다음과 같은 메시지가 출력된다.

```
% cubrid server status
@ cubrid server status
++ cubrid master is not running.
```

브로커

브로커 구동 및 종료

브로커를 구동하기 위하여 다음과 같이 입력한다.

```
% cubrid broker start
@ cubrid broker start
++ cubrid broker start: success
```

이미 브로커가 구동 중이라면 다음과 같은 메시지가 출력된다..

```
cubrid broker start
@ cubrid broker start
++ cubrid broker is running.
```

브로커를 종료하기 위하여 다음과 같이 입력한다.

```
% cubrid broker stop
@ cubrid broker stop
++ cubrid broker stop: success
```

이미 브로커가 종료되었다면 다음과 같은 메시지가 출력된다.

```
% cubrid broker stop
@ cubrid broker stop
++ cubrid broker is not running.
```

브로커 상태 확인

설명

cubrid broker status는 여러 옵션을 제공하여, 각 브로커의 처리 완료된 작업 수, 처리 대기중인 작업 수를 포함한 브로커 상태 정보를 확인할 수 있도록 한다.

구문

*args*가 주어지면 해당 브로커에 대한 상태 모니터링을 수행하고, 생략되면 CUBRID 브로커 환경 설정 파일(**cubrid_broker.conf**)에 등록된 전체 브로커에 대해 상태 모니터링을 수행한다.

```
cubrid broker status options [args]
options : [ -b | -q | -s secs | -t | -f ]
args : broker_name
```

옵션

다음은 결합할 수 있는 옵션에 관해 설명한 표이다.

옵션	설명
<i>broker_name</i>	지정한 브로커에 관한 상태 정보를 출력한다. 옵션이 지정되지 않으면 전체 브로커의 상태 정보를 출력한다.
-b	응용 서버에 관한 정보는 포함되지 않고, 브로커에 관한 상태 정보만 출력한다.
-q	작업 큐에 대기 중인 작업을 출력한다.
-s secs	브로커에 관한 상태 정보를 지정된 시간마다 주기적으로 출력한다. q 를 입력하면 명령 프롬프트로 복귀한다.
-t	화면 출력시 tty mode 로 출력한다. 출력 내용을 리다이렉션하여 파일로 쓸 수 있다.
-f	브로커가 접속한 DB 및 호스트 정보를 출력한다.

예제

전체 브로커 상태 정보 확인하기 위하여 옵션 및 인수를 입력하지 않으면 다음과 같이 출력된다.

```
% cubrid broker status
@ cubrid broker status
% query_editor - cub_cas [28433,40820] /home/CUBRID/log/broker/query_editor.access
/home/CUBRID/
JOB QUEUE:0, AUTO ADD APPL SERVER:ON, SQL LOG MODE:ALL:100000
LONG TRANSACTION TIME:60, LONG_QUERY TIME:60, SESSION_TIMEOUT:300
KEEP CONNECTION:AUTO, ACCESS MODE:RW
-----
ID   PID   QPS   LQS PSIZE STATUS
-----
1 28434    0    0 50144 IDLE
2 28435    0    0 50144 IDLE
3 28436    0    0 50144 IDLE
4 28437    0    0 50140 IDLE
5 28438    0    0 50144 IDLE
% broker1 - cub_cas [28443,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
```

```
JOB QUEUE:0, AUTO ADD APPL SERVER:ON, SQL LOG MODE:ALL:100000
LONG TRANSACTION TIME:60, LONG QUERY TIME:60, SESSION TIMEOUT:300
KEEP CONNECTION:AUTO, ACCESS MODE:RW
```

ID	PID	QPS	LQS	PSIZE	STATUS
1	28444	0	0	50144	IDLE
2	28445	0	0	50140	IDLE
3	28446	0	0	50144	IDLE
4	28447	0	0	50144	IDLE
5	28448	0	0	50144	IDLE

- % query_editor : 브로커의 이름
- cub_cas : CUBRID 응용 서버의 형태
- [28433, 40820] : PID와 브로커 접속 포트 번호
- /home/CUBRID/log/broker/query_editor.access : query_editor의 접속 로그 파일의 위치 정보
- JOB_QUEUE : 작업 큐에 대기 중인 작업 개수
- AUTO_ADD_APPL_SERVER : 자동으로 응용 서버가 추가되도록 cubrid_broker.conf의 AUTO_ADD_APPL_SERVER 파라미터 값이 ON임
- SQL_LOG_MODE : SQL 로그를 남기도록 cubrid_broker.conf 파일의 SQL_LOG 파라미터 값이 ALL임
- LONG_TRANSACTION_TIME : 장기 실행(Long-duration) 트랜잭션으로 판단하는 트랜잭션의 실행 시간임. 트랜잭션의 실행시간이 60초를 넘으면 장기 실행 트랜잭션임.
- LONG_QUERY_TIME : 장기 실행 질의(Long-duration query)으로 판단하는 질의의 실행 시간임. 질의의 실행 시간이 60초를 넘으면 장기 실행 질의임.
- SESSION_TIMEOUT : 세션 타임 아웃값으로서, cubrid_broker.conf의 SESSION_TIMEOUT 파라미터 값이 300임
- KEEP_CONNECTION : 응용 서버와 클라이언트의 연결이 AUTO가 되도록 cubrid_broker.conf 파일의 KEEP_CONNECTION 파라미터 값이 AUTO임
- ACCESS_MODE : 브로커의 동작 모드. RW는 데이터베이스 조회 뿐만 아니라 수정도 가능한 모드임.
- ID : 브로커 ID 또는 작업 큐에 대기 중인 작업 ID
- PID : 브로커의 프로세스 ID
- QPS : 초당 처리된 질의의 수
- LQS : 초당 처리되는 장기 실행 질의의 수
- PSIZE : 응용 서버 프로세스 크기
- STATUS : 응용 서버의 현재 상태로서, BUSY/ IDLE/ CLIENT_WAIT/ CLOSE_WAIT 있음

브로커에 관한 상태 정보를 확인하려면 다음과 같이 입력한다.

```
% cubrid broker status -b
@ cubrid broker status
NAME          PID  PORT  AS  JQ    REQ  TPS  QPS  LONG-T  LONG-Q  ERR-Q
=====
* query_editor 4094 30000  5   0     0    0    0    0/60    0/60    0
* broker1     4104 33000  5   0     0    0    0    0/60    0/60    0
```

- NAME : 브로커 이름
- PID : 브로커의 프로세스 ID
- PORT : 브로커의 포트 번호

- AS : 응용 서버 개수
- JQ : 작업 큐에서 대기 중인 작업 개수
- REQ : 브로커가 처리한 클라이언트 요청 개수
- TPS : 초당 처리된 트랜잭션의 수(옵션이 " -b -s < sec > " 일 때만 계산됨)
- QPS : 초당 처리된 질의의 수(옵션이 " -b -s < sec > " 일 때만 계산됨)
- LONG-T : LONG_TRANSACTION_TIME 시간을 초과한 트랜잭션 수 / LONG_TRANSACTION_TIME 파라미터의 값
- LONG-Q : LONG_QUERY_TIME 시간을 초과한 질의의 수 / LONG_QUERY_TIME 파라미터의 값
- ERR-Q : 에러가 발생한 질의의 수

-q 옵션을 이용하여 broker1을 포함하는 이름을 가진 브로커의 상태 정보를 확인하고, 해당 브로커의 작업 큐에 대기 중인 작업 상태를 확인하기 위하여 다음과 같이 입력한다. 인수로 broker1을 입력하지 않으면 모든 브로커에 대하여 작업 큐에 대기 중인 작업 리스트가 출력된다.

```
% cubrid broker status -q broker1
@ cubrid broker status
% broker1 - cub_cas [28443,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
JOB_QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
-----
ID    PID    QPS    LQS PSIZE STATUS
-----
1 28444      0      0 50144 IDLE
2 28445      0      0 50140 IDLE
3 28446      0      0 50144 IDLE
4 28447      0      0 50144 IDLE
5 28448      0      0 50144 IDLE
```

-s 옵션을 이용하여 broker1을 포함하는 이름을 가진 브로커의 모니터링 주기를 입력하고, 주기적으로 브로커의 상태를 모니터링하기 위해 다음과 같이 입력한다. 인수로 broker1을 입력하지 않으면 모든 브로커에 대하여 상태 모니터링이 주기적으로 수행된다. 또한, q를 입력하면 모니터링 화면에서 명령 프롬프트로 복귀한다.

```
% cubrid broker status -s 5 broker1
% broker1 - cub_cas [28443,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
JOB_QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
-----
ID    PID    QPS    LQS PSIZE STATUS
-----
1 28444      0      0 50144 IDLE
2 28445      0      0 50140 IDLE
3 28446      0      0 50144 IDLE
4 28447      0      0 50144 IDLE
5 28448      0      0 50144 IDLE
```

-t 옵션을 사용하여, TPS와 QPS 정보를 파일로 출력한다. 파일로 출력하는 것을 중단하려면 <Ctrl+C>를 눌러서 프로그램을 정지시킨다.

```
% cubrid broker status -b -t -s 1 > log_file
```

-b 옵션과 **-s** 옵션을 이용하여 모든 브로커의 TPS와 QPS를 포함한 상태 모니터링을 주기적으로 수행할 경우 다음과 같이 입력한다.

```
% cubrid broker status -b -s 1
NAME          PID  PORT  AS  JQ      REQ  TPS  QPS  LONG-T  LONG-Q  ERR-Q
=====
* query editor 28433 40820  5   0        0   0   0   0/60    0/60    0
* broker1      28443 40821  5   0        0   0   0   0/60    0/60    0
```

-f 옵션을 사용하여 브로커가 연결한 서버/데이터베이스 정보와 최근 접속 시간, 그리고 CAS에 접속하는 클라이언트의 IP 주소를 확인하기 위해, 다음과 같이 입력한다.

```
$ cubrid broker status -f broker1
@ cubrid broker status
% broker1 - cub_cas [28443,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
JOB_QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
-----
ID  PID  QPS  LQS PSIZE STATUS          LAST ACCESS TIME  DB  HOST  LAST
CONNECT TIME      CLIENT IP
-----
1  26946  0    0 51168 IDLE           2009/11/06 16:06:41  -  -
-      10.0.1.101
2  26947  0    0 51172 IDLE           2009/11/06 16:06:41  -  -
-      10.0.1.101
3  26948  0    0 51172 IDLE           2009/11/06 16:06:41  -  -
-      10.0.1.101
4  26949  0    0 51172 IDLE           2009/11/06 16:06:41  -  -
-      10.0.1.101
5  26950  0    0 51172 IDLE           2009/11/06 16:06:41  -  -
-      10.0.1.101
```

특정 브로커 관리

broker1만 구동하기 위하여 다음과 같이 입력한다. 단, broker1은 이미 공유 메모리에 설정된 브로커이다.

```
% cubrid broker on broker1
```

만약, broker1이 공유 메모리에 설정되지 않은 상태라면 다음과 같은 메시지가 출력된다.

```
% cubrid broker on broker1
Cannot open shared memory
```

broker1만 종료하기 위하여 다음과 같이 입력한다. 이 때, broker1의 서비스 풀을 함께 제거할 수 있다.

```
% cubrid broker off broker1
```

broker1을 재시작하기 위하여 다음과 같이 입력한다.

```
% cubrid broker restart broker1
```

브로커 파라미터의 동적 변경

설명

브로커 구동과 관련된 파라미터는 브로커 환경 설정 파일(cubrid_broker.conf)에서 설정할 수 있다. 자세한 내용은 "성능 관리 안내서"의 [브로커별 파라미터](#)를 참조한다. 그 밖에, broker_changer 유틸리티를 이용하여 구동 중에만 한시적으로 일부 브로커 파라미터를 동적으로 변경할 수 있으며, 동적으로 변경할 수 있는 브로커 파라미터는 다음과 같다.

- ACCESS_LOG
- APPL_SERVER_MAX_SIZE

- KEEP_CONNECTION
- LOG_BACKUP
- SQL_LOG
- SQL_LOG_MAX_SIZE
- STATEMENT_POOLING
- TIME_TO_KILL

구문

브로커 구동 중에 브로커 파라미터를 변경하기 위한 **broker_changer** 유틸리티의 구문은 다음과 같다. *broker_name*에는 구동 중인 브로커 이름을 입력하면 되고 *parameter*는 동적으로 변경할 수 있는 브로커 파라미터에 한정된다. 변경하고자 하는 파라미터에 따라 *value*가 지정되어야 한다.

```
broker_changer broker_name parameters value
```

예제

구동 중인 브로커에서 SQL 로그가 기록되도록 SQL_LOG 파라미터를 ON으로 설정하기 위하여 다음과 같이 입력한다. 이와 같은 파라미터의 동적 변경은 브로커 구동 중일 때만 한시적으로 효력이 있다.

```
% broker_changer query editor sql log on
OK
```

브로커 로그

브로커 구동과 관련된 로그에는 접속 로그, 에러 로그, SQL 로그가 있다. 각각의 로그는 설치 디렉터리의 log 디렉터리에서 확인할 수 있으며, 저장 디렉터리의 변경은 브로커 환경 설정 파일(cubrid_broker.conf)의 **LOG_DIR** 파라미터와 **ERROR_LOG_DIR** 파라미터를 통해 설정할 수 있다.

접속 로그 확인

접속 로그 파일은 응용 클라이언트 접속에 관한 정보를 기록하며, *broker_name.access*의 이름으로 **log/broker/** 디렉터리에 저장된다. 또한, 브로커 환경 설정 파일에서 **LOG_BACKUP** 파라미터가 ON으로 설정된 경우, 브로커의 구동이 정상적으로 종료되면 접속 로그 파일에 종료된 날짜와 시간 정보가 추가되어 로그 파일이 저장된다. 예를 들어, broker1이 2008년 6월 17일 오후 12시 27분에 정상 종료되었다면, broker1.access.20080617.1227 이라는 접속 로그 파일이 생성된다. 다음은 접속 로그의 예제를 보여준다.

다음은 log 디렉터리에 생성된 접속 로그 파일의 예제와 설명이다.

```
1 192.168.1.203 - - 972523031.298 972523032.058 2008/06/17 12:27:46~2008/06/17 12:27:47
7118 - -1
2 192.168.1.203 - - 972523052.778 972523052.815 2008/06/17 12:27:47~2008/06/17 12:27:47
7119 ERR 1025
1 192.168.1.203 - - 972523052.778 972523052.815 2008/06/17 12:27:49~2008/06/17 12:27:49
7118 - -1
```

- 1 : 브로커의 응용서버에 부여된 ID
- 192.168.1.203 : 응용 클라이언트의 IP 주소

- 972523031.298 : 클라이언트의 요청 처리를 시작한 시각의 UNIX 타임스탬프 값
- 2008/06/17 12:27:46 : 클라이언트 요청을 처리 시작한 시각
- 972523032.058 : 클라이언트의 요청 처리를 완료한 시각의 UNIX 타임스탬프 값
- 2008/06/17 12:27:47 : 클라이언트의 요청을 처리 완료한 시각
- 7118 : 응용서버의 프로세스 ID
- -1 : 요청 처리 중 발생한 에러가 없음
- ERR 1025 : 요청 처리 중 발생한 에러가 있고, 에러 정보는 에러 로그 파일의 offset=1025에 존재함

에러 로그 확인

에러 로그 파일은 응용 클라이언트의 요청을 처리하는 도중에 발생한 에러에 관한 정보를 기록하며, *broker_name_app_server_num.err*의 이름으로 저장된다.

다음은 에러 로그의 예제와 설명이다.

```
Time: 02/04/09 13:45:17.687 - SYNTAX ERROR *** ERROR CODE = -493, Tran = 1, EID = 38
Syntax: Unknown class "unknown_tbl". select * from unknown_tbl
```

- Time: 02/04/09 13:45:17.687 : 에러 발생 시각
- - SYNTAX ERROR : 에러의 종류(SYNTAX ERROR, ERROR 등)
- *** ERROR CODE = -493 : 에러 코드
- Tran = 1 : 트랜잭션 ID. -1은 트랜잭션 ID를 할당 받지 못한 경우임.
- EID = 38 : 에러 ID. SQL 문 처리 중 에러가 발생한 경우, 서버나 클라이언트 에러 로그와 관련이 있는 SQL 로그를 찾을 때 사용함.
- Syntax:... : 에러 메시지

SQL 로그 관리

SQL 로그 파일은 응용 클라이언트가 요청하는 SQL을 기록하며, *broker_name_app_server_num.sql.log*라는 이름으로 저장된다. SQL 로그는 **SQL_LOG** 파라미터 값이 ON인 경우에 설치 디렉터리의 log/broker/sql_log 디렉터리에 생성된다. 이 때, 생성되는 SQL 로그 파일의 크기는 **SQL_LOG_MAX_SIZE** 파라미터의 설정값을 초과할 수 없으므로 주의한다. CUBRID는 SQL 로그를 관리하기 위한 유틸리티로서 **broker_log_top**, **broker_log_converter**, **broker_log_runner**를 제공하며, 이 유틸리티는 SQL 로그가 존재하는 디렉터리에서 실행해야 한다.

다음은 SQL 로그 파일의 예제와 설명이다.

```
02/04 13:45:17.687 (38) prepare 0 insert into unique_tbl values (1)
02/04 13:45:17.687 (38) prepare srv h id 1
02/04 13:45:17.687 (38) execute srv h id 1 insert into unique_tbl values (1)
02/04 13:45:17.687 (38) execute error:-670 tuple 0 time 0.000, EID = 39
02/04 13:45:17.687 (0) auto rollback
02/04 13:45:17.687 (0) auto_rollback 0
*** 0.000

02/04 13:45:17.687 (39) prepare 0 select * from unique_tbl
02/04 13:45:17.687 (39) prepare srv h id 1 (PC)
02/04 13:45:17.687 (39) execute srv h id 1 select * from unique_tbl
02/04 13:45:17.687 (39) execute 0 tuple 1 time 0.000
```



```
02/04 13:45:17.687 (0) auto commit
02/04 13:45:17.687 (0) auto commit 0
*** 0.000
```

- 02/04 13:45:17.687 : 응용 클라이언트의 요청 시각
- (39) : SQL 문 그룹의 시퀀스 번호이며, prepared statement pooling을 사용하는 경우, 파일 내에서 SQL 문마다 고유(unique)하게 부여된다.
- prepare 0 : prepared statement인지 여부
- prepare srv_h_id 1 : 해당 SQL 문을 srv_h_id 1로 prepare한다.
- (PC) : 플랜 캐시에 저장되어 있는 내용을 사용하는 경우에 출력된다.
- SELECT... : 실행 SQL 문. Statement pooling한 경우, WHERE 절의 binding 변수가 ?로 표시된다.
- Execute 0 tuple 1 time 0.000 : 1개의 row가 실행되고, 소요 시간은 0.000초
- auto_commit/auto_rollback : 자동으로 커밋되거나, 롤백되는 것을 의미한다. 두 번째 auto_commit/auto_rollback은 에러 코드이며, 0은 에러가 없이 트랜잭션이 완료되었음을 뜻한다.

broker_log_top 유틸리티는 특정 기간 동안 생성된 SQL 로그를 분석하여 실행 시간이 긴 순서대로 각 SQL 문과 실행 시간에 관한 정보를 파일에 출력하며, 분석된 결과는 각각 log.top.q 및 log.top.res에 저장된다.

broker_log_top 유틸리티는 롱 쿼리(long query)를 분석할 때 유용하며, 구문은 다음과 같다.

```
broker_log_top [options] sql log file list
options : {-t | -F from_date | -T to_date}
```

-t 옵션이 주어지면 트랜잭션 단위로 결과를 출력하고, **-F** 및 **-T** 옵션이 주어지면 날짜 범위를 지정하여 일부 SQL 문만 선별적으로 분석할 수 있다. 옵션을 모두 생략하면, 모든 로그에 대해 SQL 문 단위로 결과를 출력한다.

다음 예제는 11월 11일부터 11월 12일까지 생성된 SQL 로그에 대해 실행 시간이 긴 SQL문을 확인하기 위하여 **broker_log_top** 유틸리티를 실행한 화면이다. 기간을 지정할 때, 월과 일은 공백으로 구분한다. Windows에서는 "*.sql.log"를 인식하지 않으므로 SQL 로그 파일들을 공백(space)으로 구분해서 나열해야 한다.

```
--Linux에서 broker_log_top 실행
% broker_log_top -F "11 11" -T "11 12" -t *.sql.log

query_editor_1.sql.log
query_editor_2.sql.log
query_editor_3.sql.log
query_editor_4.sql.log
query_editor_5.sql.log

--Windows에서 broker_log_top 실행
% broker log top -F "11 11" -T "11 12" -t query_editor_1.sql.log query_editor_2.sql.log
query_editor_3.sql.log query_editor_4.sql.log query_editor_5.sql.log
```

위 예제를 실행하면 SQL 로그 분석 결과가 저장되는 **log.top.q** 및 **log.top.res** 파일이 동일한 디렉터리에 생성된다. **log.top.q**에서 각 SQL 문 및 SQL 로그 상의 라인 번호를 확인할 수 있고, **log.top.res**에서 각 SQL 문에 대한 최소 실행 시간, 최대 실행 시간, 평균 실행 시간, 쿼리 실행 수를 확인할 수 있다.

```
--log.top.q 파일의 내용
[Q1]-----
broker1_6.sql.log:137734
```

```

11/11 18:17:59.396 (27754) execute all srv h id 34 select a.int col, b.var col from
dml v view 6 a, dml v view 6 b, dml v view 6 c, dml v view 6 d, dml v view 6 e where
a.int col=b.int col and b.int col=c.int col and c.int col=d.int col and
d.int col=e.int col order by 1,2;
11/11 18:18:58.378 (27754) execute all 0 tuple 497664 time 58.982
.
.
[Q4]-----
broker1_100.sql.log:142068
11/11 18:12:38.387 (27268) execute all srv h id 798 drop table list test;
11/11 18:13:08.856 (27268) execute_all 0 tuple 0 time 30.469

--log.top.res 파일의 내용
max      min      avg      cnt(err)
-----
[Q1]      58.982    30.371    44.676    2 (0)
[Q2]      49.556    24.023    32.688    6 (0)
[Q3]      35.548    25.650    30.599    2 (0)
[Q4]      30.469     0.001     0.103 1050 (0)

```

설치 디렉터리의 log/broker/sql_log 디렉터리에 생성된 SQL 로그 파일에 기록된 질의를 별도의 입력 파일로 저장하기 위하여 **broker_log_converter** 유틸리티를 실행한다. **broker_log_converter** 유틸리티의 구문은 다음과 같다.

```
broker_log_converter SQL_log_file output_file
```

다음 예제는 query_editor_1.sql.log 파일에 저장된 질의를 query_convert.in 파일로 변경한다.

```
% broker_log_converter query_editor_1.sql.log query_convert.in
```

broker_log_converter 유틸리티에 의해 생성된 질의 파일에 저장된 질의를 재실행하기 위하여

broker_log_runner 유틸리티를 실행한다. **broker_log_runner** 유틸리티의 구문은 다음과 같으며, 예제는

query_convert.in 파일에 저장된 질의를 demodb에서 재실행하며, 브로커 IP가 192.168.1.10이고,

브로커 포트는 30000인 환경임을 가정한다.

```

broker_log_runner options input_file
options : -I cas_ip -P cas_port -d dbname [-u dbuser [-p dbpasswd ]] [-t num_thread] [-r repeat_count] [-o result_file]

```

broker_log_runner 유틸리티의 옵션

옵션	설명
-I <i>broker_ip</i>	CUBRID 브로커의 IP 주소 또는 호스트 이름
-P <i>broker_port</i>	CUBRID 브로커의 포트 번호
-d <i>dbname</i>	질의를 실행할 데이터베이스 이름
-u <i>dbuser</i>	데이터베이스 사용자 이름 (디폴트 값: public)
-p <i>dbpasswd</i>	데이터베이스 암호
-t <i>numthread</i>	스레드의 개수(디폴트 값: 1)
-r <i>repeat_count</i>	질의가 수행될 횟수(디폴트 값: 1)
-o <i>result_file</i>	수행 결과를 저장할 파일 이름

```

% broker_log_runner -I 192.168.1.10 -P 30000 -d demodb -t 2 query_convert.in
cas_ip = 192.168.1.10
cas_port = 30000
num_thread = 2

```

```
repeat = 1
dbname = demodb
dbuser = public
dbpasswd =
exec time : 0.001
exec time : 0.000
0.000500 0.000500 -
```

CUBRID 매니저 서버

CUBRID 매니저 구동 및 종료

CUBRID 매니저 구동

CUBRID 매니저 서버를 구동하기 위하여 다음과 같이 입력한다.

```
% cubrid manager start
```

이미 CUBRID 매니저 서버가 구동 중에 있다면 다음과 같은 메시지가 출력된다.

```
% cubrid manager start
@ cubrid manager server start
++ cubrid manager server is running.
```

CUBRID 매니저 종료

CUBRID 매니저 서버를 종료하기 위하여 다음과 같이 입력한다.

```
% cubrid manager stop
@ cubrid manager server stop
++ cubrid manager server stop: success
```

CUBRID 매니저 서버 로그

CUBRID 매니저 서버와 관련된 로그는 설치 디렉토리의 log/manager 디렉토리에 저장되며, 매니저 서버 프로세스에 따라 다음과 같이 네 종류의 로그 파일로 저장된다.

- cub_auto.access.log : 서버에 로그인, 로그 아웃을 정상적으로 수행한 클라이언트의 접속 로그
- cub_auto.error.log : 서버에 로그인, 로그 아웃을 실패한 클라이언트의 접속 로그
- cub_js.access.log : 매니저 서버에 의해 처리된 작업에 관한 로그
- cub_js.error.log : 매니저 서버에 의해 작업 처리 도중 발생한 에러에 관한 로그

데이터베이스 관리

CUBRID 관리 유틸리티 사용법(구문)

CUBRID 관리 유틸리티의 사용법(구문)은 다음과 같다.

```
cubrid utility name
utility_name :
  createdb [option] <database_name>    --- 데이터베이스 생성
  deletedb [option] <database_name>    --- 데이터베이스 삭제
  installdb [option] <database-name>    --- 데이터베이스 설치
  renamedb [option] <source-database-name> <target-database-name> --- 데이터베이스 이름 변경
  copydb [option] <source-database-name> <target-database-name> --- 데이터베이스 복사
  backupdb [option] <database-name>    --- 데이터베이스 백업
  restoredb [option] <database-name>    --- 데이터베이스 복구
  addvoldb [option] <database-name> number-of-pages --- 데이터베이스 볼륨 파일 추가
  spacedb [option] <database-name>    --- 데이터베이스 공간 정보 출력
  lockdb [option] <database-name>    --- 데이터베이스의 lock 정보 출력
  killtran [option] <database-name>    --- 트랜잭션 제거
  optimizedb [option] <database-name> --- 데이터베이스 통계 정보 갱신
  statdump [option] <database-name>    --- 데이터베이스 서버 실행 통계 정보 출력
  compactdb [option] <database-name>    --- 사용되지 않는 영역을 해제, 공간 최적화
  diagdb [option] <database-name>    --- 내부 정보 출력
  emergency_patchlog [option] <database-name> --- 긴급한 상황에서 데이터베이스 로그 패치
  checkdb [option] <database-name>    --- 데이터베이스 일관성 검사
  alterdbhost [option] <database-name> --- 데이터베이스 호스트 변경
  plandump [option] <database-name>    --- 쿼리 플랜 캐시 정보 출력
  loaddb [option] <database-name>    --- 데이터 및 스키마 가져오기(로드)
  unloaddb [option] <database-name>    --- 데이터 및 스키마 내보내기(언로드)
  paramdump [option] <database-name>    --- 데이터베이스의 설정된 파라미터 값 확인
  changemode [option] <database-name> --- 서버의 HA 모드 출력 또는 변경
  copylogdb [option] <database-name> --- HA구성을 위해 트랜잭션 로그를 다중화하는 도구
  applylogdb [option] <database-name> --- HA 구성을 위해 트랜잭션 로그에서 복제 로그를 읽고
  적용하는 도구
```

데이터베이스 사용자

CUBRID 데이터베이스 사용자는 동일한 권한을 갖는 멤버를 가질 수 있다. 사용자에게 권한 **A**가 부여되면, 상기 사용자에게 속하는 모든 멤버에게도 권한 **A**가 동일하게 부여된다. 이와 같이 데이터베이스 사용자와 그에 속한 멤버를 '그룹'이라 하고, 멤버가 없는 사용자를 '사용자'라 한다.

CUBRID는 **DBA**와 **PUBLIC**이라는 사용자를 기본으로 제공한다.

- **DBA**는 모든 사용자의 멤버가 되며 데이터베이스의 모든 객체에 접근할 수 있는 최고 권한 사용자이다. 또한, **DBA**만이 데이터베이스 사용자를 추가, 편집, 삭제할 수 있는 권한을 갖는다.

- DBA를 포함한 모든 사용자는 **PUBLIC**의 멤버가 되므로 모든 데이터베이스 사용자는 **PUBLIC**에 부여된 권한을 가진다. 예를 들어, **PUBLIC** 사용자에게 권한 **B**를 추가하면 데이터베이스의 모든 사용자에게 일괄적으로 권한 **B**를 부여된다.

databases.txt 파일

설명

CUBRID는 존재하는 모든 데이터베이스의 위치에 관한 정보는 **databases.txt** 파일에 저장하는데, 이를 데이터베이스 위치 정보 파일이라 한다. 이러한 데이터베이스 위치 정보 파일은 데이터베이스의 생성, 이름 변경, 삭제 및 복사에 관한 유틸리티를 수행하거나 각 데이터베이스를 구동할 때 사용되며, 기본적으로는 설치 디렉터리의 **databases** 디렉터리에 위치하고, **CUBRID_DATABASES** 환경 변수로 디렉터리 위치를 지정할 수 있다.

구문

```
db_name db_directory server_host logfile_directory
```

데이터베이스 위치 정보 파일의 라인별 포맷은 구문에 정의된 바와 같으며, 데이터베이스 이름, 데이터베이스 경로, 서버 호스트 및 로그 파일의 경로에 관한 정보를 저장한다. 다음은 데이터베이스 위치 정보 파일의 내용을 확인한 예이다.

```
% more databases.txt
dist testdb /home1/user/CUBRID/bin d85007 /home1/user/CUBRID/bin
dist demodb /home1/user/CUBRID/bin d85007 /home1/user/CUBRID/bin
testdb /home1/user/CUBRID/databases/testdb d85007 /home1/user/CUBRID/databases/testdb
demodb /home1/user/CUBRID/databases/demodb d85007 /home1/user/CUBRID/databases/demodb
```

데이터베이스 위치 정보 파일의 저장 디렉터리는 기본적으로 설치 디렉터리의 **databases** 디렉터리로 지정되며, 시스템 환경 변수 **CUBRID_DATABASES**의 설정을 변경하여 기본 디렉터리를 변경할 수 있다. 데이터베이스 위치 정보 파일의 저장 디렉터리 경로가 유효해야 데이터베이스 관리를 위한 **cubrid** 유틸리티가 데이터베이스 위치 정보 파일에 접근할 수 있게 된다. 이를 위해서 사용자는 디렉터리 경로를 정확하게 입력해야 하고, 해당 디렉터리 경로에 대해 쓰기 권한을 가지는지 확인해야 한다. 다음은 **CUBRID_DATABASES** 파라미터에 설정된 값을 확인하는 예이다.

```
% set | grep CUBRID_DATABASES
CUBRID_DATABASES=/home1/user/CUBRID/databases
```

만약 **CUBRID_DATABASES** 환경 변수에서 유효하지 않은 디렉터리 경로가 설정되는 경우에는 에러가 발생하며, 설정된 디렉터리 경로는 유효하나 데이터베이스 위치 정보 파일이 존재하지 않는 경우에는 새로운 위치 정보 파일을 생성한다. 또한, **CUBRID_DATABASES** 환경 변수가 아예 설정되지 않은 경우에는 현재 작업 디렉터리에서 위치 정보 파일을 검색한다.

데이터베이스 생성

설명

cubrid createdb 유틸리티는 CUBRID 시스템에서 사용할 데이터베이스를 생성하고 미리 만들어진 CUBRID 시스템 테이블을 초기화한다. 데이터베이스에 권한이 주어진 초기 사용자를 정의할 수도 있다. 일반적으로 데이터베이스 관리자만이 **cubrid createdb** 유틸리티를 사용한다. 로그와 데이터베이스의 위치도 지정할 수 있다.

구문

```
cubrid createdb options database_name
options :
[{-p | --pages=}number]
[--comment]
[{-F | --file-path=}path]
[{-L | --log-path=}path]
[--server-name=host]
[{-r | --replace}]
[--more-volume-file=file]
[--user-definition-file=file]
[--csql-initialization-file=file]
[{-o | --output-file=} file]
[{-v | --verbose}]
[{-l | --log-page-count=} number]
[--log-page-size=size]
[{-s | --page-size=} size]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **createdb** : 새로운 데이터베이스를 생성하기 위한 명령이다.
- *options* : 단축 옵션은 -와 함께 지정해야 하고, 전체 옵션은 --와 함께 지정해야 한다.
- *database_name* : 데이터베이스가 생성될 디렉터리 경로명을 포함하지 않고, 생성하고자 하는 데이터베이스의 이름을 고유하게 부여한다. 이 때, 지정한 데이터베이스 이름이 이미 존재하는 데이터베이스 이름과 중복되는 경우, CUBRID는 기존 파일을 보호하기 위하여 데이터베이스 생성을 더 이상 진행하지 않는다.

옵션

다음은 **cubrid createdb** 유틸리티와 함께 사용할 수 있는 옵션을 정리한 표이다. 대소문자를 구별해서 사용해야 한다.

옵션	설명
-p --pages	생성될 데이터베이스 볼륨(generic)의 페이지 수를 지정한다. 디폴트 값 : 5000 페이지
-F --file-path	데이터베이스가 생성될 디렉터리 경로를 지정한다. 디폴트 값 : 현재 작업 디렉터리
-L --log-path	로그 파일이 저장되는 디렉터리 경로를 지정한다. 디폴트 값 : -F 옵션으로 지정된 디렉터리 경로
-r	생성될 데이터베이스의 이름이 기존 데이터베이스 이름과 중복

--replace	되는 경우, 덮어쓰기를 허용한다. 디폴트 값 : 비활성화
-o --output-file	데이터베이스 생성에 관한 출력 메시지가 저장되는 파일을 지정한다.
-v --verbose	데이터베이스 생성에 관한 상세 메시지가 화면 출력된다. 디폴트 값 : 비활성화
-l --log-page-count	로그 볼륨의 페이지 수를 지정한다. 디폴트 값 : -p 옵션으로 지정된 범용 볼륨의 페이지 수
--log-page-size	바이트 단위로 로그 볼륨의 페이지 크기를 지정한다. 디폴트 값 : 데이터베이스 페이지의 크기
-s --page-size	바이트 단위로 데이터베이스 페이지 크기를 지정한다. 디폴트 값 : 4096(4kbyte)
--comment	생성될 데이터베이스에 관한 정보를 주석으로 기록한다.
--server-name	접속할 서버 호스트의 이름을 지정한다. 디폴트 값 : 로컬 호스트
--more-volume-file	데이터베이스의 추가 볼륨을 생성하기 위한 명세를 포함하는 파일을 지정한다.
--user-definition-file	사용자 정의를 포함하는 파일을 지정한다.
--csql-initialization-file	csql 초기화를 위한 파일을 지정한다.

페이지 수(-p)

10,000 페이지가 할당된 testdb라는 이름의 데이터베이스를 생성하는 구문이다.

-p는 데이터베이스의 응용 프로그램을 위한 데이터베이스 페이지 수를 지정하기 위한 옵션으로서, 디폴트 값은 5,000이다.

```
cubrid createdb -p 10000 testdb
```

데이터베이스 디렉터리 경로(-F)

testdb라는 이름의 데이터베이스를 /dbtemp/new_db라는 디렉터리에 생성하는 구문이다.

-F는 새로운 데이터베이스가 생성되는 디렉터리의 절대 경로를 지정하는 옵션으로, **-F** 옵션을 지정하지 않는 경우에는 현재 작업 디렉터리에 새로운 데이터베이스가 생성된다.

```
cubrid createdb -F /dbtemp/new_db/ testdb
```

로그 파일 디렉터리 경로(-L)

testdb라는 이름의 데이터베이스를 /dbtemp/newdb라는 디렉터리에 생성하고, 로그 파일을 /dbtemp/db_log 디렉터리에 생성하는 구문이다.

-L은 데이터베이스의 로그 파일이 생성되는 디렉터리의 절대 경로를 지정하는 옵션으로, **-L** 옵션을 지정하지 않는 경우에는 **-F** 옵션에서 지정한 디렉터리에 생성된다. 만약, **-F** 옵션과 **-L** 옵션을 둘 다 지정하지 않는 경우에는 데이터베이스와 로그 파일이 현재 작업 디렉터리에 생성된다.

```
cubrid createdb -F /dbtemp/new_db/ -L /dbtemp/db_log/ testdb
```

덮어쓰기(-r)

testdb라는 이름의 데이터베이스가 이미 존재하더라도 기존의 testdb를 덮어쓰기하고 새로운 testdb를 생성하라는 구문이다.

-r은 지정된 데이터베이스 이름이 이미 존재하는 데이터베이스 이름과 중복되더라도 새로운 데이터베이스를 생성하고, 기존의 데이터베이스를 덮어쓰도록 하는 옵션이다. 만약, **-r** 옵션을 지정하지 않으면 더 이상 데이터베이스 생성을 진행하지 않는다.

```
cubrid createdb -r testdb
```

출력 메시지를 파일에 저장(-o)

testdb라는 데이터베이스를 생성하면서 이에 관한 유틸리티의 출력을 콘솔 화면이 아닌 db_output 파일에 저장하는 구문이다.

-o 옵션은 데이터베이스 생성에 관한 메시지를 파라미터로 지정된 파일에 저장하는 옵션이며, 파일은 데이터베이스와 동일한 디렉터리에 생성된다. **-o** 옵션이 지정되지 않으면 메시지는 콘솔 화면에 출력된다. **-o** 옵션은 데이터베이스가 생성되는 중에 출력되는 메시지를 지정된 파일에 저장함으로써 특정 데이터베이스의 생성 과정에 관한 정보를 활용할 수 있게 한다.

```
cubrid createdb -o db_output testdb
```

버보스(Verbose) 출력(-v)

testdb라는 데이터베이스를 생성하면서 이에 관한 상세한 연산 정보를 화면에 출력하는 구문이다.

-v 옵션은 데이터베이스 생성 연산에 관한 모든 정보를 화면에 출력하는 옵션으로서, **-o** 옵션과 마찬가지로 특정 데이터베이스 생성 과정에 관한 정보를 확인하는데 유용하다. 따라서, **-v** 옵션과 **-o** 옵션을 함께 지정하면, **-o** 옵션의 파라미터로 지정된 출력 파일에 **cubrid createdb** 유틸리티의 연산 정보와 생성 과정에 관한 출력 메시지를 저장할 수 있다.

```
cubrid createdb -v testdb
```

로그 페이지(-l)

testdb라는 이름의 데이터베이스를 생성하고, testdb의 로그 볼륨 페이지 수를 1,000으로 지정하는 구문이다.

-l 옵션은 생성되는 데이터베이스의 로그 볼륨 페이지 수를 지정하는 옵션으로서, 디폴트 값은 **-p** 옵션에 의해 지정된 범용(**generic**) 볼륨의 페이지 수와 같다. 데이터 변경 처리량과 트랜잭션 지속 시간에 따라 요구되는 로그 볼륨의 페이지 수가 달라지지만, 최소한 데이터베이스 볼륨의 페이지 수와 동일하거나 그 보다 2배 정도 큰 것이 적당하다.


```
cubrid createdb -l 1000 testdb
```

로그 페이지 크기(--log-page-size)

testdb를 생성하고, testdb의 로그 볼륨 페이지 크기를 8kbyte로 지정하는 구문이다.

--log-page-size 옵션은 생성되는 데이터베이스의 로그 볼륨 페이지 크기를 지정하는 옵션으로서, 디폴트 값은 데이터 페이지 크기와 같다. 로그 페이지 크기는 트랜잭션 간격, 로그 용량 및 운영체제의 I/O 페이지 크기를 고려하여 설정되어야 한다. 트랜잭션 커밋 주기가 짧아 디스크로 내려쓰기(flush)되는 로그 용량이 적은 환경에서 이 옵션 값을 너무 크게 설정하면 불필요한 디스크 I/O비용이 증가할 수 있다.

```
cubrid createdb -log-page-size 8192 testdb
```

데이터 페이지 크기(-s)

testdb의 범용(generic) 볼륨의 데이터 페이지 크기를 8192바이트로 지정하는 구문이다.

-s 옵션은 데이터베이스 페이지의 크기를 1024, 2048, 4096, 8192, 16384바이트 중 하나의 값으로 지정하는 옵션이며, 디폴트 값은 4096이다. 만약, 이러한 값이 아닌 다른 숫자를 지정할 경우에는 시스템이 올림값으로 페이지 크기를 지정한다. **-s** 옵션을 통해 데이터베이스 볼륨의 페이지 크기를 변경하면, 이와 연관된 데이터베이스 파라미터 **data_buffer_pages**, **sort_buffer_pages**, **log_buffer_pages**에도 영향을 미치므로 주의해야 한다.

```
cubrid createdb -s 8192 testdb
```

주석(--comment)

testdb라는 이름의 데이터베이스를 생성하면서, 데이터베이스 볼륨에 이에 관한 주석을 추가하는 구문이다.

--comment 옵션은 데이터베이스의 볼륨 헤더에 지정된 주석을 포함하는 옵션으로 문자열에 공백이 포함되는 경우에는 큰 따옴표로 감싸주어야 한다.

```
cubrid createdb --comment "a new database for study" testdb
```

서버 호스트 이름(--server-name)

testdb라는 이름의 데이터베이스를 *aa_host* 호스트 상에 생성 및 등록하는 구문이다.

--server-name 옵션은 CUBRID의 클라이언트/서버 버전을 사용할 때 특정 데이터베이스에 대한 서버가 지정한 호스트 상에 구동되도록 하는 옵션이다. 이 옵션으로 지정된 서버 호스트의 정보는 데이터베이스 위치 정보 파일(**databases.txt**)에 기록된다. 만약, 이 옵션이 지정되지 않으면 디폴트 값은 현재 로컬 호스트이다.

```
cubrid createdb --server-name aa_host testdb
```

데이터베이스 볼륨 추가(--more-volume-file)

testdb를 생성함과 동시에 *vol_info.txt*에 저장된 명세를 기반으로 볼륨을 추가 생성하는 구문이다.

--more-volume-file 옵션은 데이터베이스가 생성되는 디렉터리에 추가 볼륨을 생성하는 옵션으로 지정된 파일에 저장된 명세에 따라 추가 볼륨을 생성한다. 이 옵션을 이용하지 않더라도, **cubrid addvoldb** 유틸리티를 이용하여 볼륨을 추가할 수 있다.

```
cubrid createdb --more-volume-file vol_info.txt testdb
```

다음은 *vol_info.txt*에 저장된 추가 볼륨에 관한 명세이다. 각 볼륨에 관한 명세는 라인 단위로 작성되어야 한다.

```
#####
# NAME volname COMMENTS volcmnts PURPOSE volpurp NPAGES volnpgs
NAME data_v1 COMMENTS "데이터 정보 볼륨" PURPOSE data NPAGES 1000
NAME data_v2 COMMENTS "데이터 정보 볼륨" PURPOSE data NPAGES 1000
NAME data_v3 PURPOSE data NPAGES 1000
NAME index_v1 COMMENTS "인덱스 정보 볼륨" PURPOSE index NPAGES 500
NAME temp_v1 COMMENTS "임시 정보 볼륨" PURPOSE temp NPAGES 500
NAME generic_v1 COMMENTS "일반 정보 볼륨" PURPOSE generic NPAGES 500
#####
```

예제 파일에서와 같이 각 볼륨에 관한 명세는 다음과 같이 구성된다.

```
NAME volname COMMENTS volcmnts PURPOSE volpurp NPAGES volnpgs
```

- **volname**: 추가 생성될 볼륨의 이름으로 Unix 파일 이름 규약을 따라야 하고, 디렉터리 경로를 포함하지 않는 단순한 이름이어야 한다. 볼륨명에 관한 명세는 생략할 수 있으며, 이 경우 시스템에 의해 "생성될 데이터베이스 이름_볼륨 식별자"로 볼륨명이 생성된다.
- **volcmnts**: 볼륨 헤더에 기록되는 주석 문장으로, 추가 생성되는 볼륨에 관한 정보를 임의로 부여할 수 있다. 볼륨 주석에 관한 명세 역시 생략할 수 있다.
- **volpurp**: 볼륨 저장의 목적으로, **data**, **index**, **temp**, **generic** 중 하나여야 한다. 볼륨 목적에 관한 명세는 생략할 수 있으며, 이 경우 디폴트 값은 **generic**이다.
- **volnpgs**: 추가 생성되는 볼륨의 페이지 수이다. 볼륨 페이지 수에 관한 명세는 생략할 수 없으며, 반드시 지정해야 한다.

CSQL 구문 저장 파일(--csql-initialization-file)

testdb를 생성함과 동시에 table_schema.sql에 정의된 SQL 구문을 CSQL 인터프리터에서 실행시키는 구문이다.

--csql-initialization-file 옵션은 생성하고자 하는 데이터베이스에 대해 CSQL 인터프리터에서 구문을 실행하는 옵션으로, 파라미터로 지정된 파일에 저장된 SQL 구문에 따라 스키마를 생성할 수 있다.

```
cubrid createdb --csql-initialization-file table_schema.sql testdb
```

사용자 정보 파일(--user-definition-file)

testdb를 생성함과 동시에 user_info.txt에 정의된 사용자 정보를 기반으로 testdb에 대한 사용자를 추가하는 구문이다.

--user-definition-file 옵션은 생성하고자 하는 데이터베이스에 대해 권한이 있는 사용자를 추가하는 옵션으로, 파라미터로 지정된 사용자 정보 파일에 저장된 명세에 따라 사용자를 추가한다. --user-definition-file 옵션을 이용하지 않더라도 **CREATE USER** 구문을 이용하여 사용자를 추가할 수 있다.

```
cubrid createdb --user-definition-file user_info.txt testdb
```

사용자 정보 파일의 구문은 아래와 같다.

```
USER user_name [ groups_clause | members_clause ]

groups_clause:
[ GROUPS group_name [ { group_name }... ] ]

members_clause:
[ MEMBERS member_name [ { member_name }... ] ]
```

- *user_name*은 데이터베이스에 대해 권한을 가지는 사용자 이름이며, 공백이 포함되지 않아야 한다.
- **GROUPS** 절은 옵션이며, *group_name*은 지정된 *user_name*을 포함하는 상위 그룹의 이름이다. 이 때, *group_name*은 하나 이상이 지정될 수 있으며, **USER**로 미리 정의되어야 한다.
- **MEMBERS** 절은 옵션이며, *member_name*은 지정된 *user_name*에 포함되는 하위 멤버의 이름이다. 이 때, *member_name*은 하나 이상이 지정될 수 있으며, **USER**로 미리 정의되어야 한다.

사용자 정보 파일에서는 주석을 사용할 수 있으며, 주석 라인은 연속된 하이픈(--)으로 시작된다. 공백 라인은 무시된다.

다음 예제는 그룹 sedan에 granduer와 sonata가, 그룹 suv에 tuscan이, 그룹 hatchback에 i30가 포함되는 것을 정의하는 사용자 정보 파일이다. 사용자 정보 파일명은 user_info.txt로 예시한다.

```
--
-- 사용자 정보 파일의 예1
--
USER sedan
USER suv
USER hatchback
USER granduer GROUPS sedan
USER sonata GROUPS sedan
USER tuscan GROUPS suv
USER i30 GROUPS hatchback
```

위 예제와 동일한 사용자 관계를 정의하는 파일이다. 다만, 아래 예제에서는 **MEMBERS** 절을 이용하였다.

```
--
-- 사용자 정보 파일의 예 2
--
USER granduer
USER sonata
USER tuscan
USER i30
USER sedan MEMBERS sonata granduer
USER suv MEMBERS tuscan
USER hatchback MEMBERS i30
```

데이터베이스 볼륨 추가

설명

CUBRID 매니저로 데이터베이스 볼륨을 추가하는 방법은 [저장 공간](#)을 참조한다.

구문

```
cubrid addvoldb options [args] database_name number_of_pages
options : [{-n | --volume_name= } name] [{-F | --file-path=} path] [ {-p | --
page=}number]
[-S | -C | --SA-mode | --CS-mode] [--comment]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **addvoldb** : 지정된 데이터베이스에 지정된 페이지 수만큼 새로운 볼륨을 추가하기 위한 명령이다.
- **options** : 단축 옵션은 -와 함께 지정해야 하고, 전체 옵션은 --와 함께 지정해야 한다.
- **database_name** : 데이터베이스가 생성될 디렉터리 경로명을 포함하지 않고, 볼륨을 추가하고자 하는 데이터베이스의 이름을 지정한다.
- **number_of_pages** : 지정된 데이터베이스 볼륨으로 추가할 페이지 수이다. 추가할 페이지 수는 용도에 따라 여유있게 설정하고, 용도별로 디스크를 분리하여 저장하는 것이 성능 측면에서 유리하다.

옵션

다음은 **cubrid addvoldb** 유틸리티와 함께 사용할 수 있는 옵션을 정리한 표이다.

옵션	설명
-n --volume-name	추가되는 데이터베이스 볼륨의 이름을 지정한다. 디폴트 값 : 시스템에 의해 <i>dbname_number</i> 의 형식으로 부여
-F --file-path	추가되는 데이터베이스 볼륨이 생성될 디렉터리 경로를 지정한다. 디폴트 값 : 데이터베이스 파라미터인 volume_extension_path 의 설정 값
-p --purpose	추가되는 데이터베이스 볼륨의 용도를 지정한다. 디폴트 값 : 범용(generic) 볼륨
-S --SA-mode	독립 모드에서 데이터베이스 볼륨 추가 작업을 실행한다.
-C --CS-mode	클라이언트/서버 모드(client/server)에서 데이터베이스 볼륨 추가 작업을 실행한다.
--comment	추가되는 데이터베이스 볼륨에 관한 주석을 입력한다.

확장 볼륨 이름(-n)

독립모드(standalone) 상태에서 *testdb*라는 데이터베이스에 1,000페이지가 할당된 볼륨을 추가하는 구문이며, 생성되는 볼륨명은 *testdb_v101*이 된다.

-n은 지정된 데이터베이스에 대하여 추가될 볼륨의 이름을 지정하는 옵션이다. 볼륨명은 운영체제의 파일 이름 규약을 따라야 하고, 디렉터리 경로나 공백을 포함하지 않는 단순한 이름이어야 한다. **-n** 옵션이 생략되는 경우, 추가되는 볼륨의 이름은 시스템에 의해 "데이터베이스 이름_볼륨 식별자"로 볼륨명이 자동 부여된다. 예를 들어, 데이터베이스 명이 *testdb*인 경우, 자동 부여된 볼륨명은 *testdb_x00101*이 된다.

```
cubrid addvoldb -S -n testdb_v1 testdb 1000
```

볼륨 확장 경로(-F)

독립모드(standalone) 상태에서 *testdb*라는 데이터베이스에 1,000페이지가 할당된 볼륨을 추가하는 구문이며, 추가 볼륨은 */dbtemp/addvol/* 디렉터리에 생성된다. 볼륨명에 관한 **-n** 옵션을 지정하지 않았으므로, 생성되는 볼륨명은 *testdb_x0010*이 된다.

-F 옵션은 지정된 데이터베이스에 대하여 추가될 볼륨이 저장되는 디렉터리 경로를 지정하는 옵션이다. **-F** 옵션이 생략되면, 데이터베이스 파라미터인 **volume_extension_path**의 값이 디폴트 값으로 사용된다.

```
cubrid addvoldb -S -F /dbtemp/addvol/ testdb 1000
```

볼륨 용도(-p)

독립모드(standalone) 상태에서 *testdb*라는 데이터베이스에 1,000페이지가 할당된 인덱스 볼륨을 추가하는 구문이다.

-p 옵션은 추가할 볼륨의 사용 목적에 따라 볼륨의 종류를 지정하는 옵션이다. 이처럼 볼륨의 사용 목적에 맞는 볼륨을 지정해야 볼륨 종류별로 디스크 드라이브에 분리 저장할 수 있어 I/O 성능을 높일 수 있다. **-p** 옵션의 파라미터로 가능한 값은 **data**, **index**, **temp**, **generic** 중 하나이며, 디폴트 값은 **generic** 이다. 각 볼륨 용도에 관해서는 [데이터베이스 볼륨 구조](#)를 참조한다.

```
cubrid addvoldb -S -p index testdb 1000
```

독립 모드(-S)

-S 옵션은 서버 프로세스를 구동하지 않고 데이터베이스에 접근하는 독립 모드(standalone)로 작업하기 위해 지정되며, 인수는 없다. **-S** 옵션을 지정하지 않으면, 시스템은 클라이언트/서버 모드로 인식한다.

```
cubrid addvoldb -S testdb 1000
```

클라이언트/서버 모드(-C)

-C 옵션은 서버 프로세스와 클라이언트 프로세스를 각각 구동하여 데이터베이스에 접근하는 클라이언트/서버 모드로 작업하기 위한 옵션이며, 인수는 없다. **-C** 옵션을 지정하지 않더라도 시스템은 기본적으로 클라이언트/서버 모드로 인식한다.

-S 또는 **-C** 옵션이 지정되지 않았고 환경 변수 **CUBRID_MODE**도 정의되지 않았다면, 시스템은 클라이언트/서버 모드로 가정한다.

```
cubrid addvoldb -C -testdb 1000
```

추가 볼륨에 관한 주석(--comment)

독립모드(standalone) 상태에서 *testdb*라는 데이터베이스에 1,000페이지가 할당된 볼륨을 추가하는 구문이며, 해당 볼륨에 관한 정보를 주석으로 남긴다.

--comment 옵션은 추가된 볼륨에 관한 정보 검색을 쉽게 하기 위하여 볼륨에 관한 정보를 주석으로 처리하는 옵션이다. 이 때, 주석의 내용은 볼륨을 추가하는 **DBA**의 이름이나 볼륨 추가의 목적을 포함하는 것이 바람직하며, 큰 따옴표로 감싸야 한다.

```
cubrid addvoldb -S --comment "데이터 볼륨 추가_김철수" testdb 1000
```

데이터베이스 삭제

설명

cubrid deletedb는 데이터베이스를 삭제하는 유틸리티이다. 데이터베이스가 몇 개의 상호 의존적 파일들로 만들어지기 때문에, 데이터베이스를 제거하기 위해 운영체제 파일 삭제 명령이 아닌 **cubrid deletedb** 유틸리티를 사용해야 한다. **cubrid deletedb** 유틸리티는 데이터베이스 위치 파일(**databases.txt**)에 지정된 데이터베이스에 대한 정보도 같이 삭제한다. **cubrid deletedb** 유틸리티는 오프라인 상에서 즉, 아무도 데이터베이스를 사용하지 않는 상태에서 독자 실행 모드로 사용해야 한다.

구문

```
cubrid deletedb options database_name
options : [{-o|--output-file=} file] [-d|--delete-backup]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **deletedb** : 데이터베이스 및 관련 데이터, 로그, 백업 파일을 전부 삭제하기 위한 명령으로, 데이터베이스 서버가 구동 정지 상태인 경우에만 정상적으로 수행된다.
- *options* : **-o** 옵션과 **-d** 옵션이 제공된다.
- *database_name* : 디렉터리 경로명을 포함하지 않고, 삭제하고자 하는 데이터베이스의 이름을 지정한다

옵션

출력 메시지 저장(-o 또는 --output-file)

-o 옵션을 이용하여 testdb를 삭제하면서 출력 메시지를 인수로 지정한 파일에 기록하는 명령이다. **cubrid deletedb** 유틸리티를 사용하면 데이터베이스 위치 정보 파일(**databases.txt**)에 기록된 데이터베이스 정보가 함께 삭제된다.

```
cubrid deletedb -o deleted_db.out testdb
```

만약, 존재하지 않는 데이터베이스를 삭제하는 명령을 입력하면 다음과 같은 메시지가 출력된다.

```
cubrid deletedb testdb
Database "testdb" is unknown, or the file "databases.txt" cannot be accessed.
```

백업 파일도 함께 삭제(-d 또는 --delete-backup)

-d 옵션을 이용하면 testdb를 삭제하면서 testdb의 백업 볼륨 및 백업 정보 파일도 함께 삭제할 수 있다. 만약, **-d** 옵션을 지정하지 않으면 백업 볼륨 및 백업 정보 파일은 삭제되지 않는다.

```
cubrid deletedb -d testdb
```

데이터베이스 변경

설명

cubrid renamedb 유틸리티는 존재하는 데이터베이스의 현재 이름을 변경한다. 정보 볼륨, 로그 볼륨, 제어 파일들이 새로운 이름과 일치되게 이름을 변경한다.

cubrid alterdbhost 유틸리티는 지정된 데이터베이스의 호스트 이름을 설정하거나 변경한다.

databases.txt에 있는 호스트 이름을 변경한다.

구문

```
cubrid renamedb options src database name dest database name
options : [{-E | --extended-volume-path=}path ] [ {-i | --control-file=} file ] [-d | --delete-backup]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **renamedb** : 현재 존재하는 데이터베이스의 이름을 새로운 이름으로 변경하기 위한 명령으로, 데이터베이스가 구동 정지 상태인 경우에만 정상적으로 수행된다. 관련된 정보 볼륨, 로그 볼륨, 제어 파일도 함께 새로 지정된 이름으로 변경된다.
- **options** : **-E**, **-i**, **-d** 옵션이 제공되며, 각 옵션에 관한 설명은 옵션 설명 및 예제를 참조한다.
- **src_database_name** : 이름을 바꾸고자 하는 현재 존재하는 데이터베이스의 이름이며, 데이터베이스가 생성될 디렉터리 경로명을 포함하지 않는다.
- **dest_database_name** : 새로 부여하고자 하는 데이터베이스의 이름이며, 현재 존재하는 데이터베이스 이름과 중복되어서는 안 된다. 이 역시, 데이터베이스가 생성될 디렉터리 경로명을 포함하지 않는다.

옵션

변경된 이름의 확장 볼륨을 새 디렉터리에 저장(-E 또는 --extended-volume-path)

확장 볼륨의 이름을 변경한 후 새 디렉터리 경로로 이동하는 명령으로서, **-E** 옵션을 이용하여 변경된 이름을 가지는 확장 볼륨을 이동시킬 새로운 디렉터리 경로(예: /dbtemp/newaddvols/)를 지정한다. **-E** 옵션을 주지 않으면, 확장 볼륨은 기존 위치에서 이름만 변경된다. 이때, 기존 데이터베이스 볼륨의 디스크 파티션 외부에 있는 디렉터리 경로 또는 유효하지 않은 디렉터리 경로가 지정되는 경우 데이터베이스 이름 변경 작업은 수행되지 않으며, **-i** 옵션과 병행될 수 없다.

```
cubrid renamedb -E /dbtemp/newaddvols/ testdb testdb_1
```

디렉터리 정보가 저장된 입력 파일을 지정(-i 또는 --control-file)

각 볼륨 또는 파일에 대하여 일괄적으로 데이터베이스 이름을 변경하면서 디렉터리 경로를 상이하게 지정하기 위하여, 디렉터리 정보가 저장된 입력 파일을 지정하는 명령으로서, **-i** 옵션을 이용한다. 이때, **-i** 옵션은 **-E** 옵션과 병행될 수 없다.

```
cubrid renamedb -i rename_path testdb testdb_1
```

다음은 개별적 볼륨들의 이름과 현재 디렉터리 경로, 그리고 변경된 이름의 볼륨들이 저장될 디렉터리 경로를 포함하는 파일의 구문 및 예시이다.

```
valid    source_fullvolname    dest_fullvolname
```

- *valid*: 각 볼륨을 식별하기 위한 정수이며, 데이터베이스 볼륨 정보 제어 파일(database_name_vinf)를 통해 확인할 수 있다.
- *source_fullvolname*: 각 볼륨에 대한 현재 디렉터리 경로이다.
- *dest_fullvolname*: 이름이 변경된 새로운 볼륨이 이동될 목적지 디렉터리 경로이다. 만약, 목적지 디렉터리가 유효하지 않은 경우 데이터베이스 이름 변경 작업은 수행되지 않는다.

```
-5 /home1/user/testdb vinf /home1/CUBRID/databases/testdb_1 vinf
-4 /home1/user/testdb lginf /home1/CUBRID/databases/testdb_1 lginf
-3 /home1/user/testdb_bkvinf /home1/CUBRID/databases/testdb_1_bkvinf
-2 /home1/user/testdb lgat /home1/CUBRID/databases/testdb_1_lgat
0 /home1/user/testdb /home1/CUBRID/databases/testdb_1
1 /home1/user/backup/testdb_x001 /home1/CUBRID/databases/backup/testdb_1_x001
```

백업 파일을 삭제하며 이름 변경(-d 또는 --delete-backup)

-d 옵션을 이용하여 testdb의 이름을 변경하면서 testdb와 동일 위치에 있는 모든 백업 볼륨 및 백업 정보 파일을 함께 강제 삭제하는 명령이다. 일단, 데이터베이스 이름이 변경되면 이전 이름의 백업 파일은 이용할 수 없으므로 주의해야 한다. 만약, -d 옵션을 지정하지 않으면 백업 볼륨 및 백업 정보 파일은 삭제되지 않는다.

```
cubrid renamedb -d testdb testdb_1
```

데이터베이스 등록

설명

cubrid installdb 유틸리티는 데이터베이스 위치 정보를 저장하는 databases.txt에 새로 설치된 데이터베이스 정보를 등록한다. 이 유틸리티의 실행은 등록 대상 데이터베이스의 동작에 영향을 끼치지 않는다.

구문

```
cubrid installdb options database name
options : [{--server-name=}host] [{-F | --file-path=} database_path ] [ {-L | --log-path=} log_path ]
```

- **cubrid**: CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **installdb**: 이동 또는 복사된 데이터베이스의 정보를 databases.txt에 등록하는 명령이다.
- **options**: --server-name, -F, -L 옵션이 제공되며, 각 옵션에 관한 설명은 옵션 설명 및 예제를 참조한다. 옵션을 생략하는 경우, 해당 데이터베이스가 존재하는 디렉터리에서 명령을 수행해야 한다.
- **database_name**: databases.txt에 등록하고자 하는 데이터베이스의 이름이다.

옵션

호스트 이름을 등록(--server-name)

대상 데이터베이스의 서버 호스트 정보를 지정된 호스트 명으로 databases.txt에 등록한다. 이 옵션을 생략하면, 현재의 호스트 정보가 등록된다.

```
cubrid installdb --server-name=cub_server1 testdb
```

데이터베이스 볼륨의 디렉터리 경로를 등록(-F 또는 --file-path)

-F 옵션을 이용하여 대상 데이터베이스 볼륨의 디렉터리 경로를 databases.txt에 등록한다. 이 옵션을 생략하면 기본값인 현재 디렉토리 경로가 등록된다.

```
cubrid installdb -F /home/cubrid/CUBRID/databases/testdb testdb
```

데이터베이스 로그 볼륨의 디렉터리 경로를 등록(-L 또는 --log-path)

-L 옵션을 이용하여 대상 데이터베이스 로그 볼륨의 디렉터리 경로를 databases.txt에 등록한다. 이 옵션을 생략하면 데이터베이스 볼륨의 디렉토리 경로가 등록된다.

```
cubrid installdb -L /home/cubrid/CUBRID/databases/logs/testdb testdb
```

사용 공간 확인

설명

cubrid spacedb 유틸리티는 사용 중인 데이터베이스 볼륨의 공간을 확인하기 위해서 사용된다.

cubrid spacedb 유틸리티는 데이터베이스에 있는 모든 영구 데이터 볼륨의 간략한 설명을 보여준다.

cubrid spacedb 유틸리티에 의해 반환되는 정보는 볼륨 ID와 이름, 각 볼륨의 목적, 각 볼륨과 관련된 총(total) 공간과 빈(free) 공간이다.

구문

```
cubrid spacedb options database name
options : [{-o|--output-file=}file] [--size_unit=PAGE|M|G|T|H] [-S|--SA-mode | -C|--CS-mode]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **spacedb** : 대상 데이터베이스에 대한 공간을 확인하는 명령으로 데이터베이스 서버가 구동 정지 상태인 경우에만 정상적으로 수행된다.
- **options** : -o, -S, -C, --size_unit 옵션이 제공되며, 각 옵션에 관한 설명은 옵션 설명 및 예제를 참고한다.
- **database_name** : 공간을 확인하고자 하는 데이터베이스의 이름이며, 데이터베이스가 생성될 디렉터리 경로명을 포함하지 않는다.

옵션

출력 메시지를 파일에 저장(-o)

testdb의 공간 정보에 대한 결과를 화면이 아닌 **db_output**이라는 파일에 저장하는 구문이다.

```
cubrid spacedb -o db_output testdb
```

독립 모드(stand-alone) 실행(-S 또는 --SA-mode)

-S 옵션은 서버 프로세스를 구동하지 않고 데이터베이스에 접근하는 독립 모드(standalone)로 작업하기 위해 지정되며, 인수는 없다. -S 옵션을 지정하지 않으면, 시스템은 클라이언트/서버 모드로 인식한다.

```
cubrid spacedb --SA-mode testdb
```

클라이언트/서버 모드 실행(-C 또는 --CS-mode)

-C 옵션은 서버 프로세스와 클라이언트 프로세스를 각각 구동하여 데이터베이스에 접근하는 클라이언트/서버 모드로 작업하기 위한 옵션이며, 인수는 없다. -C 옵션을 지정하지 않더라도 시스템은 기본적으로 클라이언트/서버 모드로 인식한다.

```
cubrid spacedb --CS-mode testdb
```

메가바이트 단위로 출력(--size_unit=M)

```
cubrid spacedb --size_unit=M testdb
```

보기 편한 형태로 출력(--size_unit=H)

1MB ≤ DB 크기 < 1024MB 이면 MB 단위로, 1GB ≤ DB 크기 < 1024GB 이면 GB 단위로 자동으로 결정해 준다

```
cubrid spacedb --size_unit=H testdb
```

사용 공간 정리

설명

cubrid compactdb 유틸리티는 데이터베이스 볼륨 중에 사용되지 않는 공간을 확보하기 위해서 사용된다. 데이터베이스 서버가 정지된 경우(offline)에는 독립 모드(stand-alone mode)로, 데이터베이스가 구동 중인 경우(online)에는 클라이언트 서버 모드(client-server mode)로 공간 정리를 수행할 수 있다.

cubrid compactdb 유틸리티는 삭제된 객체들의 OID와 클래스 변경에 의해 점유되고 있는 공간을 확보한다. 객체를 삭제하면 삭제된 객체를 참조하는 다른 객체가 있을 수 있기 때문에 삭제된 객체에 대한 OID는 바로 사용 가능한 빈 공간이 될 수 없다. **cubrid compactdb** 유틸리티를 수행하면 삭제된 객체에 대한 참조를 **NULL**로 표시하는데, 이렇게 **NULL**로 표시된 공간은 OID가 재사용할 수 있는 공간임을 의미한다.

구문

```
cubrid compactdb options database name [class name1, class name2, ...]  
options : [-v | --verbose] [-S | --SA-mode | -C | --CS-mode]
```

- **cubrid** : 큐브리드 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **compactdb** : 대상 데이터베이스에 대하여 삭제된 데이터에 할당되었던 OID가 재사용될 수 있도록 공간을 정리하는 명령으로서, 데이터베이스가 구동 정지 상태인 경우에만 정상적으로 수행된다.
- **options** : -v, -S, -C 옵션을 제공하며, 클라이언트/서버 모드에서만 -l, -i, -c, -d, -p 옵션을 사용할 수 있다.

- *database_name* : 공간을 정리할 데이터베이스의 이름이며, 데이터베이스가 생성될 디렉터리 경로명을 포함하지 않는다.
- *class_name_list* : 공간을 정리할 테이블 이름 리스트를 데이터베이스 이름 뒤에 직접 명시할 수 있으며, **-i** 옵션과 함께 사용할 수 없다. 클라이언트/서버 모드에서만 명시할 수 있다.

옵션

작업 중 메시지 상세 출력(-v)

-v 옵션을 이용하여 어느 클래스가 현재 정리되고 있는지, 얼마나 많은 인스턴스가 그 클래스를 위하여 처리되었는지를 알리는 메시지를 화면에 출력할 수 있다.

```
cubrid compactdb -v testdb
```

독립 모드(stand-alone) 실행(-S 또는 --SA-mode)

-S 옵션은 데이터베이스 서버가 구동 중단된 상태에서 독립 모드(standalone)로 공간 정리 작업을 수행하기 위해 지정되며, 인수는 없다. **-S** 옵션을 지정하지 않으면, 시스템은 클라이언트/서버 모드로 인식한다.

```
cubrid compactdb --SA-mode testdb
```

클라이언트/서버 모드 실행(-C 또는 --CS-mode)

-C 옵션은 데이터베이스 서버가 구동 중인 상태에서 클라이언트/서버 모드로 공간 정리 작업을 수행하기 위해 지정되며, 인수는 없다. **-C** 옵션이 생략되더라도 시스템은 기본적으로 클라이언트/서버 모드로 인식한다. 다음은 클라이언트/서버 모드에서만 사용할 수 있는 옵션이다.

- **-i, --input-class-file=FILE** : 이 옵션을 사용하여 대상 테이블 이름을 포함하는 입력 파일 이름을 지정할 수 있다. 라인 당 하나의 테이블 이름을 명시하며, 유효하지 않은 테이블 이름은 무시된다. 이 옵션을 지정하는 경우, 데이터베이스 이름 뒤에 대상 테이블 이름 리스트를 직접 명시할 수 없으므로 주의한다.
- **-p, --pages-commited-once=NUMBER** : 이 옵션을 사용하여 한 번에 커밋할 수 있는 최대 페이지 수를 지정할 수 있다. 디폴트 값은 **10**이며, 최소 값은 1, 최대 값은 10이다. 옵션 값이 작으면 클래스/인스턴스에 대한 잠금 비용이 작으므로 동시성은 향상될 수 있으나 작업 속도는 저하될 수 있고, 옵션 값이 크면 동시성은 저하되나 작업 속도는 향상될 수 있다.
- **-d, --delete-old-repr** : 이 옵션을 사용하여 카탈로그에서 과거 테이블 표현을 삭제할 수 있다.
- **-l, --instance-lock-timeout** : 이 옵션을 사용하여 인스턴스 잠금 타임아웃 값을 지정할 수 있다. 디폴트 값은 **2(초)**이며, 최소 값은 1, 최대 값은 10이다. 설정된 시간동안 잠금 인스턴스를 대기하므로, 옵션 값이 작을수록 작업 속도는 향상될 수 있으나 처리 가능한 인스턴스 개수가 적어진다. 반면, 옵션 값이 클수록 작업 속도는 저하되나 더 많은 인스턴스에 대해 작업을 수행할 수 있다.
- **-c, --class-lock-timeout** : 이 옵션을 사용하여 클래스 잠금 타임아웃 값을 지정할 수 있다. 디폴트 값은 **10(초)**이며, 최소값은 1, 최대 값은 10이다. 설정된 시간동안 잠금 테이블을 대기하므로, 옵션 값이 작을수록 작업 속도는 향상될 수 있으나 처리 가능한 테이블 개수가 적어진다. 반면, 옵션 값이 클수록 작업 속도는 저하되나 더 많은 테이블에 대해 작업을 수행할 수 있다.

```
cubrid compactdb --CS-mode -p 10 testdb tbl1, tbl2, tbl5
```

통계 정보 갱신

설명

CUBRID의 질의 최적화기가 사용하는 테이블에 있는 객체들의 수, 접근하는 페이지들의 수, 속성 값들의 분산 같은 통계 정보를 갱신한다.

구문

```
cubrid optimizedb options database_name  
options : [{-n|--class-name=} name]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **optimizedb** : 대상 데이터베이스에 대하여 비용 기반 질의 최적화에 사용되는 통계 정보를 업데이트한다. 옵션을 지정하는 경우, 지정한 클래스에 대해서만 업데이트한다.
- **options** : **-n** 옵션을 지원한다.
- **database_name** : 비용기반 질의 최적화용 통계 자료를 업데이트하려는 데이터베이스 이름이다.

옵션

대상 데이터베이스에 대해 질의 통계 업데이트

대상 데이터베이스의 전체 클래스의 질의 통계 정보를 업데이트하는 명령이다.

```
cubrid optimizedb testdb
```

대상 데이터베이스의 특정 클래스에 대해 질의 통계 업데이트(-n 또는 --class-name)

-n 옵션을 이용하여 해당 클래스의 질의 통계 정보를 업데이트하는 명령이다.

```
cubrid optimizedb -n event_table testdb
```

데이터베이스 서버 실행 통계 정보 출력

설명

cubrid statdump 유틸리티를 이용해 CUBRID 데이터베이스 서버가 실행한 통계 정보를 확인할 수 있으며, 통계 정보 항목은 크게 File I/O 관련, 페이지 버퍼 관련, 로그 관련, 트랜잭션 관련, 동시성 관련, 인덱스 관련, 쿼리 수행 관련, 네트워크 요청 관련으로 구분된다. 단, 유틸리티 실행 전에 **cubrid.conf** 파일에 **communication_histogram** 파라미터를 **yes**로 설정해야 한다. 또한, csql에서 세션 명령어(**;.h on** 또는 **;.h all**)을 이용하여 서버의 통계 정보를 확인할 수 있다.

구문

```
cubrid statdump options database_name  
options : [{-o|--output-file=} file_name] [{-i|--interval=}secs] [-c|--cumulative]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.

- **statdump** : 대상 데이터베이스 서버 실행 통계 정보를 출력하는 명령어이다. 데이터베이스가 동작 중일 때에만 정상 수행된다.
- *options* : **-o** 옵션, **-i** 옵션, **-c** 옵션을 지원한다.
- *database_name* : 통계 자료를 확인하고자 하는 대상 데이터베이스 이름이다.

옵션

실행 통계 정보를 주기적으로 출력(-i 또는 --interval)

```
cubrid statdump -i 5 testdb

Wed March 31 11:23:56 KST 2010

*** SERVER EXECUTION GLOBAL STATISTICS ***
Num file creates           = 0
Num file removes           = 0
Num file ioreads           = 0
Num file iowrites          = 0
Num file iosynches         = 0
Num data page fetches      = 0
Num data page dirties      = 0
Num_data_page_ioreads      = 0
Num_data_page_iowrites     = 0
Num_data_page_victims      = 0
Num_data_page_iowrites_for_replacement = 0
Num_log_page_ioreads       = 0
Num_log_page_iowrites      = 0
Num_log_append_records     = 0
Num_log_archives           = 0
Num_log_checkpoints        = 0
Num_log_wals               = 0
Num_page_locks_acquired    = 0
Num object locks acquired  = 0
Num page locks converted   = 0
Num object locks converted  = 0
Num page locks re-requested = 0
Num object locks re-requested = 0
Num_page_locks_waits       = 0
Num object locks waits     = 0
Num_tran_commits           = 0
Num_tran_rollbacks         = 0
Num_tran_savepoints        = 0
Num_tran_start_topops      = 0
Num_tran_end_topops        = 0
Num_tran_interrupts        = 0
Num_btree_inserts          = 0
Num_btree_deletes          = 0
Num_btree_updates          = 0
Num_query_selects          = 0
Num_query_inserts          = 0
Num_query_deletes          = 0
Num_query_updates          = 0
Num_query_scans            = 0
Num_query_iscans           = 0
Num_query_lscans           = 0
Num_query_setscans         = 0
Num_query_methscans        = 0
Num_query_nljoins          = 0
Num_query_mjoins           = 0
Num_query_objfetches       = 0
Num_network_requests       = 2
Num_adaptive_flush_pages   = 0
Num_adaptive_flush_log_pages = 0
Num_adaptive_flush_max_pages = 90

*** OTHER STATISTICS ***
Data_page_buffer_hit_ratio = 0.00
```

데이터베이스 서버 실행 통계 정보 항목

분류	항목	설명
File I/O 관련	Num_file_removes	삭제한 파일 개수
	Num_file_creates	생성한 파일 개수
	Num_file_ireads	디스크로부터 읽은 횟수
	Num_file_iowrites	디스크로 저장한 횟수
	Num_file_iosynches	디스크와 동기화를 수행한 횟수
페이지 버퍼 관 련	Num_data_page_fetches	가져오기(fetch)한 페이지 수
	Num_data_page_dirties	더티 페이지 수
	Num_data_page_ireads	읽은 페이지 수
	Num_data_page_iowrites	저장한 페이지 수
로그 관 련	Num_log_page_ireads	읽은 로그 페이지의 수
	Num_log_page_iowrites	저장한 로그 페이지의 수
	Num_log_append_records	추가(append)한 로그 레코드의 수
	Num_log_archives	보관 로그의 개수
	Num_log_checkpoints	체크포인트 수행 횟수
트랜잭 션 관련	Num_tran_commits	커밋한 횟수
	Num_tran_rollback	롤백한 횟수
	Num_tran_savepoints	세이브포인트 횟수
	Num_tran_start_topops	시작한 top operation 의 개수
	Num_tran_end_topops	종료한 top operation 의 개수
	Num_tran_interrupts	인터럽트 개수
동시성/ 잠금 관 련	Num_page_locks_acquired	페이지 잠금을 획득한 횟수
	Num_object_locks_acquired	오브젝트 잠금을 획득한 횟수
	Num_page_locks_converted	페이지 잠금 타입을 변환한 횟수
	Num_object_locks_converted	오브젝트 잠금 타입을 변환한 횟수
	Num_page_locks_re-requested	페이지 잠금을 재요청한 횟수
	Num_object_locks_re-requested	오브젝트 잠금을 재요청한 횟수
	Num_page_locks_waits	잠금을 대기하는 페이지 개수

	Num_object_locks_waits	잠금을 대기하는 오브젝트 개수
인덱스 관련	Num_btree_inserts	삽입된 항목의 개수
	Num_btree_deletes	삭제된 항목의 개수
	Num_btree_updates	갱신된 항목의 개수
쿼리 관 련	Num_query_selects	SELECT 쿼리의 수행 횟수
	Num_query_inserts	INSERT 쿼리의 수행 횟수
	Num_query_deletes	DELETE 쿼리의 수행 횟수
	Num_query_updates	UPDATE 쿼리의 수행 횟수
	Num_query_sscans	순차 스캔(풀 스캔) 횟수
	Num_query_iscans	인덱스 스캔 횟수
	Num_query_lscans	LIST 스캔 횟수
	Num_query_setscans	SET 스캔 횟수
	Num_query_methscans	METHOD 스캔 횟수
	Num_query_nljoins	Nested Loop 조인 횟수
	Num_query_mjoins	병합 조인 횟수
	Num_query_objfetches	객체를 가져오기(fetch)한 횟수
네트워크 요청 관련	Num_network_requests	네트워크 요청 횟수
	Data_page_buffer_hit_ratio	페이지 버퍼의 Hit Ratio (Num_data_page_fetches - Num_data_page_ireads)*100 / Num_data_page_fetches

대상 데이터베이스 서버에 대한 실행 통계 정보를 파일에 저장(-o 또는 --output-file)

-o 옵션을 이용하여 대상 데이터베이스 서버의 실행 통계 정보를 지정된 파일에 저장할 수 있다.

```
cubrid statdump -o statdump.log testdb
```

누적된 실행 통계 정보를 출력(-c 또는 --cumulative)

-c 옵션을 이용하여 대상 데이터베이스 서버의 누적된 실행 통계 정보를 출력할 수 있다. -i 옵션과 결합하면, 지정된 시간 간격(interval)마다 리셋되지 않는 실행 통계 정보를 확인할 수 있다.

```
cubrid statdump -i 5 -c testdb
```

잠금(Lock) 상태 확인

설명

cubrid lockdb는 대상 데이터베이스에 대하여 현재 트랜잭션에서 사용되고 있는 잠금 정보를 확인하는 유틸리티이다.

구문

```
cubrid lockdb options database_name
options : [{-o|--output-file=} file ]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **lockdb** : 대상 데이터베이스에 대하여 현재 트랜잭션에서 사용되고 있는 잠금 정보를 확인하는 명령이다.
- **options** : **-o** 옵션을 지원한다.
- **database_name** : 현재 트랜잭션의 잠금 정보를 확인하는 데이터베이스 이름이다.

옵션

잠금 정보를 화면에 출력

옵션없이 testdb 데이터베이스의 잠금 정보를 화면에 출력하는 구문이다.

```
cubrid lockdb testdb
```

잠금 정보를 지정한 파일에 출력(-o)

-o 옵션을 이용하여 testdb 데이터베이스의 잠금 정보를 output.txt로 출력하는 구문이다.

```
cubrid lockdb -o output.txt testdb
```

데이터베이스 일관성 확인

설명

cubrid checkdb 유틸리티는 데이터베이스를 확인하기 위해 사용된다. **cubrid checkdb** 유틸리티를 사용하면 인덱스와 다른 데이터 구조를 확인하기 위해 데이터와 로그 블록의 내부적인 물리적 일치를 확인할 수 있다. 만일 **cubrid checkdb** 유틸리티의 실행 결과가 불일치로 나온다면 **--repair** 옵션으로 자동 수정을 시도해 보아야 한다.

구문

```
cubrid checkdb options database_name
options : [-S|--SA-mode | -C|--CS-mode] [-r | --repair]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **checkdb** : 대상 데이터베이스에 대하여 데이터의 일관성(consistency)을 확인하는 명령이다.
- **options** : **-S**, **-C**, **-r** 옵션을 지원한다.

- *database_name* : 일관성을 확인하거나 복구하려는 데이터베이스 이름이다.

옵션

독립 모드에서 데이터베이스의 일관성 확인(-S 또는 --SA-mode)

-S 옵션은 서버 프로세스를 구동하지 않고 데이터베이스에 접근하는 독립 모드(standalone)로 작업하기 위해 지정되며, 인수는 없다. -S 옵션을 지정하지 않으면, 시스템은 클라이언트/서버 모드로 인식한다.

```
cubrid checkdb -S testdb
```

클라이언트 - 서버 모드에서 데이터베이스의 일관성 확인(-C 또는 --CS-mode)

-C 옵션은 서버 프로세스와 클라이언트 프로세스를 각각 구동하여 데이터베이스에 접근하는 클라이언트/서버 모드로 작업하기 위한 옵션이며, 인수는 없다. -C 옵션을 지정하지 않더라도 시스템은 기본적으로 클라이언트/서버 모드로 인식한다.

```
cubrid checkdb -C testdb
```

데이터베이스의 일관성에 문제가 있을 때 복구(-r 또는 --repair)

-r 옵션은 데이터베이스의 일관성에 문제가 발견되었을 때 복구를 수행한다.

```
cubrid checkdb -r testdb
```

데이터베이스 트랜잭션 제거

설명

cubrid killtran은 대상 데이터베이스의 트랜잭션을 확인하거나 특정 트랜잭션을 강제 종료하는 유틸리티로서, DBA 사용자만 수행할 수 있다.

구문

```
cubrid killtran options database_name
options :
[{-i|--kill-transaction-index=}index] [--kill-user-name=id] [--kill-host-name=host] [--kill-program-name=program name] [{-p|--dba-password=}password] [-d|--display-information] [-f|--force]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **killtran** : 지정된 데이터베이스에 대해 트랜잭션을 관리하는 명령어이다.
- *options* : 옵션에 따라 특정 트랜잭션을 지정하여 제거하거나, 현재 활성화된 트랜잭션을 화면 출력할 수 있다. 옵션이 지정되지 않으면, -d 옵션이 디폴트로 적용되어 모든 트랜잭션을 화면 출력한다. -p 옵션 뒤에 오는 값은 DBA의 암호이며 생략하면 프롬프트에서 입력해야 한다.
- *database_name* : 대상 데이터베이스의 이름이다.

옵션

모든 트랜잭션의 정보 출력(옵션 생략)

```
cubrid killtran testdb
```

Tran index	User name	Host name	Process id	Program name
1(+)	dba	myhost	664	cub_cas
2(+)	dba	myhost	6700	csql
3(+)	dba	myhost	2188	cub_cas
4(+)	dba	myhost	696	csql
5(+)	public	myhost	6944	csql

지정한 인덱스에 해당하는 트랜잭션 제거(-i 또는 --kill-transaction-index)

```
cubrid killtran -i 1 testdb
```

Ready to kill the following transactions:

Tran index	User name	Host name	Process id	Program name
1(+)	dba	myhost	4760	csql

Do you wish to proceed ? (Y/N)y

Killing transaction associated with transaction index 1

모든 트랜잭션의 정보 출력(-d 또는 --display)

```
cubrid killtran -d testdb
```

Tran index	User name	Host name	Process id	Program name
2(+)	dba	myhost	6700	csql
3(+)	dba	myhost	2188	cub_cas
4(+)	dba	myhost	696	csql
5(+)	public	myhost	6944	csql

지정한 OS 사용자 ID에 해당하는 트랜잭션 제거(--kill-user-name)

```
cubrid killtran --kill-user-name=os_user_id testdb
```

지정한 클라이언트 호스트의 트랜잭션 제거(--kill- host-name)

```
cubrid killtran --kill-host-name=myhost testdb
```

지정한 프로그램에 해당하는 트랜잭션 제거(--kill-program-name)

```
cubrid killtran --kill-program-name=cub_cas testdb
```

중지할 트랜잭션을 확인하는 프롬프트 생략(-f 또는 --force)

```
cubrid killtran -f -i 1 testdb
```

질의 수행 계획 캐시 확인

설명

cubrid plandump 유틸리티를 사용해서 서버에 저장(캐시)되어 있는 질의 수행 계획들의 정보를 출력할 수 있다.

구문

```
cubrid plandump options database_name  
options : [-d|--drop] [{-o|--output-file=} file]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **plandump** : 대상 데이터베이스에 대하여 현재 캐시에 저장되어 있는 질의 수행 계획을 출력하는 명령이다.
- *options* : **-d**와 **-o** 옵션만 지원한다.
- *database_name* : 데이터베이스 서버 캐시로부터 질의 수행 계획을 확인 또는 제거하고자 하는 데이터베이스 이름이다

옵션

캐시에 저장된 질의 수행 계획 확인

```
cubrid plandump testdb
```

캐시에 저장된 질의 수행 계획 제거(-d 또는 --drop)

```
cubrid plandump -d testdb
```

캐시에 저장된 질의 수행 계획 결과 파일에 저장(-o 또는 --output)

```
cubrid plandump -o output.txt testdb
```

데이터베이스 로그 응급 복구

설명

cubrid emergency_patchlog 유틸리티는 시스템 충돌 등으로 인해 데이터베이스의 로그 파일에 손상이 생겨 데이터베이스가 강제 종료되고 재시작되지 않는 상황에서 손상된 로그 파일을 복구하고자 할 때 사용한다. 옵션 없이 유틸리티를 실행하면 손상된 데이터베이스 로그를 수정하고, **-r** 옵션을 지정하면 로그 파일을 새로 생성한다.

구문

```
cubrid emergency_patchlog options database_name
options : [ -r | --recreate-log ]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **emergency_patchlog** : 데이터베이스가 구동 정지 상태인 경우에만 정상적으로 수행된다. 손상된 로그를 복구하기 위하여 처음에는 반드시 **-r** 옵션 없이 유틸리티를 실행한다.
- *options* : 필요한 로그가 존재하지 않거나 손상된 로그를 복구할 수 없는 경우에만 **-r** 옵션을 사용하여 로그를 재생성한다. 손상된 데이터베이스 로그를 범용(generic) 로그 볼륨 크기로 생성한다.
- *database_name* : 복구하려는 데이터베이스 이름이다.

옵션

기존 로그를 이용하여 복구 시도(1차 수행)

```
cubrid emergency_patchlog testdb
```

기존 로그를 버리고 빈 로그를 새로 생성(2차 수행)

```
cubrid emergency_patchlog -r testdb
```

데이터베이스 내부 정보 출력

설명

cubrid diagdb 유틸리티를 이용해 다양한 데이터베이스 내부 정보를 확인할 수 있다. **cubrid diagdb** 유틸리티가 제공하는 정보들은 현재 데이터베이스의 상태를 진단하거나 문제를 파악하는데 도움이 된다.

구문

```
cubrid diagdb options database_name  
options : [{-d | --dump-type=} type]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **diagdb** : CUBRID에 저장되는 바이너리 형태의 파일 정보를 텍스트 형태로 출력하여 현재의 데이터베이스 저장 상태를 확인하고자 할 때 사용하는 명령어이다. 데이터베이스가 구동 정지 상태인 경우에만 정상적으로 수행된다. 전체를 확인하거나 옵션을 사용하여 파일 테이블, 파일 용량, 힙 용량, 클래스 이름, 디스크 비트맵을 선택해 확인할 수 있다.
- *options* : **-d** 옵션이 제공된다.
- *database_name* : 설치하려는 데이터베이스 이름이다.

옵션

출력 범위 지정(-d 또는 --dump-type)

testdb라는 데이터베이스의 전체 파일에 대한 기록 상태를 출력할 때 사용된다. 생략하면 디폴트 값인 1이 지정된다.

```
cubrid diagdb -d 1 myhost testdb
```

-d 옵션에 적용되는 타입은 모두 9가지로, 그 종류는 다음과 같다.

타입	설명
-1	전체 데이터베이스 정보를 출력한다.
1	파일 테이블 정보를 출력한다.
2	파일 용량 정보를 출력한다.
3	힙 용량 정보를 출력한다.
4	인덱스 용량 정보를 출력한다.
5	클래스 이름 정보를 출력한다.
6	디스크 비트맵 정보를 출력한다.
7	카탈로그 정보를 출력한다.
8	로그 정보를 출력한다.

백업 및 복구

데이터베이스 관리자(DBA)는 시스템의 장애에 대비하여 데이터베이스를 일정 시점의 데이터베이스로 복구할 수 있도록 주기적으로 백업을 수행해야 한다. 상세한 내용은 [데이터베이스 백업](#)을 참조한다.

내보내기와 가져오기

신규 버전의 CUBRID 데이터베이스를 사용하기 위해서는 기존 버전의 CUBRID 데이터베이스를 신규 버전의 CUBRID 데이터베이스로 이전하는 작업을 진행해야 할 경우가 있다. 이때 CUBRID에서 제공하는 텍스트 파일로 내보내기와 텍스트 파일에서 가져오기 기능을 활용할 수 있다. 내보내기와 가져오기에 대한 보다 자세한 설명은 [데이터베이스 마이그레이션](#)을 참고한다.

서버/클라이언트에서 사용하는 파라미터 출력

설명

cubrid paramdump 유틸리티는 서버/클라이언트 프로세스에서 사용하는 파라미터 정보를 출력한다.

구문

```
cubrid paramdump options database_name
options : [{-o|--output-file=}filename] [{-b|--both}] [{-S|--SA-mode}] [{-C|--CS-mode}]
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **paramdump** : 서버/클라이언트 프로세스에서 사용하는 파라미터 정보를 출력하는 명령이다.
- **options** : 단축 옵션은 -와 함께 지정해야 하고, 전체 옵션은 --와 함께 지정해야 한다. **-o**, **-b**, **-S**, **-C** 옵션이 제공된다.
- **database_name** : 파라미터 정보를 출력할 데이터베이스 이름이다.

옵션

출력 정보를 파일에 저장 (-o)

-o 옵션은 데이터베이스의 서버/클라이언트 프로세스에서 사용하는 파라미터 정보를 지정된 파일에 저장하는 옵션이며, 파일은 현재 디렉터리에 생성된다. **-o** 옵션이 지정되지 않으면 메시지는 콘솔 화면에 출력한다.

```
cubrid paramdump -o db_output testdb
```

서버/클라이언트 파라미터 정보 출력 (-b)

-b 옵션은 데이터베이스의 서버/클라이언트 프로세스에서 사용하는 파라미터 정보를 콘솔 화면에 출력하는 옵션이며, **-b** 옵션을 사용하지 않으면 서버 프로세스의 파라미터 정보만 출력한다.

```
cubrid paramdump -b testdb
```

독립 모드에서 서버 프로세스의 파라미터 정보 출력 (-S 또는 --SA-mode)

```
cubrid paramdump -S testdb
```

클라이언트 - 서버 모드에서 서버 프로세스의 파라미터 정보 출력 (-C 또는 --CS-mode)

```
cubrid paramdump -C testdb
```

데이터베이스 마이그레이션

데이터베이스 마이그레이션

신규 버전의 CUBRID 데이터베이스를 사용하기 위해서는 기존 버전의 CUBRID 데이터베이스를 신규 버전의 CUBRID 데이터베이스로 이전하는 작업을 진행해야 할 경우가 있다. 이때 CUBRID에서 제공하는 텍스트 파일로 내보내기과 텍스트 파일에서 가져오기 기능을 활용할 수 있다. **cubrid unloaddb** 및 **cubrid loaddb** 유틸리티를 이용하는 마이그레이션 절차를 설명한다.

권장 시나리오 및 절차

기존 버전의 CUBRID가 운영 중인 상태에서 적용할 수 있는 마이그레이션 시나리오를 설명한다.

데이터베이스 마이그레이션을 위해서는 **cubrid unloaddb**와 **cubrid loaddb** 유틸리티를 사용한다. 자세한 내용은 [데이터베이스 내보내기\(unload\)](#) 및 [데이터베이스 가져오기\(load\)](#)를 참조한다.

1. 기존 데이터베이스 백업

cubrid backupdb 유틸리티를 이용하여 기존 버전의 데이터베이스 백업을 수행한다. 그 이유는 데이터베이스 언로드/로드 작업 중 발생 가능한 장애에 대비하기 위함이다. 데이터베이스 백업에 관한 자세한 내용은 [데이터베이스 백업](#)을 참조한다.

2. 기존 데이터베이스 언로드

cubrid unloaddb 유틸리티를 이용하여 기존 버전의 CUBRID에서 생성된 데이터베이스를 언로드한다. 데이터베이스 언로드에 관한 자세한 내용은 [데이터베이스 내보내기\(unload\)](#)를 참조한다.

3. 기존 CUBRID의 환경 설정 파일 보관

CUBRID/conf 디렉터리 아래의 **cubrid.conf**, **cubrid_broker.conf**, **cm.conf** 등의 환경 설정 파일을 보관한다. 이는 기존 CUBRID 데이터베이스 환경에 적용된 파라미터 설정값을 신규 CUBRID 데이터베이스 환경에서 편리하게 적용할 수 있기 때문이다.

4. 신규 버전의 CUBRID 설치

기존 버전의 CUBRID에서 생성된 데이터의 백업 및 언로드 작업이 완료되었으므로, 기존 버전의 CUBRID 및 데이터베이스를 삭제하고 신규 버전의 CUBRID를 설치한다. CUBRID 설치에 대한 자세한 내용은 "CUBRID 시작"의 [설치 방법](#)을 참조한다.

5. 신규 CUBRID의 환경 설정

기존 CUBRID의 환경 설정 파일 보관하기에서 보관한 기존 데이터베이스의 환경 설정 파일을 참고하여 신규 버전의 CUBRID 환경을 설정할 수 있다. 환경 설정에 대한 자세한 내용은 "CUBRID 시작"의 [설치 방법](#)을 참조한다.

6. 신규 데이터베이스 로드

cubrid createdb 유틸리티를 이용하여 데이터베이스를 생성하고, **cubrid loaddb** 유틸리티를 이용하여 언로드한 데이터를 해당 데이터베이스에 로드한다. 데이터베이스 생성에 대한 자세한 내용은 "관리자 안

내서"의 [데이터베이스 생성](#)을 참조하고, 데이터베이스 로드와 관련한 자세한 내용은 [데이터베이스 가져오기\(load\)](#)를 참조한다.

7. 신규 데이터베이스 백업

신규 데이터베이스에 데이터 로딩이 완료되면, **cubrid backupdb** 유틸리티를 이용하여 신규 버전의 CUBRID 환경에서 생성된 데이터베이스를 백업한다. 그 이유는 기존 버전의 CUBRID 환경에서 백업한 데이터를 신규 버전의 CUBRID 환경에서 복구할 수 없기 때문이다. 데이터베이스 백업에 대한 자세한 내용은 [데이터베이스 백업](#)을 참조한다.

데이터베이스 내보내기(unload)

설명

데이터베이스를 언로드/로드하는 목적은 다음과 같다.

- 데이터베이스 볼륨을 재구성하여 데이터베이스 재구축
- 시스템이 다른 환경에서 마이그레이션 수행
- 버전이 다른 DBMS에서 마이그레이션 수행

구문

```
cubrid unloaddb [ options ] database name
[ options ]
-i | -O | -s | -d | -v | -S | -C |
--input-class-file | --output-path | --schema-only | --data-only | --verbose | --SA-mode |
--CS-mode | --include-reference | --input-class-only | --lo-count | --estimated-size | --
cached-pages | --output-prefix | --hash-file | --use-delimiter
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **unloaddb** : 데이터베이스를 아스키 파일로 생성하는 유틸리티로 시스템 교체, 제품 버전 업그레이드, 또는 데이터베이스 볼륨의 재구성을 위해 **cubrid loaddb** 유틸리티와 함께 사용된다. 독립 모드 및 클라이언트/서버 모드에서 모두 사용할 수 있으며, 해당 데이터베이스가 운영 중일 때에도 데이터를 언로드할 수 있다.
- *options* : 단축 옵션은 -와 함께 지정해야 하고, 전체 옵션은 --와 함께 지정해야 한다. 대소문자에 유의한다.
- *database_name* : 언로드할 데이터베이스의 이름을 지정한다.

반환 값

cubrid unloaddb 유틸리티의 리턴 값은 다음과 같다.

- 0 : 성공
- Non-zero : 실패

생성 파일

- 스키마 파일(*database-name_schema*) : 해당 데이터베이스에 정의된 스키마 정보를 포함하는 파일이다.
- 객체 파일(*database-name_objects*) : 해당 데이터베이스에 포함된 인스턴스 정보를 포함하는 파일이다.

- 인덱스 파일(*database-name_indexes*) : 해당 데이터베이스에 정의된 인덱스 정보를 포함하는 파일이다.
- 트리거 파일(*database-name_trigger*) : 해당 데이터베이스에 정의된 트리거 정보를 포함하는 파일이다.
만약 데이터를 로딩하는 동안 트리거가 구동되는 것을 원치 않는다면, 데이터 로딩을 완료한 후에 트리거 정의를 로딩하면 된다.

이러한 스키마, 객체, 인덱스, 트리거 파일은 같은 디렉터리에 생성된다.

옵션

다음은 **cuprid unloaddb** 유틸리티와 함께 사용할 수 있는 옵션을 정리한 표이다. 대소문자를 구별해서 사용해야 한다.

옵션	설명
-i --input-class-file	인수로 지정된 입력 파일에 지정된 클래스를 대상으로 데이터베이스를 언로드한다.
-O --output-path	스키마와 객체 파일이 생성될 디렉터리를 지정한다. 옵션이 지정되지 않으면 현재 디렉터리에 생성된다.
-s --schema-only	데이터 파일은 생성하지 않고, 스키마 파일만 생성한다.
-d --data-only	스키마 파일은 생성하지 않고, 데이터 파일만 생성한다.
-v --verbose	언로드되는 데이터베이스의 상세 정보를 화면에 출력한다.
-S --SA-mode	독립 모드에서 데이터베이스를 언로드한다.
-C --CS-mode	클라이언트/서버 모드에서 데이터베이스를 언로드한다.
--include-reference	-i 옵션과 함께 사용되며, 객체 참조도 함께 생성한다.
--input-class-only	-i 옵션과 함께 사용되며, 입력 파일에 포함된 테이블에 관한 스키마 파일만 생성한다.
--lo-count	한 디렉터리에 생성될 큰 객체(LO) 데이터 파일의 수를 설정한다. 디폴트 값 : 0
--estimated-size	예상되는 레코드 수를 지정한다.
--cached-pages	메모리에 캐시할 객체 테이블의 수를 설정한다. 디폴트 값 : 100
--output-prefix	스키마와 객체 파일명 앞에 붙이는 prefix 를 지정한다.
--hash-file	해시 파일의 이름을 지정한다.

--use-delimiter 속성 이름을 따옴표로 감싸서 출력한다.

엔로드할 테이블 목록이 포함된 입력 파일(-i 또는 --input-class-file)

다음은 입력 파일 table_list.txt의 예이다.

```
table_1
table_2
..
table_n
```

-i 옵션은 데이터베이스의 일부 클래스만 엔로드하기 위하여, 엔로드할 테이블 목록이 저장된 입력 파일을 지정한다.

```
cubrid unloaddb -i table_list.txt demodb
```

-i 옵션이 --input-class-only와 결합되면, 입력 파일에 포함된 테이블에 관한 스키마 파일만 생성된다.

```
cubrid unloaddb --input-class-only -i table_list.txt demodb
```

-i 옵션이 --include-reference와 결합되면, 객체 참조도 함께 생성된다.

```
cubrid unloaddb --include-reference -i table_list.txt demodb
```

생성 파일이 저장되는 디렉터리 지정(-O 또는 --output-path)

-O 옵션은 엔로드 작업을 통해 생성되는 출력 파일이 저장되는 디렉터리를 지정하기 위한 옵션이다. 만약, -O 옵션이 지정되지 않으면 현재 작업 디렉터리에 출력 파일이 생성된다.

```
cubrid unloaddb -O ./CUBRID/Databases/demodb demodb
```

지정된 디렉터리가 존재하지 않는 경우 다음과 같은 예러 메시지가 출력된다.

```
unloaddb: No such file or directory.
```

스키마 파일만 생성(-s 또는 --schema-only)

-s 옵션은 엔로드 작업을 통해 생성되는 출력 파일 중 스키마 파일만 생성되도록 지정하는 옵션이다.

```
cubrid unloaddb -s demodb
```

데이터 파일만 생성(-d 또는 --data-only)

-d 옵션은 엔로드 작업을 통해 생성되는 출력 파일 중, 데이터 파일만 생성되도록 지정하는 옵션이다.

```
cubrid unloaddb -d demodb
```

엔로드 상태 정보 출력(-v 또는 --verbose)

-v 옵션은 엔로드 작업이 진행되는 동안 엔로드되는 데이터베이스의 테이블 및 인스턴스에 관한 상세 정보를 화면에 출력하는 옵션이다.

```
cubrid unloaddb -v demodb
```

독립 모드(-S 또는 --SA-mode)

-S 옵션은 지정된 데이터베이스에 독립 모드로 접근하여 엔로드 작업을 수행하는 옵션이다.

```
cubrid unloaddb -S demodb
```

클라이언트/서버 모드(-C 또는 --CS-mode)

-C 옵션은 지정된 데이터베이스에 클라이언트/서버 모드로 접근하여 언로드 작업을 수행하는 옵션이다.

```
cubrid unloaddb -C demodb
```

예상되는 레코드 수(--estimated-size)

--estimated-size 옵션은 언로드할 데이터베이스의 레코드 저장에 필요한 해시 메모리를 사용자 임의로 할당하기 위한 옵션이다. 만약 --estimated-size 옵션이 지정되지 않으면 최근의 통계 정보를 기반으로 데이터베이스의 레코드 수를 결정하게 되는데, 만약 최근 통계 정보가 갱신되지 않았거나 해시 메모리를 크게 할당하고 싶은 경우 이 옵션을 이용할 수 있다. 따라서, 옵션의 인수로 너무 적은 레코드 개수를 정의한다면 해시 충돌로 인해 언로드 성능이 저하된다.

```
cubrid unloaddb --estimated-size 1000 demodb
```

캐시되는 페이지 수(--cached-pages)

--cached-pages는 메모리에 캐시되는 테이블의 페이지 수를 지정하기 위한 옵션이다. 각 페이지는 4,096 바이트이며, 관리자는 메모리의 크기와 속도를 고려하여 캐시되는 페이지 수를 지정할 수 있다. 만약, 이 옵션이 지정되지 않으면 디폴트 값은 100페이지가 된다.

```
cubrid unloaddb --cached-pages 500 demodb
```

생성 파일명의 프리픽스 지정(--output-prefix)

--output-prefix는 언로드 작업에 의해 생성되는 스키마 파일과 객체 파일의 이름 앞에 붙는 prefix를 지정하기 위한 옵션이다. 예를 들어 수행하면 스키마 파일명은 abcd_schema가 되고, 객체 파일명은 abcd_objects가 된다. 만약, --output-prefix 옵션을 지정하지 않으면 언로드할 데이터베이스 이름이 prefix로 사용된다.

```
cubrid unloaddb --output-prefix abcd demodb
```

데이터베이스 가져오기(load)

설명

데이터베이스 로드는 다음과 같은 경우에 **cubrid loaddb** 유틸리티를 이용하여 수행된다.

- 이전 버전의 CUBRID 데이터베이스를 새로운 버전의 데이터베이스로 마이그레이션하는 경우
- 타 DBMS의 데이터베이스를 CUBRID 데이터베이스로 마이그레이션하는 경우
- **INSERT** 구문 실행보다 빠른 성능으로 대용량 데이터를 입력하는 경우

일반적으로 **cubrid loaddb** 유틸리티는 **cubrid unloaddb** 유틸리티가 생성한 파일(스키마 정의 파일, 객체 입력 파일, 인덱스 정의 파일)을 사용한다.

구문

```
cubrid loaddb [ options ] database_name
[ options ]
-u | -p | -l | -v | -c | -s | -i | -d |
--user | --password | --load-only | --verbose | --periodic-commit | --schema-file | --
index-file | --data-file | --data-file-check-only | --estimated-size | --no-oid | --no-
statistics | --ignore-class-file | --error-control-file |
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **loaddb** : 엔로드 작업에 의해 생성된 파일을 로드하여 새로운 데이터베이스를 생성하는 유틸리티로 사용자가 작성한 별도의 입력 파일을 로드하여 대용량 데이터를 데이터베이스에 보다 신속하게 입력하는 목적으로도 이용된다. 데이터베이스 로드 작업은 독립 모드에서 **DBA** 권한으로 수행된다.
- *options* : 단축 옵션은 -와 함께 사용하고, 전체 옵션은 --와 함께 사용해야 한다. 대소문자는 구별해서 사용해야 한다.
- *database_name* : 새로 생성될 데이터베이스의 이름을 지정한다.

리턴 값

cubrid loaddb 유틸리티의 리턴 값은 다음과 같다.

- 0 : 성공
- Non-zero : 실패

입력 파일

- 스키마 파일(*database-name_schema*): 엔로드 작업에 의해 생성된 파일로서, 데이터베이스에 정의된 스키마 정보를 포함하는 파일이다.
- 객체 파일(*database-name_objects*) : 엔로드 작업에 의해 생성된 파일로서, 데이터베이스에 포함된 레코드 정보를 포함하는 파일이다.
- 인덱스 파일(*database-name_indexes*) : 엔로드 작업에 의해 생성된 파일로서, 데이터베이스에 정의된 인덱스 정보를 포함하는 파일이다.
- 트리거 파일(*database-name_trigger*) : 엔로드 작업에 의해 생성된 파일로서, 데이터베이스에 정의된 트리거 정보를 포함하는 파일이다.
- 사용자 정의 객체 파일(*user_defined_object_file*) : 대용량 데이터 입력을 위해 사용자가 테이블 형식으로 작성한 입력 파일이다.

옵션

다음은 **cubrid loaddb** 유틸리티와 결합할 수 있는 옵션을 정리한 표이다. 대소문자를 구별해서 사용해야 한다.

옵션	설명
-u	데이터베이스 사용자의 계정을 입력한다. 디폴트 값은 PUBLIC 이다.
--user	

-p --password	데이터베이스 사용자의 암호를 입력한다.
-l --load-only	객체 파일에 포함된 구문과 데이터 타입 검사를 생략하고 레코드를 로드한다.
-v --verbose	데이터 로딩 상태에 관한 상세 정보를 화면에 출력한다.
-c --periodic-commit	지정된 개수의 레코드가 입력될 때마다 트랜잭션을 커밋한다.
-s --schema-file	언로드 작업에 의해 생성된 스키마 파일을 지정하여, 스키마 로딩을 수행한다.
-i --index-file	언로드 작업에 의해 생성된 인덱스 파일을 지정하여, 인덱스 로딩을 수행한다.
-d --data-file	언로드 작업에 의해 생성된 데이터 파일을 지정하여, 레코드 로딩을 수행한다.
--data-file-check-only	데이터 파일에 포함된 구문과 데이터 타입 검사만 수행하고, 레코드 로딩은 수행하지 않는다.
--estimated-size	예상되는 레코드 수를 지정한다.
--no-oid	데이터 파일에 포함된 OID 참조 관계를 무시하고 레코드를 로딩한다.
--no-statistics	데이터베이스에 관한 통계 정보를 갱신하지 않는다.
--ignore-class-file	지정된 파일에 포함된 클래스를 제외하고 로딩 작업을 수행한다.
--error-control-file	데이터 로딩 중에 발생하는 특정 에러의 처리 방식을 명시한 파일을 지정한다.

사용자 계정 입력(-u 또는 --user)

-u는 레코드를 로딩할 데이터베이스의 사용자 계정을 지정하는 옵션이다. 옵션을 지정하지 않으면 디폴트 값인 **PUBLIC**이 된다.

```
cubrid loaddb -u admin -d demodb_objects newdb
```

암호 입력(-p 또는 --password)

-p는 레코드를 로딩할 데이터베이스의 사용자 암호를 지정하는 옵션이다. 옵션을 지정하지 않으면 암호 입력을 요청하는 프롬프트가 출력된다.

```
cubrid loaddb -p admin -d demodb_objects newdb
```

구문을 확인하지 않고 레코드 로딩(-l 또는 --load-only)

-l은 로딩할 데이터의 구문을 확인하지 않고 곧바로 데이터를 로딩하는 옵션이다. 다음은 demodb_objects에 포함된 데이터를 newdb로 로딩하는 구문이다.

-l 옵션을 사용하면 demodb_objects에 포함된 데이터의 구문을 확인하지 않고 곧바로 데이터를 로딩하기 때문에 속도는 빠르지만, 오류가 발생할 수도 있다.

```
cubrid loaddb -l -d demodb_objects newdb
```

로딩 상태 정보 출력(-v 또는 --verbose)

데이터베이스 로딩 작업이 진행되는 동안, 로딩되는 데이터베이스의 테이블 및 레코드에 관한 상세 정보를 화면에 출력하는 구문이다. -v 옵션을 이용하여 진행되는 단계, 로딩되는 클래스, 입력된 레코드의 개수와 같은 상세 정보를 확인할 수 있다.

```
cubrid loaddb -v -d demodb_objects newdb
```

커밋 주기 설정(-c 또는 --periodic-commit)

-c 옵션을 이용하여 100개의 레코드가 newdb로 입력될 때마다 커밋을 주기적으로 실행하는 명령이다. 만약, -c 옵션을 지정하지 않으면 demodb_objects에 포함된 모든 레코드가 newdb로 로딩된 후에 트랜잭션이 커밋된다. 또한, -c 옵션이 -s 옵션이나 -i 옵션과 함께 사용하는 경우에는 100개의 DDL문이 로딩될 때마다 커밋을 주기적으로 실행한다.

권장되는 커밋 주기는 로딩되는 데이터에 따라 다른데, 스키마 로딩의 경우에는 -c의 인수를 50으로 설정하고, 레코드로딩의 경우에는 1,000으로 설정하며, 인덱스 로딩의 경우에는 1로 설정하는 것이 바람직하다.

```
cubrid loaddb -c 100 -d demodb_objects newdb
```

스키마 로딩(-s 또는 --schema-file)

demodb에 정의된 스키마 정보를 새로 생성한 newdb에 로딩하는 구문이다. demodb_schema 파일은 언로드 작업에 의해 생성된 파일이며, 언로드된 데이터베이스의 스키마 정보를 포함한다. -s 옵션을 이용하여 스키마 정보를 먼저 로딩한 후, 실제 레코드를 로딩할 수 있다.

```
cubrid loaddb -u dba -s demodb_schema newdb

Start schema loading.
Total      86 statements executed.
Schema loading from demodb schema finished.
Statistics for Catalog classes have been updated.
```

인덱스 로딩(-i 또는 --index-file)

demodb에 정의된 인덱스 정보를 새로 생성한 newdb에 로딩하는 명령이다. demo_indexes 파일은 언로드 작업에 의해 생성된 파일이며, 언로드된 데이터베이스의 인덱스 정보를 포함한다. -d 옵션을 이용하여 레코드를 로딩한 후, -i 옵션을 이용하여 인덱스를 생성할 수 있다.

```
cubrid loaddb -u dba -i demodb_indexes newdb
```

데이터 로딩(-d 또는 --data-file)

-d 옵션을 이용하여 데이터 파일 또는 사용자 정의 객체 파일을 지정함으로써 레코드 정보를 newdb로 로딩하는 명령이다. demodb_objects 파일은 언로드 작업에 의해 생성된 객체 파일이거나, 사용자가 대량의 데이터 로딩을 위하여 작성한 사용자 정의 객체 파일 중 하나이다.

```
cubrid loaddb -u dba -d demodb_objects newdb
```

로딩될 데이터의 구문 체크만 수행(--data-file-check-only)

--data-file-check-only 옵션을 이용하여 demodb_objects에 포함된 데이터의 구문을 확인만 하는 명령이다. 따라서, 위 명령을 수행하여도 newdb에는 레코드가 로딩되지 않는다.

```
cubrid loaddb --data-file-check-only -d demodb_objects newdb
```

예상되는 레코드 수(--estimated-size)

--estimated-size 옵션은 언로드할 레코드의 수가 디폴트 값인 5,000개보다 많은 경우 로딩 성능 향상을 위해 사용할 수 있는 옵션이다. 즉, 이 옵션을 통해 레코드 저장을 위한 해시 메모리를 크게 할당함으로써 로드 성능을 향상시킬 수 있다.

```
cubrid loaddb --estimated-size 8000 -d demodb_objects newdb
```

참조 관계 무시하고 레코드 로딩(--no-oid)

demodb_objects에 포함된 OID를 무시하고 레코드를 newdb로 로딩하는 명령이다.

```
cubrid loaddb --no-oid -d demodb_objects newdb
```

통계 정보를 갱신하지 않고 레코드 로딩(--no-statistics)

demodb_objects를 로딩한 후 newdb의 통계 정보를 갱신하지 않는 명령이다. 특히, 대상 데이터베이스의 데이터 용량에 비해 매우 적은 데이터만 로딩할 경우 이 옵션을 이용하여 로드 성능을 향상시킬 수 있다.

```
cubrid loaddb --no-statistics -d demodb_objects newdb
```

제외 대상 클래스 지정(--ignore-class-file)

로딩 작업 중 무시할 클래스 목록을 명시한 파일을 지정하는 옵션이다. 지정된 파일에 포함된 클래스를 제외한 나머지 클래스의 레코드만 로딩된다.

```
cubrid loaddb --ignore-class-file=skip_class_list -d demodb_objects newdb
```

에러 정보 파일 지정(--error-control-file)

데이터베이스 로드 작업 중에 발생하는 에러 중 특정 에러를 처리하는 방식에 관해 명시한 파일을 지정하는 옵션이다.

```
cubrid loaddb --error-control-file=error_test -d demodb_objects newdb
```

가져오기용 파일 작성 방법

cubrid loaddb 유틸리티에서 사용되는 객체 입력 파일을 직접 작성하여 사용하면 데이터베이스에 대량의 데이터를 보다 신속하게 추가할 수 있다. 객체 입력 파일은 간단한 테이블 모양의 형식으로 구성되며 주석, 명령 라인, 데이터 라인으로 이루어진 텍스트 파일이다.

주석

CUBRID에서는 주석은 두 개의 연속된 하이픈(--)을 이용하여 처리한다.

```
-- This is a comment!
```

명령 라인

명령 라인은 퍼센트(%) 문자로 시작하며, 명령어로는 클래스를 정의하는 **%class** 명령어와, 클래스 식별을 위해 사용하는 별칭(alias)이나 식별자(identifier)를 정의하는 **%id** 명령어가 있다.

클래스에 식별자 부여

%id를 이용하여 참조 관계에 있는 클래스에 식별자를 부여할 수 있다.

구문

```
%id class_name class_id
class name:
    identifier
class id:
    integer
```

%id 명령어에 의해 명시된 *class_name*은 해당 데이터베이스에 정의된 클래스 이름이며, *class_id*는 객체 참조를 위해 부여한 숫자형 식별자를 의미한다.

예제 1

```
%id employee 2
%id office 22
%id project 23
%id phone 24
```

클래스 및 속성 명시

%class 명령어를 이용하여 데이터가 로딩될 클래스(테이블) 및 속성(컬럼)을 명시하며, 명시된 속성의 순서에 따라 데이터 라인이 작성되어야 한다.

구문

```
%class class_name ( attr_name [ { attr_name }_ ]
```

데이터를 로딩하고자 하는 데이터베이스에는 이미 스키마가 정의되어 있어야 한다.

%class 명령어에 의해 명시된 *class_name*은 해당 데이터베이스에 정의된 클래스 이름이며, *attr_name*은 정의된 속성 이름을 의미한다.

예제 2

다음은 employee라는 클래스에 데이터를 입력하기 위하여 **%class** 명령으로 클래스 및 3개의 속성을 명시한 예제이다. **%class** 명령 다음에 나오는 데이터 라인에서는 3개의 데이터가 입력되어야 하며, 이는 "참조 관계 설정하기"의 [예제 3](#)을 참조한다.

```
%class employee (name age department)
```

데이터 라인

데이터 라인은 **%class** 명령 라인 다음에 위치하며, 입력되는 데이터는 **%class** 명령에 의해 명시된 클래스 속성과 타입이 일치해야 한다. 만약, 명시된 속성과 타입이 일치하지 않으면 데이터 로드 작업은 중지된다.

또한, 각각의 속성에 대응되는 데이터는 적어도 하나의 공백에 의해 분리되어야 하며, 1라인에 작성되는 것이 원칙이다. 다만, 입력되는 데이터가 많은 경우에는 첫 번째 데이터 라인의 맨 마지막 데이터 다음에 플러스 기호(+)를 명시하여 다음 라인에 데이터를 연속적으로 입력할 수 있다. 이 때, 맨 마지막 데이터와 플러스 기호 사이에는 공백이 허용되지 않음을 유의한다.

인스턴스 입력

다음과 같이 명시된 클래스 속성과 타입이 일치하는 인스턴스를 입력할 수 있다. 각각의 데이터는 적어도 하나의 공백에 의해 구분된다.

예제 1

```
%class employee (name)
'jordan'
'james'
'garnett'
'malone'
```

인스턴스 번호 부여

데이터 라인의 처음에 '번호:'의 형식으로 해당 인스턴스에 대한 번호를 부여할 수 있다. 인스턴스 번호는 명시된 클래스 내에서 유일한 양수이며, 번호와 콜론(:) 사이에는 공백이 허용되지 않는다. 이와 같이 인스턴스 번호를 부여하는 이유는 추후 객체 참조 관계를 설정하기 위함이다.

예제 2

```
%class employee (name)
1: 'jordan'
2: 'james'
3: 'garnett'
4: 'malone'
```

참조 관계 설정

@ 다음에 참조하는 클래스를 명시하고, 수직바(|) 다음에 참조하는 인스턴스의 번호를 명시하여 객체 참조 관계를 설정할 수 있다.

구문

```
@class ref | instance no
class ref:
  class name
  class id
```

@ 다음에는 클래스명 또는 클래스 id를 명시하고, 수직바(|) 다음에는 인스턴스 번호를 명시한다. 수직바(|)의 양쪽에는 공백을 허용하지 않는다.

예제 3

다음은 paycheck 클래스에 인스턴스를 입력하는 예제이며, name 속성은 employee 클래스의 인스턴스를 참조한다. 마지막 라인과 같이 앞에서 정의되지 아니한 인스턴스 번호를 이용하여 참조 관계를 설정하는 경우 해당 데이터는 **NULL**로 입력된다.

```
%class paycheck(name department salary)
@employee|1 'planning' 8000000
@employee|2 'planning' 6000000
@employee|3 'sales' 5000000
@employee|4 'development' 4000000
@employee|5 'development' 5000000
```

예제 4

[클래스에 식별자 부여](#)에서 %id 명령어로 employee 클래스에 21이라는 식별자를 부여했으므로, 예제 3을 다음과 같이 작성할 수 있다.

```
%class paycheck(name department salary)
@21|1 'planning' 8000000
@21|2 'planning' 6000000
@21|3 'sales' 5000000
@21|4 'development' 4000000
@21|5 'development' 5000000
```

데이터베이스 백업 및 복구

데이터베이스 백업

설명

데이터베이스 백업은 CUBRID 데이터베이스 볼륨, 제어 파일, 로그 파일을 저장하는 작업으로 **cubrid backupdb** 유틸리티 또는 CUBRID 매니저를 이용하여 수행된다. **DBA**는 저장 매체의 오류 또는 파일 오류 등의 장애에 대비하여 데이터베이스를 정상적으로 복구할 수 있도록 주기적으로 데이터베이스를 백업해야 한다. 이 때 복구 환경은 백업 환경과 동일한 운영체제 및 동일한 버전의 CUBRID가 설치된 환경이어야 한다. 이러한 이유로 데이터베이스를 새로운 버전으로 마이그레이션한 후에는 즉시 새로운 버전의 환경에서 백업을 수행해야 한다.

cubrid backupdb 유틸리티는 모든 데이터베이스 페이지들, 제어 파일들, 데이터베이스를 백업 시와 일치된 상태로 복구하기 위해 필요한 로그 레코드들을 복사한다.

구문

```
cubrid backupdb [ options ] database name
[ options ]
-D | -r | -l | -o | -S | -C | -t | -z | -e |
--destination-path | --remove-archive | --level | --output-file | --SA-mode | --CS-mode |
--thread-count | --compress | --except-active-log | --no-check
```

- **cubrid** : CUBRID 서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **backupdb** : 지정한 데이터베이스를 백업하는 명령이며, 결합되는 옵션에 따라 온라인 백업, 오프라인 백업, 증분 백업, 압축 백업, 병렬 백업을 수행할 수 있다. 또한, 백업 권한을 가진 사용자(예: **DBA**)만 수행할 수 있다.
- **options** : 단축 옵션은 대시(-)와 결합하여 지정되어야 하고, 전체 옵션은 이중 대시(--)와 결합하여 지정되어야 한다. 대소문자 사용에 주의한다.
- **database_name** : 백업할 데이터베이스의 이름을 지정한다.

리턴 값

- 0 : 성공
- Non-zero : 실패

옵션

다음은 **cubrid backupdb** 유틸리티와 결합할 수 있는 옵션을 정리한 표이다. 대소문자가 구분됨을 주의한다.

옵션	설명
-D	백업 볼륨을 생성하고자 하는 디렉터리 경로명이나 장치 이름을
--destination-path	지정한다.

	기본값은 데이터베이스가 생성되었을 때, 데이터베이스 위치 정보 파일(databases.txt)에 지정된 log_path 위치
-r --remove-archive	백업 수행한 후, 복구에 필요하지 않은 보관 로그를 삭제한다.
-l --level	0, 1, 2 중에서 백업 수준을 설정한다. 기본값은 전체 백업(0)이다.
-o --output-file	진행 정보를 출력할 파일 이름을 지정한다.
-S --SA-mode	독립 모드에서 백업 수행한다. 기본값은 시스템 파라미터 CUBRID_MODE 의 설정값이다.
-C --CS-mode	클라이언트/서버 모드에서 백업 수행한다. 기본값은 시스템 파라미터 CUBRID_MODE 의 설정값이다.
-t --thread-count	병렬 백업 시 허용되는 스레드의 최대 개수를 지정한다. 기본값은 시스템의 CPU 개수이다.
-z --compress	압축 백업을 수행한다.
-e --except-active-log	백업 시 활성 로그 볼륨을 포함하지 않도록 설정한다.
--no-check	백업 전에 데이터베이스 일관성 점검을 수행하지 않는다.
-sp --safe-page-id	복제를 구성하는 환경에서 -r 옵션과 함께 설정되는 옵션이다. -r 옵션과 -sp 옵션을 함께 사용해야만, 복제 수행 시 필요한 정보를 제외한 나머지 불필요한 보관 로그가 삭제된다.

백업 파일이 저장될 디렉터리를 지정하여 백업 수행(-D 또는 --destination-path)

-D 옵션을 이용하여 지정된 디렉터리에 백업 파일이 저장되도록 하는 명령이며, 현재 존재하는 디렉터리가 지정되어야 한다. 그렇지 않으면 지정된 이름의 백업 파일이 생성된다. **-D** 옵션이 지정되지 않으면 백업 파일은 해당 데이터베이스의 위치 정보를 저장하는 파일인 **databases.txt**에 명시된 디렉터리에 생성된다.

```
cubrid backupdb -D /home/cubrid/backup demodb
```

-D 옵션을 이용하여 현재 디렉터리에 백업 파일이 저장되도록 하는 구문이다. **-D** 옵션의 인수로 "."을 입력하면 현재 디렉터리가 지정된다.

```
cubrid backupdb -D . demodb
```

백업 후 보관 로그를 삭제(-r 또는 --remove-archive)

데이터베이스 관련 파라미터 중 **media_failure_support** 파라미터의 설정값이 1이면, 활성 로그(active log)가 꼭 찾을 때 활성 로그를 새로운 보관 로그 파일에 기록한다. 이때 백업을 수행하여 백업 볼륨이

생성되면, 백업 시점 이전의 보관 로그는 추후 복구 작업에 필요 없다. **-r** 옵션은 백업을 수행한 후에, 추후 복구 작업에 더 이상 사용되지 않을 보관 로그 파일을 제거하는 옵션이다.

-r 옵션은 백업 시점 이전의 불필요한 보관 로그만 제거하므로 복구 작업에는 영향을 끼치지 않지만, 관리자가 백업 시점 이후의 보관 로그까지 제거하는 경우 전체 복구가 불가능할 수도 있다. 따라서 보관 로그를 제거할 때에는 추후 복구 작업에 필요한 것인지 반드시 검토해야 한다.

-r 옵션을 사용하여 증분 백업(백업 수준 1 또는 2)을 수행하는 경우, 추후 데이터베이스의 정상 복구가 불가능할 수도 있으므로 **-r** 옵션은 전체 백업 수행 시에만 사용하는 것을 권장한다.

```
cubrid backupdb -r demodb
```

복제에 필요한 페이지 정보를 저장(-sp 또는 --safe-page-id)

-sp 옵션은 복제 환경에서 **-r** 옵션과 함께 사용되는 옵션이다. 이 옵션이 지정되어야, 복제를 수행한 마지막 로그 페이지 ID를 제외하고 이보다 작은 로그 페이지 ID를 가지는 보관 로그만 안전하게 삭제한다. 즉, 복제에 필요한 보관 로그를 제외하고 **-r** 옵션이 수행되는 것이다. 이 옵션의 인자로 복제 스크립트 중 하나인 'repl_safe_page DB_NAME'을 지정해야 한다.

```
cubrid backupdb -r -sp 'repl_safe_page demodb' demodb
```

백업 수준을 지정하여 증분 백업 수행(-l 또는 --level)

-l 옵션을 이용하여 지정된 백업 수준으로 증분 백업을 수행하는 구문이다. **-l** 옵션이 지정되지 않으면 전체 백업이 수행된다. 백업 수준에 대한 자세한 내용은 [증분 백업](#)을 참조한다.

```
cubrid backupdb -l 1 demodb
```

지정된 파일에 백업 진행 정보 저장(-o 또는 --output-file)

-o 옵션을 이용하여 대상 데이터베이스의 백업에 관한 진행 정보를 info_backup이라는 파일에 기록하는 구문이다.

```
cubrid backupdb -o info_backup demodb
```

다음은 info_backup 파일 내용의 예시로서, 스레드 개수, 압축 방법, 백업 시작 시간, 영구 볼륨의 개수, 백업 진행 정보, 백업 완료 시간 등의 정보를 확인할 수 있다.

```
[ Database(demodb) Full Backup start ]
- num-threads: 1
- compression method: NONE
- backup start time: Mon Jul 21 16:51:51 2008
- number of permanent volumes: 1
- backup progress status
-----
volume name          | # of pages | backup progress status | done
-----
demodb_vinf          |          1 | #####                 | done
demodb                |        25000 | #####                 | done
demodb_lginf         |          1 | #####                 | done
demodb_lgat          |        25000 | #####                 | done
-----
# backup end time: Mon Jul 21 16:51:53 2008
[Database(demodb) Full Backup end]
```

독립 모드에서 백업 수행 (-S 또는 --SA-mode)

-S 옵션에 의해 독립모드에서 백업을 수행하는 구문으로서, demodb를 오프라인 백업한다. -S 옵션이 지정되지 않으면 **CUBRID_MODE** 환경 변수에 의해 설정된 모드에서 백업이 수행된다.

```
cubrid backupdb -S demodb
```

클라이언트/서버 모드에서 백업 수행(-C 또는 --CS-mode)

-C 옵션에 의해 클라이언트/서버 모드에서 백업을 수행하는 구문으로서, demodb를 온라인 백업한다. -C 옵션이 지정되지 않으면 **CUBRID_MODE** 환경 변수에 의해 설정된 모드에서 백업이 수행된다.

```
cubrid backupdb -C demodb
```

병렬 백업(-t 또는 --thread-count)

-t 옵션을 이용하여 관리자가 임의로 스레드의 개수를 지정함으로써 병렬 백업을 수행하는 명령이다. -t 옵션의 인수를 지정하지 않더라도 시스템의 CPU 개수만큼 스레드를 자동 부여하여 병렬 백업을 수행한다.

```
cubrid backupdb -t 4 demodb
```

압축 백업(-z 또는 --compress)

-z 옵션을 이용하여 대상 데이터베이스를 압축하여 백업 파일에 저장하는 구문이다. -z 옵션을 사용하면, 백업 파일의 크기 및 백업 시간을 단축시킬 수 있다.

```
cubrid backupdb -z demodb
```

활성 로그 볼륨 비포함 활성화(-e 또는 --except-active-log)

-e 옵션을 이용하여 대상 데이터베이스의 활성 로그(active log)를 포함하지 않고 백업을 수행하는 구문이다. -e 옵션을 이용하면 활성 로그를 생성하지 않고 백업이 이루어지므로 백업 시간을 단축시킬 수 있으나, 백업 시점 이후 최근 시점까지의 데이터를 복구할 수 없으므로 상당한 주의를 요한다.

```
cubrid backupdb -e demodb
```

데이터베이스 일관성 점검 비활성화(--no-check)

--no-check 옵션을 이용하여 대상 데이터베이스의 일관성을 체크하지 않고, 바로 백업을 수행하는 명령이다.

```
cubrid backupdb --no-check demodb
```

백업 정책 및 방식

백업을 진행할 때 고려해야 할 사항은 다음과 같다.

- 백업할 대상 데이터 선별
 - 보존 가치가 있는 유효한 데이터인지 판단한다.
 - 데이터베이스 전체를 백업할 것인지, 일부만 백업할 것인지 결정한다.
 - 데이터베이스와 함께 백업해야 할 다른 파일이 있는지 확인한다.
- 백업 방식 결정

- 증분 백업, 온라인 백업 방식을 결정한다. 부가적으로 압축 백업, 병렬 백업 모드 사용 여부를 결정한다.
- 사용 가능한 백업 도구 및 백업 장비를 준비한다.

• 백업 시기 판단

- 데이터베이스 사용이 가장 적은 시간을 파악한다.
- 보관 로그의 양을 파악한다.
- 백업할 데이터베이스를 이용하는 클라이언트 수를 파악한다.

온라인 백업

온라인 백업(또는 핫 백업)은 운영 중인 데이터베이스에 대해 백업을 수행하는 방식으로, 특정 시점의 데이터베이스 이미지의 스냅샷을 제공한다. 운영 중인 데이터베이스를 대상으로 백업을 수행하기 때문에 커밋되지 않은 데이터가 저장될 우려가 있고, 다른 데이터베이스 운영에도 영향을 줄 수 있다.

온라인 백업을 하려면 **cubrid backupdb -C** 명령어를 사용한다.

오프라인 백업

오프라인 백업(또는 콜드 백업)은 정지 상태인 데이터베이스에 대해 백업을 수행하는 방식으로 특정 시점의 데이터베이스 이미지의 스냅샷을 제공한다.

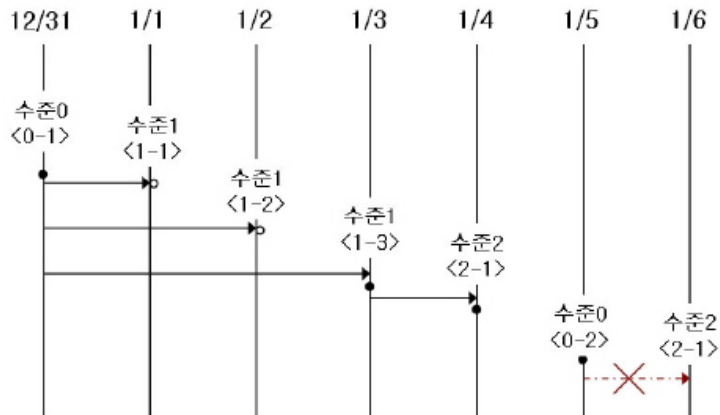
오프라인 백업을 하려면 **cubrid backupdb -S** 명령어를 사용한다.

증분 백업

증분 백업(incremental backup)은 전체 백업에 증속적으로 수행되는 백업으로 이전에 수행된 백업 이후의 변경된 사항만을 선택적으로 백업하는 방식이다. 이는 전체 백업보다 백업 볼륨이 적고, 백업 소요 시간이 짧다는 장점이 있다. CUBRID는 0, 1, 2의 백업 수준을 제공하며, 낮은 백업 수준으로 백업을 수행한 이후에만 순차적으로 다음 수준의 백업을 수행할 수 있다.

증분 백업을 하려면 **cubrid backupdb -I <level>** 명령어를 사용한다.

다음은 증분 백업에 관한 예시로서, 이를 참조하여 백업 수준에 관해 상세하게 살펴보기로 한다.



- **전체 백업(백업 수준 0)** : 백업 수준 0은 모든 데이터베이스 페이지를 포함하는 전체 백업이다.

데이터베이스에 최초 시도되는 백업 수준은 당연히 수준 0이 된다. DBA는 복구 상황을 대비하여 정기적으로 전체 백업을 수행해야 하며, 예시에서는 12월 31일과 1월 5일에 전체 백업을 수행하였다.

- **1차 증분 백업(백업 수준 1)** : 백업 수준 1은 수준 0의 전체 백업 이후의 변경 사항만 저장하는 증분 백업으로서, 이를 "1차 증분 백업"이라 한다.

주의할 점은 예시의 <1-1>, <1-2>, <1-3>과 같이 1차 증분 백업이 연속적으로 시도되더라도 언제나 수준 0의 전체 백업을 기본으로 증분 백업을 수행한다는 점이다.

만약, 동일 디렉터리에서 백업 파일이 생성된다고 할 때, 1월 1일에 이미 1차 증분 백업 <1-1>이 수행되고, 1월 2일에 또다시 1차 증분 백업 <1-2>가 시도되면, <1-1>에서 생성된 증분 백업 파일을 덮어쓰게 된다. 1월 3일에 1차 증분 백업이 다시 수행되었으므로, 최종 증분 파일은 이 때 생성된다.

그러나, 1월 1일이나 1월 2일의 상태로 데이터베이스를 복구해야 하는 상황이 발생할 수 있으므로, DBA는 최종 증분 파일로 덮어쓰기 전에 <1-1>과 <1-2> 각각의 증분 백업 파일을 저장 매체에 별도로 보관하는 것이 좋다.

- **2차 증분 백업(백업 수준 2)** : 백업 수준 2는 1차 증분 백업 이후의 변경 사항만 저장하는 증분 백업으로 이를 "2차 증분 백업"이라 한다.

1차 증분 백업이 선행되어야만 2차 증분 백업을 수행할 수 있으므로, 1월 4일에 시도한 2차 증분 백업 시도는 성공할 것이고, 1월 6일에 시도한 2차 증분 백업 시도는 당연히 허용되지 않을 것이다.

이러한 백업 수준 0, 1, 2로 생성된 백업 파일들은 모두 데이터베이스를 복구할 때 필요하므로, 2차 증분 백업이 완료된 1월 4일의 상태로 데이터베이스를 복구하기 위해서는 <2-1>에서 생성된 2차 증분 백업 파일, <1-3>에서 생성된 1차 증분 백업 파일, <0-1>에서 생성된 전체 백업 파일이 모두 필요하다. 즉, 완전한 복구를 위해서는 직전에 생성된 증분 백업 파일로부터 이전 최종으로 생성된 전체 백업 파일이 요구된다.

압축 백업 모드

압축 백업(compress backup)은 데이터베이스를 압축하여 백업을 수행하기 때문에 백업 볼륨의 크기가 줄어들어 디스크 I/O 비용을 감소시킬 수 있고, 디스크 공간을 절약할 수 있다.

압축 백업을 하려면 **cubrid backupdb -z--compress** 명령어를 사용한다.

병렬 백업 모드

병렬 백업 또는 다중 백업(multi-thread backup)은 지정된 스레드 개수만큼 동시 백업을 수행하기 때문에 백업 시간을 크게 단축시켜 준다. 기본적으로 시스템의 CPU 수만큼 스레드를 부여하게 된다.

병렬 백업을 하려면 **cubrid backupdb -t--thread-count** 명령어를 사용한다.

백업 파일 관리

백업 대상 데이터베이스의 크기에 따라 하나 이상의 백업 파일이 연속적으로 생성될 수 있으며, 각각의 백업 파일의 확장자에는 생성 순서에 따라 000, 001~0xx와 같은 유닛 번호가 순차적으로 부여된다.

백업 작업 중 디스크 용량 관리

백업 작업 도중, 백업 파일이 저장되는 디스크 용량에 여유가 없는 경우 백업 작업을 진행할 수 없다는 안내 메시지가 화면에 나타난다. 안내 메시지에는 백업 대상이 되는 데이터베이스의 이름과 경로명, 백업 파일명, 백업 파일의 유닛 번호, 백업 수준이 표시된다. 백업 작업을 계속 진행하려는 관리자는 다음과 같이 옵션을 선택할 수 있다.

- 옵션 0 : 백업 작업을 더이상 진행하지 않을 경우, 0을 입력한다.
- 옵션 1 : 백업 작업을 진행하기 위해 관리자는 현재 장치에 새로운 디스크를 삽입한 후 1을 입력한다.
- 옵션 2 : 백업 작업을 진행하기 위해 관리자는 장치를 변경하거나 백업 파일이 저장되는 디렉터리 경로를 변경한 후 2를 입력한다.

```
*****
Backup destination is full, a new destination is required to continue:
Database Name: /local/testing/demodb
Volume Name: /dev/rst1
Unit Num: 1
Backup Level: 0 (FULL LEVEL)
Enter one of the following options:
Type
- 0 to quit.
- 1 to continue after the volume is mounted/loaded. (retry)
- 2 to continue after changing the volume's directory or device.
*****
```

데이터베이스 복구

설명

데이터베이스 복구는 동일 버전의 CUBRID 환경에서 수행된 백업 작업에 의해 생성된 백업 파일, 활성 로그 및 보관 로그를 이용하여 특정 시점의 데이터베이스로 복구하는 작업이다. 데이터베이스 복구를 진행하려면 **cubrid restoredb** 유틸리티 또는 CUBRID 매니저를 사용한다.

cubrid restoredb 유틸리티는 백업이 수행된 이후에 모든 보관 및 활동 로그들에 기록된 정보들을 이용하여 데이터베이스 백업으로부터 데이터베이스를 복구한다.

구문

```
cubrid restoredb [ options ] database name
[ options ]
-d | -B | -l | -p | -o | -u |
--up-to-date | --backup-file-path | --level | --partial-recovery | --output-file | --
replication-mode | --use-database-location-path | --list
```

- **cubrid** : CUBRID서비스 및 데이터베이스 관리를 위한 통합 유틸리티이다.
- **restoredb** : 지정한 데이터베이스를 복구하는 명령이며, 정상적인 복구를 위해서는 백업 파일, 활성 로그 파일, 보관 로그 파일이 준비되어야 한다. 또한, 독립 모드(standalone)에서만 수행 가능하다.
- *options* : 단축 옵션은 대시(-)와 결합하여 지정되어야 하고, 전체 옵션은 이중 대시(--)와 결합하여 지정되어야 한다. 대소문자 사용에 주의한다.
- *database_name* : 복구하려는 데이터베이스의 이름을 지정한다.

리턴 값

- 0 : 성공
- Non-zero : 실패

옵션

다음은 **cubrid restoredb** 유틸리티와 결합할 수 있는 옵션을 정리한 표이다. 대소문자가 구분됨을 주의한다.

옵션	설명
-d --up-to-date	데이터베이스를 복구할 시간을 직접 지정하거나, backuptime 키워드를 지정한다.
-B --backup-file-path	백업 파일이 위치하는 디렉터리 경로명이나 장치명을 지정한다.
-l --level	0, 1, 2 중에서 복구할 수준을 설정한다. 디폴트 값은 전체 복구(0)이다.
-p --partial-recovery	부분 복구를 수행한다.
-o	복구 작업에 관한 진행 정보를 출력할 파일 이름을 지정한다.

--output-file	
-u	데이터베이스 위치 정보 파일(databases.txt)에 지정된 경로로
--use-database-location-path	대상 데이터베이스를 복구한다.
--list	대상 데이터베이스의 백업 볼륨에 관한 정보를 화면 출력한다.

복구 시점을 지정하여 복구 수행(-d 또는 --up-to-date)

demodb를 복구하는 명령이며, 어떠한 옵션도 지정되지 않은 경우 기본적으로 마지막 커밋 시점까지 demodb가 복구된다. 만약, 마지막 커밋 시점까지 복구하기 위해 필요한 활성 로그/보관 로그 파일이 없다면 마지막 백업 시점까지만 부분 복구된다.

```
cubrid restoredb demodb
```

-d 옵션을 이용하여 복구 날짜 및 시간을 지정하는 구문으로 demodb를 해당 시점까지 복구한다. 사용자는 dd-mm-yyyy:hh:mi:ss(예: 14-10-2008:14:10:00)의 형식으로 복구 시점을 직접 지정할 수 있다. 만약 지정된 복구 시점까지 복구하기 위해 필요한 활성 로그/보관 로그 파일이 없다면 마지막 백업 시점까지만 부분 복구된다.

```
cubrid restoredb -d 14-10-2008:14:10:00 demodb
```

-d 옵션 및 **backuptime**이라는 키워드를 이용하여 복구 시점을 지정하는 구문으로 demodb를 마지막 백업이 수행된 시점까지 복구한다.

```
cubrid restoredb -d backuptime demodb
```

백업 파일이 위치하는 디렉터리 경로를 지정하여 복구 수행(-B 또는 --backup-file-path)

-B 옵션을 이용하여 백업 파일이 위치하는 디렉터리를 지정할 수 있다. 만약, 이 옵션이 지정되지 않으면 시스템은 데이터베이스 위치 정보 파일인 **databases.txt**에 지정된 **log_path** 디렉터리에서 대상 데이터베이스를 백업했을 때 생성된 백업 정보 파일(**dbname_bkvinf**)을 검색하고, 백업 정보 파일에 지정된 디렉터리 경로에서 백업 파일을 찾는다. 그러나, 백업 정보 파일이 손상되거나 백업 파일의 위치 정보가 삭제된 경우라면 시스템이 백업 파일을 찾을 수 없으므로, 관리자가 **-B** 옵션을 이용하여 백업 파일이 위치하는 디렉터리 경로를 직접 지정해야 한다.

```
cubrid restoredb -B /home/cubrid/backup demodb
```

demodb의 백업 파일이 현재 디렉터리에 있는 경우, 관리자는 **-B** 옵션을 이용하여 백업 파일이 위치하는 디렉터리를 지정할 수 있다.

```
cubrid restoredb -B . demodb
```

백업 수준을 지정하여 복구 수행(-l 또는 --level)

-l 옵션은 대상 데이터베이스의 백업 수준(0, 1, 2)을 지정하여 복구를 수행한다. 백업 수준에 대한 자세한 내용은 [중분 백업](#)을 참조한다.

```
cubrid restoredb -l 1 demodb
```

부분 복구 수행(-p 또는 --partial-recovery)

-p 옵션은 사용자 응답을 요청하지 않고 부분 복구를 수행하라는 명령이다. 백업 시점 이후에 기록된 활성 로그나 보관 로그가 완전하지 않을 때, 기본적으로 시스템은 로그 파일이 필요하다는 것을 알리면서 실행 옵션을 입력하라는 요청 메시지를 출력하는데, -p 옵션을 이용하면 이러한 요청 메시지의 출력 없이 직접 부분 복구를 수행할 수 있다. 따라서, -p 옵션을 이용하여 복구를 수행하면 언제나 마지막 백업 시점까지 데이터가 복구된다.

```
cubrid restoredb -p demodb
```

-p 옵션이 지정되지 않은 경우, 사용자에게 실행 옵션을 선택하라는 요청 메시지는 다음과 같다.

```
*****
Log Archive /home/cubrid/test/log/demodb lgar002
is needed to continue normal execution.
Type
- 0 to quit.
- 1 to continue without present archive. (Partial recovery)
- 2 to continue after the archive is mounted/loaded.
- 3 to continue after changing location/name of archive.
*****
```

- 옵션 0 : 복구 작업을 더이상 진행하지 않을 경우, 0을 입력한다.
- 옵션 1 : 로그 파일 없이 부분 복구를 진행하려면, 1을 입력한다.
- 옵션 2 : 복구 작업을 진행하기 위해 관리자는 현재 장치에 보관 로그를 위치시킨 후 2를 입력한다.
- 옵션 3 : 복구 작업을 계속하기 위해 관리자는 로그 위치를 변경한 후 3을 입력한다.

지정된 파일에 복구 진행 정보 저장(-o 또는 --output-file)

-o 옵션을 이용하여 대상 데이터베이스의 복구에 관한 진행 정보를 info_restore라는 파일에 기록하는 명령이다.

```
cubrid restoredb -o info_restore demodb
```

데이터베이스 위치 정보 파일에 지정된 디렉터리로 데이터를 복구(-u 또는 --use-database-location-path)

-u 옵션을 이용하여 데이터베이스 위치 정보 파일(databases.txt)에 지정된 경로에서 대상 데이터베이스를 복구하는 구문이다. -u 옵션은 A 서버에서 백업을 수행하고 B 서버에서 백업 파일을 복구하고자 할 때 사용할 수 있는 유용한 옵션이다.

```
cubrid restoredb -u demodb
```

대상 데이터베이스의 백업 정보 확인하기(--list)

--list 옵션을 이용하여 대상 데이터베이스의 백업 파일에 관한 정보를 화면 출력하는 구문으로 복구는 수행하지 않는다.

```
cubrid restoredb --list demodb
```

다음은 --list 옵션에 의해 출력되는 백업 정보의 예로서, 복구 작업을 수행하기 이전에 대상 데이터베이스의 백업 파일이 최초 저장된 경로와 백업 수준을 검증할 수 있다.

```
*** BACKUP HEADER INFORMATION ***
Database Name: /local1/testing/demodb
```

```

DB Creation Time: Mon Oct 1 17:27:40 2008
Pagesize: 4096
Backup Level: 1 (INCREMENTAL LEVEL 1)
  Start lsa: 513|3688
  Last lsa: 513|3688
Backup Time: Mon Oct 1 17:32:50 2008
Backup Unit Num: 0
Release: 8.1.0
  Disk Version: 8
Backup Pagesize: 4096
Zip Method: 0 (NONE)
  Zip Level: 0 (NONE)
Previous Backup level: 0 Time: Mon Oct 1 17:31:40 2008
(start lsa was -1|-1)
Database Volume name: /local1/testing/demodb_vinf
  Volume Identifier: -5, Size: 308 bytes (1 pages)
Database Volume name: /local1/testing/demodb
  Volume Identifier: 0, Size: 2048000 bytes (500 pages)
Database Volume name: /local1/testing/demodb_lginf
  Volume Identifier: -4, Size: 165 bytes (1 pages)
Database Volume name: /local1/testing/demodb_bkvinf
  Volume Identifier: -3, Size: 132 bytes (1 pages)

```

--list 옵션을 이용하여 출력된 백업 정보를 확인하면, 백업 파일이 백업 수준 1로 생성되었고, 이전 백업 수준 0의 전체 백업이 수행된 시점을 확인할 수 있다. 따라서, 예시된 데이터베이스의 복구를 위해서는 백업 수준 0인 백업 파일과 백업 수준 1인 백업 파일이 준비되어야 한다.

복구 정책과 절차

데이터베이스를 복구할 때 고려해야 할 사항은 다음과 같다.

- **백업 파일 준비**
 - 백업 파일 및 로그 파일이 저장된 디렉터리를 파악한다.
 - 증분 백업으로 대상 데이터베이스가 백업된 경우, 각 백업 수준에 따른 백업 파일이 존재하는지를 파악한다.
 - 백업이 수행된 CUBRID 데이터베이스의 버전과 복구가 이루어질 CUBRID 데이터베이스 버전이 동일한지를 파악한다.
- **복구 방식 결정**
 - 부분 복구인지 전체 복구인지를 결정한다.
 - 증분 백업 파일을 이용한 복구인지를 결정한다.
 - 사용 가능한 복구 도구 및 복구 장비를 준비한다.
- **복구 시점 판단**
 - 데이터베이스 서버가 종료된 시점을 파악한다.
 - 장애 발생 전에 이루어진 마지막 백업 시점을 파악한다.
 - 장애 발생 전에 이루어진 마지막 커밋 시점을 파악한다.

데이터베이스 복구 절차

다음은 백업 및 복구 작업의 절차를 시간별로 예시한 것이다.

- 2008/8/14 04:30분에 운영이 중단된 demodb를 전체 백업을 수행한다.

- 2008/8/14 10:00분에 운영 중인 demodb를 1차 증분 백업 수행한다.
- 2008/8/14 15:00분에 운영 중인 demodb를 1차 증분 백업을 수행한다. 2번의 1차 증분 백업 파일을 덮어쓴다.
- 2008/8/14 15:30분에 시스템 장애가 발생하였고, 관리자는 demodb의 복구 작업을 준비한다. 장애 발생 이전의 마지막 커밋 시점이 15:25분이므로 이를 복구 시점으로 지정한다.
- 관리자는 1.에서 생성된 전체 백업 파일 및 3.에서 생성된 1차 증분 백업 파일, 활성화 로그 및 보관 로그를 준비하여 마지막 커밋 시점인 15:25 시점까지 demodb를 복구한다.

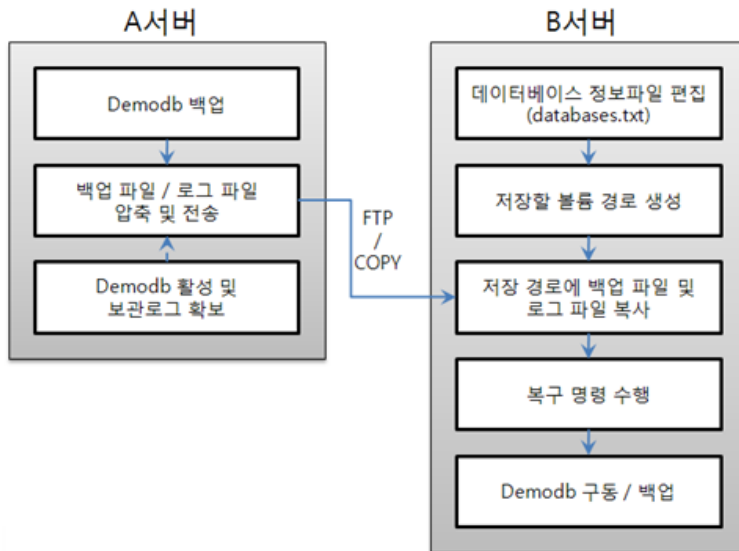
Time	Command	설명
2008/8/14 04:25	cubrid server stop demodb	demodb 운영을 중단한다.
2008/8/14 04:30	cubrid backupdb -S -D /home/backup -l 0 demodb	오프라인에서 demodb 를 전체 백업하여 지정된 디렉터리에 백업 파일을 생성한다.
2008/8/14 05:00	cubrid server start demodb	demodb 운영을 시작한다.
2008/8/14 10:00	cubrid backupdb -C -D /home/backup -l 1 demodb	온라인에서 demodb 를 1 차 증분 백업하여 지정된 디렉터리에 백업 파일을 생성한다.
2008/8/14 15:00	cubrid backupdb -C -D /home/backup -l 1 demodb	온라인에서 demodb 를 1 차 증분 백업하여 지정된 디렉터리에 백업 파일을 생성한다. 10:00에 생성된 1 차 증분 백업파일을 덮어쓴다.
2008/8/14 15:30		시스템 장애가 발생한 시각이다.
2008/8/14 15:40	cubrid restoredb -l 1 -d 08/14/2008:15:25:00 demodb	전체 백업 파일, 1 차 증분 백업 파일, 활성화 로그 및 보관 로그를 기반으로 demodb 를 복구한다. 전체 백업 파일, 1 차 증분된 백업 파일, 활성화 로그 및 보관 로그에 의해 15:25 시점까지 복구된다.

다른 서버로의 데이터베이스 복구

다음은 A 서버에서 demodb를 백업하고, 백업된 파일을 기반으로 B 서버에서 demodb를 복구하는 방법이다.

백업 환경과 복구 환경

A 서버의 /home/cubrid/db/demodb 디렉터리에서 demodb를 백업하고, B 서버의 /home/cubrid/data/demodb 디렉터리에 demodb를 복구하는 것으로 가정한다.



1. A 서버에서 백업

A 서버에서 demodb를 백업한다. 이전에 백업을 수행하였다면 이후 변경된 부분만 증분 백업을 수행할 수 있다. 백업 파일이 생성되는 디렉터리는 **-D** 옵션에 의해 지정하지 않으면, 기본적으로 로그 볼륨이 저장되는 위치에 생성된다. 다음은 권장되는 옵션을 결합한 백업 명령이며, 옵션에 관한 보다 자세한 내용은 [데이터베이스 백업](#)을 참조한다.

```
cubrid backupdb -z -t demodb
```

2. B 서버에서 데이터베이스 위치 정보 파일 편집

동일한 서버에서 백업 및 복구 작업이 이루어지는 일반적인 시나리오와는 달리, 타 서버 환경에서 백업 파일을 복구하는 시나리오에서는 B 서버의 데이터베이스 위치 정보 파일(**databases.txt**)에서 데이터베이스를 복구할 위치 정보를 추가해야 한다. 위 그림에서는 B 서버(호스트명은 pmlinux)의 `/home/cubrid/data/demodb` 디렉터리에 demodb를 복구하는 것을 가정하였으므로, 이에 따라 데이터베이스 위치 정보 파일을 편집하고, 해당 디렉터리를 B 서버에서 생성한다.

데이터베이스 위치 정보는 한 라인으로 작성하고, 각 항목은 공백으로 구분한다. 한 라인은 [데이터베이스명] [데이터볼륨경로] [호스트명] [로그볼륨경로]의 형식으로 작성한다. 따라서 다음과 같이 demodb의 위치 정보를 작성한다.

```
demodb /home/cubrid/data/demodb pmlinux /home/cubrid/data/demodb
```

3. B 서버로 백업 파일 및 로그 파일 전송

복구를 위해서는 대상 데이터베이스의 백업 파일(예: demodb_bk0v000) 및 백업 정보 파일(예: demodb_bkvinf)이 필수적으로 준비되어야 하고, 마지막 커밋 시점까지 전체 데이터를 복구하기 위해서는 활성 로그(예: demodb_lgat) 및 보관 로그(예: demodb_lgar000)가 준비되어야 한다. 따라서, A 서버에서 생성된 백업 파일, 백업 정보 파일, 활성 로그 파일, 보관 로그 파일을 B 서버에 전송한다. 즉, B 서버의 임의 디렉터리(예: `/home/cubrid/temp`)에는 백업 파일, 백업 정보 파일, 활성 로그 파일, 보관 로그 파일이 위치해야 한다.

4. B 서버에서 복구

B 서버로 전송한 백업 파일, 백업 정보 파일, 활성 로그 파일, 보관 로그 파일이 있는 디렉터리에서 **cubrid restoredb** 유틸리티를 호출하여 데이터베이스 복구 작업을 수행한다. **-u** 옵션에 의해 **databases.txt**에 지정된 디렉터리 경로에 demodb가 복구된다.

```
cubrid restoredb -u demodb
```

만약, 다른 위치에서 **cubrid restoredb** 유틸리티를 호출하려면, 다음과 같이 **-B** 옵션을 이용하여 백업 파일이 위치하는 디렉터리 경로를 지정해야 한다.

```
cubrid restoredb -u -B /home/cubrid/temp demodb
```

5. B 서버에서 복구한 데이터베이스를 다시 백업

대상 데이터베이스의 복구가 완료되면, 해당 데이터베이스를 구동하여 정상적으로 복구되었는지를 확인한다. 또한, 복구한 데이터베이스를 안정적으로 관리하기 위해서는 B 서버 환경에서 대상 데이터베이스를 새로 백업하는 것이 좋다.

데이터베이스 복제

복제 개념

개요

데이터베이스 복제는 하나의 데이터베이스에 저장된 객체를 물리적으로 분리된 다른 데이터베이스에 복사하여 두 개 이상의 데이터베이스 서버에서 사용될 수 있도록 하는 분산 데이터베이스 기술 중 하나이다. 자세한 내용은 [복제 제약 사항](#)과 [복제 주의 사항](#)을 참고한다. 복제 기술은 다음과 같은 용도로 사용할 수 있다.

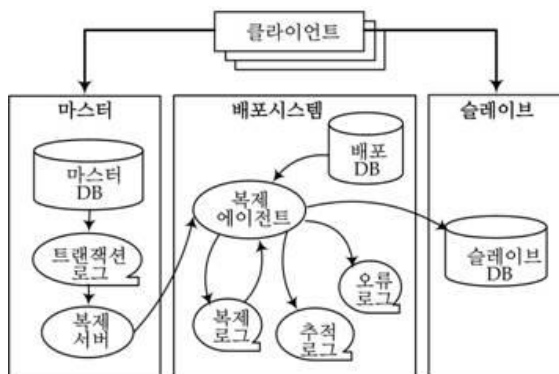
- 같은 객체를 이용하는 응용의 접근을 여러 데이터베이스 서버에 분산시킴으로써 성능을 높일 수 있다.
- 복제된 데이터베이스 서버를 다른 용도로 사용할 수 있도록 하여 서로 다른 운영 요구 사항을 만족시킬 수 있다.
- 운영 중인 데이터베이스 서버에 장애가 발생했을 경우 복제 데이터베이스 서버로 대체함으로써 장애 발생 시 긴급하게 대처할 수 있다.

주의 CUBRID 2008 R3.0 버전부터 복제 기능은 더 이상 개발되지 않는다. 복제 기능 대신 HA 기능을 사용하여 이중화 환경을 구성하기를 권장한다.

복제 시스템 구조와 용어

CUBRID 복제 시스템의 구성은 다음 그림과 같다.

복제 대상이 되는 원본 데이터베이스를 마스터 데이터베이스(master database)라 하고, 원본 데이터베이스가 복제되어 채워지는 데이터베이스를 슬레이브 데이터베이스(slave database)라 부른다. 마스터 데이터베이스는 일반 데이터베이스로서 읽기/쓰기 등 모든 연산이 허용되나, 슬레이브 데이터베이스는 읽기 연산만 허용한다.



- 복제가 설정되면, 마스터에 위치하는 복제 서버(replication server)는 마스터 데이터베이스의 트랜잭션 로그를 읽는다.
- 복제 서버는 트랜잭션 로그 중 복제와 관련된 정보만으로 복제 로그(replication log)를 만들어, 배포 시스템(distributor)에 위치하는 복제 에이전트(replication agent)에 전달한다. 배포 시스템은 마스터 호스트나 슬레이브 호스트에 설정될 수도 있으나, 가용성을 최대한 확보하기 위하여 별도의 호스트에 설정할 수도 있다.
- 복제 에이전트는 배포 데이터베이스(distributor database)를 읽어 들여, 마스터로부터 받은 복제 로그를 어느 슬레이브에 어떻게 복제할 것인지를 판단한다. 배포 데이터베이스는 CUBRID 데이터베이스의 하나로서, 복제 구성에 관한 모든 메타데이터를 관리한다.
- 복제 에이전트는 복제 서버에서 받은 복제 로그를 일단 디스크에 저장한다.
- 배포 데이터베이스에 지정된 슬레이브 데이터베이스에 복제 로그를 배포하여, 슬레이브 데이터베이스에 마스터 데이터베이스의 변경 사항을 반영하게 된다. 이 과정에서 각 슬레이브 데이터베이스에 반영한 상태를 추적하기 위하여 추적 로그(trail log)를 관리하고, 배포 중 발생한 모든 오류를 오류 로그(error log)에 저장한다.

CUBRID 복제 시스템을 구성하는 각 요소를 정리하면 다음과 같다.

마스터 데이터베이스

일반적인 CUBRID 데이터베이스로서, 복제의 대상이 되는 원본 데이터베이스라 할 수 있다. CUBRID 복제는 비동기식(asynchronous)이기 때문에 마스터 데이터베이스의 운영은 복제 시스템 설정에 영향을 받지 않는다(비동기식, 동기식 복제에 대한 더 자세한 설명은 차후에 설명한다). 슬레이브 데이터베이스와 달리 마스터 데이터베이스는 읽기, 쓰기 등 모든 데이터베이스 연산이 허용된다.

트랜잭션 로그

트랜잭션 로그는 복제 시스템과 무관하게 마스터 데이터베이스의 트랜잭션 무결성과 장애 발생시 데이터베이스를 복구하여 가용성을 보장하는 데 사용한다. 복제 시스템은 마스터 데이터베이스에 발생한 모든 변경 정보를 트랜잭션 로그를 통하여 알아 낸다.

복제 서버

복제 서버는 복제가 설정되었을 때, 마스터 데이터베이스의 트랜잭션 로그를 분석하여 복제 배포를 위해 필요한 복제 로그를 생성하고, 이를 배포 서버 호스트의 복제 에이전트에 전송한다. CUBRID의 복제 서버는 **cubrid repl_server**라는 명령어로 지원된다. 이 명령어를 사용하는 상세한 방법은 이 장에서 차후에 설명한다.

배포 시스템

마스터의 복제 서버에 받은 복제 로그를 슬레이브에 복제하는 시스템으로서, 복제 에이전트, 배포 데이터베이스, 그리고 연관된 몇 가지 로그 파일로 구성된다. 배포 서버는 마스터 호스트 혹은 슬레이브

호스트에 설정될 수도 있고, 장애 발생시 가용성을 높이고, 성능을 높이기 위하여 별도의 호스트에 설정할 수도 있다.

복제 에이전트

복제 서버에서 전송된 복제 로그를 이용하여 실제로 복제 작업을 수행하는 프로세스이다. 하나의 마스터에 대하여 여러 개의 슬레이브가 설정될 수 있기 때문에 복제 에이전트는 먼저, 복제 서버에 받은 복제 로그를 파일에 저장한다. 이어서, 복제 에이전트는 배포 데이터베이스에 설정된 모든 슬레이브 데이터베이스에 복제 로그를 반영한다. CUBRID의 복제 에이전트는 **cubrid repl_agent**라는 명령어로 지원된다. 이 명령어를 사용하는 상세한 방법은 이 장에서 차후에 설명한다.

배포 데이터베이스

배포 시스템의 복제 에이전트로 하여금 마스터와 슬레이브 배치 정보를 알려주는 데이터베이스로서, CUBRID 데이터베이스의 하나로 관리된다. 배포 데이터베이스의 스키마를 비롯한 상세한 정보는 이 장에서 차후에 설명한다.

복제 로그

마스터의 복제 서버로부터 받은 로그를 파일에 저장한 것으로서, <master_db_name>.copy의 형식의 이름을 갖는다. 복제 로그의 크기는 마스터의 변경 양에 따라 달라질 수 있기 때문에, 복제를 설정할 때, 배포 시스템은 복제 로그를 저장할 수 있을 정도의 충분한 디스크 공간을 확보하고 있어야 한다.

추적 로그

추적 로그는 복제 에이전트가 실제로 복제를 수행하면서 각 슬레이브 데이터베이스 별로 최종 적용된 로그 순차 번호 등 배포의 진행 상태를 기록하는 파일이다. 추적 로그는 복제 에이전트마다 하나의 파일로 관리되는데, <dist_db_name>.trail의 형식의 이름을 갖는다

오류 로그

오류 로그는 복제 에이전트가 복제를 실행하면서 발생하는 모든 오류를 기록하는 파일로서, <dist_db_name>.err의 형식의 이름을 갖는다.

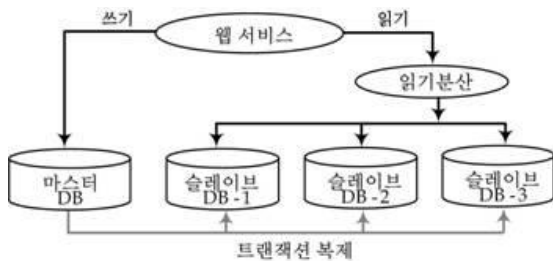
슬레이브 데이터베이스

복제의 목적지라 할 수 있는 데이터베이스로서, 마스터의 변경이 복제 시스템에 의하여 자동적으로 반영되는 데이터베이스이다. 마스터 데이터베이스와 달리, 슬레이브 데이터베이스는 읽기 연산에만 사용될 수 있다.

복제 사용 시나리오

부하 분산을 통한 성능 향상

복제는 하나의 데이터베이스 서버로 응용의 성능 요구 사항을 만족하기 힘들 때, 특히, 읽기 연산을 여러 데이터베이스 서버에 분산시킴으로써 전체 시스템의 성능을 향상시키는 데 활용할 수 있다. CUBRID 복제의 경우, 마스터는 읽기/쓰기 연산을 모두 허용하나, 슬레이브는 읽기 연산만 허용하기 때문에, CUBRID 복제를 이용한 성능 향상 효과는 쓰기 연산은 제한적이고 읽기 연산이 많은 응용에서 탁월하다고 할 수 있다. 인터넷 서비스에 사용되고 있는 블로그, 게시판, 뉴스 등 많은 응용이 이러한 범주에 든다고 할 수 있다. 아래 그림은 부하 분산을 통해 성능 향상 효과를 얻기 위한 웹 서비스 구조를 보인 것이다.



위와 같은 복제 환경을 구축하기 위해서는 배포 데이터베이스를 설정한 후 슬레이브 데이터베이스를 추가하면 된다.

장애 대비 가용성 향상

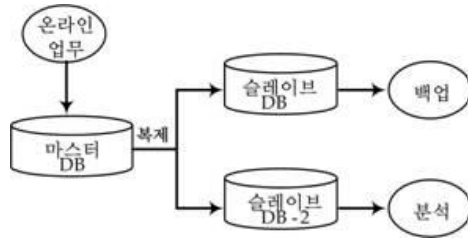
CUBRID 데이터베이스를 복제를 이용하여 슬레이브를 설정하여 두면, 마스터에 장애가 발생하였을 때, 슬레이브를 마스터로 교체하여 장애에 효과적으로 대응할 수 있다. 장애의 종류에 따라, 마스터 데이터베이스의 백업을 이용한 복구 기능으로 충분할 수도 있으나, 만약, 마스터 호스트 자체의 장애가 발생할 경우에는 마스터 호스트의 이중화(HA: high-availability라고도 부름)나 복제를 통하여 대처할 수 있다. 예를 들어, 앞의 성능 향상 예에서 마스터 데이터베이스가 운영되고 있는 호스트에 장애가 발생하면 슬레이브 데이터베이스 중 하나를 마스터로 교체하고, 응용에서도 쓰기 연산이 새로운 마스터로 전달될 수 있도록 설정하면 된다.

슬레이브를 마스터로 교체하는 작업은 그리 단순하지는 않다. 특히, 슬레이브와 마스터의 복제 설정 상태에 따라 슬레이브로 하여금 마스터 역할을 모두 수행하게 하는 것은 매우 복잡한 절차가 필요할 수도 있다. 슬레이브를 마스터로 교체하는 상세한 절차는 이 장에서 차후에 설명한다.

분리 작업을 통한 유연성 향상

CUBRID 복제를 이용하면 별도의 호스트에 마스터와 같은 내용을 갖는 데이터베이스를 확보할 수 있기 때문에 마스터 데이터베이스가 운영되는 환경에 영향을 주지 않고 여러 가지 추가적인 업무를 수행할 수 있어, 데이터베이스 사용의 유연성이 향상된다. 예를 들어, 마스터 데이터베이스가 24x7 온라인 업무에

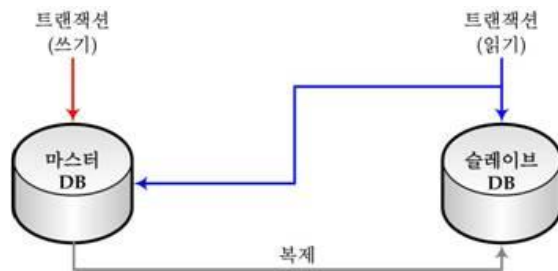
사용되고 있고, 어느 시간 대에도 백업, 분석 등 온라인 업무에 영향을 줄 만한 배치 연산을 수행할 수 없을 경우, 아래 그림과 같이 슬레이브를 만들어, 마스터에 영향을 주지 않고 배치 연산을 수행할 수 있다.



복제 기능의 특징

단방향 복제

CUBRID의 복제는 단방향 복제만을 지원한다. 즉, 다음 그림과 같이 마스터 데이터베이스의 경우에는 읽기/쓰기 연산이 모두 가능하나 슬레이브 데이터베이스에는 읽기 연산만을 허용한다. 단방향 복제 방식을 이용한 1:N 구성이 가능하다.



동기식/비동기식

데이터를 복제하는 기법으로는 마스터의 데이터를 갱신하는 작업과 슬레이브의 데이터를 동일하게 갱신하는 작업이 하나의 트랜잭션으로 처리되어 데이터의 일관성을 보장하는 동기식 복제(synchronous replication)와 이 두 작업을 분리하여 수행하여 비동기식 복제(asynchronous replication)가 있다.

- **동기식** : 동기식 모델의 경우에는 한 시스템의 장애가 전체 시스템의 장애로 확대되어 가용성 측면에서 유용성이 떨어질 우려가 있다. 예를 들어 슬레이브 시스템의 장애로 인해 마스터 시스템까지 서비스 운영이 중단될 가능성 있다.
- **비동기식** : 비동기식 복제의 경우, 마스터 시스템의 트랜잭션 로그를 활용하여 비동기식으로 슬레이브 데이터베이스에 반영함으로써 복제로 구성된 서버 간의 종속성을 최소화하여 슬레이브 시스템에 장애가 발생하여도 마스터 시스템에 영향을 주지 않는다. 따라서 CUBRID는 트랜잭션 로그를 이용한 비동기식 방식으로 복제 기능을 제공하여 데이터 일관성 보장은 물론 최대한의 가용성을 확보할 수 있도록 한다.



트랜잭션 단위의 복제

CUBRID는 트랜잭션 로그 기반으로 복제를 수행한다. 트랜잭션 로그는 데이터베이스에 대한 모든 쓰기 연산에 대해 로깅을 하여 시스템 장애 시 복구를 가능하게 한다. CUBRID는 트랜잭션 로그를 분석하고, 마스터 데이터베이스에서 변경된 항목들을 추출하여 마스터 데이터베이스에 변경된 순서대로 슬레이브 데이터베이스에 반영한다. 따라서 복제에 대해 항상 트랜잭션의 일관성을 보장할 수 있다.

온라인 복제

CUBRID는 마스터 데이터베이스 시스템의 중지 없이 온라인으로 슬레이브 데이터베이스를 동기화하여 복제를 구성할 수 있다.

스키마 독립성

슬레이브 데이터베이스의 유연성을 위해 마스터 데이터베이스와 독립적으로 스키마를 정의할 수 있다. 마스터 데이터베이스로부터 복제 받는 클래스 외 별도의 클래스, 인덱스, 사용자 계정, 트리거를 생성할 수 있으며 추가로 생성된 클래스에 대해서는 쓰기 연산도 가능하다. 예를 들어, 마스터 데이터베이스로부터 class1, class2를 복제 받으면서 슬레이브 독자적으로 class3, class4를 정의하여 운영할 수 있다.

최대한 슬레이브 데이터베이스의 유연성을 제공하기 위해 인덱스의 경우에는 기본키 인덱스를 제외한 모든 인덱스는 복제 대상에서 제외된다. 슬레이브 시스템의 목적에 따라 관리자가 적절하게 슬레이브 데이터베이스의 인덱스를 별도로 관리해야 한다. 예를 들어, 슬레이브 시스템을 데이터 분석용으로 구축한 경우에는 온라인 트랜잭션 처리용으로 구축된 마스터 데이터베이스와는 다른 인덱스 설계가 이루어져야 한다.

사용자 계정의 경우에도 마스터 데이터베이스의 계정은 복제되지 않는다. 슬레이브 자체적으로 계정을 관리해야 하며 특히 쓰기 계정 관리에 유의해야 한다. 초기 슬레이브 데이터베이스를 구성할 때에 사용하는 **repl_make_slavedb** 유틸리티에서 복제 대상 클래스의 소유자 계정을 모두 복제용 계정으로 변경하므로, 이후 추가되는 계정은 반드시 읽기만 가능한 계정으로 관리하는 것이 바람직하다.

마스터 데이터베이스에 정의된 트리거로 인한 데이터 변경 내역은 이미 마스터 데이터베이스의 트랜잭션 로그에 반영이 되었으므로 동일한 트리거가 슬레이브 데이터베이스에 정의되어 있으면 복제 시 오류가 발생하게 된다. 따라서 마스터 데이터베이스의 트리거는 복제 대상에서 제외되며 초기 슬레이브 데이터베이스 구성 시 모든 트리거는 삭제된다. 그러나, 슬레이브 데이터베이스의 활용 목적에 따라 마스터

데이터베이스와는 별도로 트리거가 필요하다면 독립적으로 트리거를 정의하여 사용할 수 있지만 이 트리거의 동작 범위가 복제 대상 클래스가 되지 않아야 한다. 예를 들어 class1, class2가 마스터로부터 복제 받는 클래스이고 class3, class4가 슬레이브 자체적으로 관리하는 클래스인 경우 class1에 데이터가 삽입되면 class3에도 동일한 데이터를 삽입하는 트리거는 슬레이브 자체적으로 정의하여 운영하여도 무방하다. 그러나 class2에 데이터가 삭제되는 경우 class1에 데이터를 삽입하는 트리거의 경우에는 복제 시 오류를 발생하게 된다.

복제 대상 오브젝트

오브젝트	복제여부
데이터	O
인덱스	O
트리거	X
사용자계정	X
스키마	O

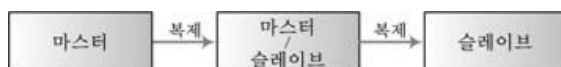
기본 키(Primary Key)의 사용

CUBRID의 복제는 기본 키 기반으로 이루어진다. 즉, 슬레이브 데이터베이스에 반영할 데이터 항목을 식별하기 위한 메커니즘으로 기본 키가 사용된다. 따라서 기본 키가 설정되지 않은 클래스는 복제 대상에서 제외된다.

현재 CUBRID의 복제는 데이터베이스 단위이나 데이터베이스 내에 존재하는 클래스 중 기본 키가 설정된 클래스만이 슬레이브로 복제 가능하다. 기본 키 설정에 대해서는 테이블 생성의 [제약 조건 정의](#)를 참고한다.

연쇄 복제(Chained Replication)

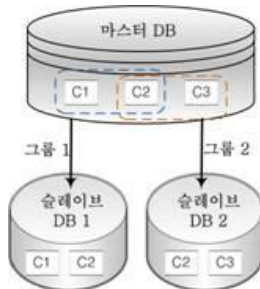
계층적 구조로 복제를 구성하는 방식을 연쇄복제(chained replication)이라고 한다. 즉, 슬레이브 데이터베이스가 동시에 마스터 데이터베이스가 되어 복제 받은 데이터 변경 사항을 다른 슬레이브 데이터베이스로 전달할 수 있다.



CUBRID에서는 이러한 방식의 복제 구조를 지원한다. 이 경우, 두 번째 마스터/슬레이브에 있는 마스터는 읽기만 가능하다.

복제 그룹(Replication Group)

하나의 마스터 데이터베이스에서 N개의 슬레이브 데이터베이스 구성을 지원한다. 이 때 관리자는 마스터 데이터베이스의 일부만 복제 그룹(Replication Group)으로 지정하여 슬레이브 데이터베이스로 복제할 수 있다.

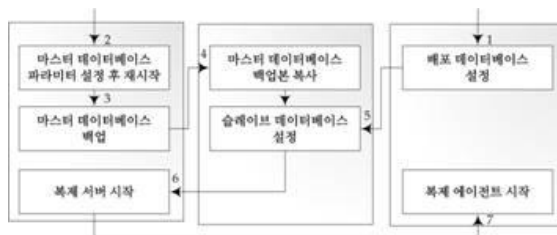


이러한 방식은 슬레이브 서버를 특정 용도로 구성하고자 할 경우 유용한 방식이다. 예를 들어, 통계 데이터만 복제하여 분석 서버로 구성하고자 할 경우 복제 그룹을 통해 구성할 수 있다.

복제 설정

복제 설정 단계

복제는 다음과 같은 단계로 설정한다.



1. 배포 데이터베이스 설정

배포 데이터베이스를 생성하고 복제에 필요한 파라미터를 설정하는 단계로서 **repl_make_distdb** 유틸리티를 이용한다.

2. 마스터 데이터베이스의 파라미터 조정 후 재시작(이미 복제 관련 파라미터가 설정되어 있는 경우 불필요)

마스터 데이터베이스의 **cubrid.conf** 파일에 **replication**이 **no**로 되어 있는 경우 이를 **yes**로 변경한 후 마스터 데이터베이스 서버를 재시작한다.

3. 마스터 데이터베이스 백업

마스터 데이터베이스와 슬레이브 데이터베이스를 동기화 하기 위해 마스터 데이터베이스를 백업한다.

4. 마스터 데이터베이스 백업본을 슬레이브 데이터베이스가 위치할 서버로 이동

마스터 데이터베이스의 백업 볼륨 및 볼륨 정보 파일을 슬레이브 시스템으로 복사한다. (ftp 나 파일 복사 이용)

5. 슬레이브 데이터베이스 설정(추가)

repl_make_slavedb 유틸리티를 이용하여 슬레이브 데이터베이스를 마스터 데이터베이스와 동기화하고 복제에 필요한 작업을 수행한다.

6. 복제 서버 시작

마스터 시스템에서 복제 서버를 구동한다.

7. 복제 에이전트 시작

슬레이브 시스템에서 복제 에이전트를 구동한다. 이 작업이 완료되면 복제 환경 구성이 종료되고 마스터 데이터베이스 백업 이후에 발생한 변경 데이터가 슬레이브 데이터베이스로 동기화된다.

배포 데이터베이스 설정

설명

배포 데이터베이스는 복제에 필요한 메타 정보를 관리하는 데이터베이스로서 복제 구성을 위해 첫 단계로 반드시 배포 데이터베이스가 구성되어야 한다. 배포 데이터베이스를 구성하기 위해서는 **repl_make_distdb** 유틸리티를 수행한다.

배포 데이터베이스는 어느 호스트에 위치해도 상관없으나 가급적 마스터 데이터베이스가 운영 중인 호스트에는 구축하지 않는 것이 성능상 유리하다.

보안을 위해 배포 데이터베이스는 항상 **DBA** 계정으로 운영되어야 한다. **repl_make_distdb** 유틸리티는 배포 데이터베이스명과 배포 데이터베이스의 **DBA** 계정 암호를 입력으로 받는다.

구문

```
repl_make_distdb dist_db_name [ -p password ]
```

항상 배포 데이터베이스가 위치할 디렉토리에서 **repl_make_distdb** 유틸리티를 수행한다. 암호를 입력하지 않으면 암호가 설정되지 않으나 보안을 위해서는 반드시 암호를 설정하는 것이 바람직하다.

repl_make_distdb 유틸리티를 수행하면 다음과 같은 안내가 출력된 후 사용법이 출력된다.

```
#####
#
#          CUBRID 복제 환경 구성 : 복제 에이전트 구성          #
#
# 복제 환경을 구성하는 작업은 다음과 같은 순서로 진행되어야 합니다. #
#   1. 복제 에이전트 구성 (repl_make_distdb 스크립트 구동)      #
#   2. 마스터 DB 전체 백업 (cubrid backupdb 유틸리티구동)      #
#   3. 마스터 DB 백업본 복사 (마스터 DB 호스트->슬레이브 DB 호스트) #
#   4. 슬레이브 DB 구축 (repl_make_slavedb 스크립트 구동)      #
#   5. 복제 서버구동 (cubrid repl_server 유틸리티 구동)        #
#   6. 복제 에이전트 구동 (cubrid repl_agent 유틸리티 구동)    #
#
# NOTE: 마스터 DB를 백업받는 시점은 슬레이브 DB 구축시점 이전이면 #
#       언제든지 상관없으나 온라인백업을 하는 경우 가급적     #
#       슬레이브 DB 구축 바로 전에 수행하시면 초기 복제에       #
#       시간을 절약하실 수 있습니다.                             #
#       필요한 백업 파일은 다음과 같습니다.                     #
#           - master_db_name.bk_vinf                             #
#           - master_db_name.bk0v???                             #
#
# 복제 에이전트가 작업하기 위해 필요한 배포 DB를 생성합니다.    #
#   - 배포 DB는 반드시 DBA 계정으로 동작해야 하므로 DBA 계정을 #
#     지정하시기 바랍니다.                                       #
#
# CUBRID 에서는 하나의 슬레이브 DB 당 하나의 배포 DB가 생성됩니다. #
#####
```

```
# - 배포 DB 및 복제 에이전트(repl_agent)는 슬레이브 DB가 위치한 #
# 호스트에서 수행되어야 합니다. #
# Ctrl-C 등으로 비정상적으로 스크립트를 중단하는 경우에는 #
# cubrid server stop/cubrid deletedb 등의 유틸리티를 이용하여 #
# 배포 DB를 삭제한 후 다시 수행하십시오. #
#####
```

repl_make_distdb 유틸리티는 총 6단계로 이루어지며 이중 단계 5, 6에서 필요한 정보를 입력해야 한다.

```
repl_make_distdb sample_dist -p dbal
```

위와 같이 **repl_make_distdb** 유틸리티를 수행하면 유틸리티는 다음과 같이 단계별 진행상황을 출력하면서 사용자 입력이 필요한 경우 입력 값을 묻는 프롬프트를 출력한다.

```
STEP 1 : 배포 DB를 생성 중입니다. 잠시만 기다려 주십시오.
STEP 2 : 배포 DB 서버를 구동하고 있습니다. 잠시만 기다려 주십시오.
STEP 3 : 배포 DB 의 DBA 계정을 설정합니다.
STEP 4 : 복제에 필요한 테이블들을 생성합니다.
STEP 5 : 복제 대상 마스터 DB 정보를 입력합니다.
    1. 마스터 DB의 이름을 입력하십시오. >>
    2. 마스터 DB가 위치한 호스트의 IP 주소를 입력하십시오.
       - IP 주소가 부정확하게 입력되면 복제가 수행되지 않습니다.
         마스터 DB IP >>
    3. 복제서버(cubrid repl_server)가 사용하는 TCP/IP 포트번호를 입력 하시오.>>
    4. 복제에 필요한 복제로그를 저장할 디렉토리를 입력하십시오.
       /home1/cubrid/CUBRID이면 엔터 키를 입력하십시오. >>
STEP 6 : 복제 환경 변수를 설정합니다.
    1. 추적 로그를 저장할 디렉토리를 입력하십시오.
       /home1/cubrid/CUBRID이면 엔터 키를 입력하십시오. >>
    2. 에러로그를 저장할 디렉토리를 입력하십시오.
       /home1/cubrid/CUBRID이면 엔터 키를 입력하십시오. >>
    3. 복제 에이전트(repl_agent)의 상태 표시를 위한 TCP/IP 포트번호를 입력하십시오.>>
    4. 복제 지연 시간 log 파일의 크기(line 수)를 입력하십시오. >>
    5. 네트워크 오류시 복제의 재 시작 여부를 입력하십시오. (y/n) >>
```

- **(단계 1) 배포 DB 생성** : 배포 데이터베이스를 생성한다.
- **(단계 2) 배포 DB 서버 구동** : 배포 데이터베이스 서버를 구동한다.
- **(단계 3) 배포 DB의 DBA 계정 설정** : 입력 값으로 지정된 암호로 DBA 계정을 설정한다.
- **(단계 4) 배포 DB 스키마 생성** : 복제에 필요한 배포 DB의 각종 클래스를 생성한다.
- **(단계 5) 복제 대상 마스터 데이터베이스 정보 입력** : 복제 대상 마스터 데이터베이스 명을 입력하고 마스터 데이터베이스가 위치한 호스트의 IP 주소를 입력한다. 그리고 복제 서버와 통신하기 위한 복제 서버의 TCP 포트 번호를 입력한다. 복제 서버는 마스터의 트랜잭션 로그를 읽어 복제 에이전트에 전달하는 프로세스로서 복제 에이전트와 통신하기 위해 TCP 포트를 사용한다. 또한 복제 로그를 저장하기 위한 경로를 입력한다. 복제 로그는 마스터 데이터베이스의 트랜잭션 로그를 복제를 위해 임시로 저장하는 파일이며 복제가 시작되면 <master_db_name>.copy 라는 파일이 지정된 경로에 생성된다.
- **(단계 6) 복제 환경 변수 설정** : 배포 DB 설정의 마지막 단계로 추적 로그를 저장할 경로와 에러로그를 저장할 경로를 입력한다. 추적 로그는 복제 에이전트가 마지막으로 처리한 트랜잭션 로그의 위치를 저장하는 파일로서 장애가 발생했을 때 이 파일에 기록된 위치 정보로 재 시작 시점을 결정한다. 따라서 이 파일은 항상 손상되지 않도록 유지해야 한다. 에러로그는 복제 수행 시 발생하는 오류를 기록하는 파일이다.

추가로 복제 에이전트 상태 관리 및 복제 지연 시간 관리, 접속 유지를 위해서는 다음의 세 가지 정보가 필요하다.

- **복제 에이전트(repl_agent)의 상태 표시를 위한 TCP/IP 포트 번호** : repl_check_sync는 repl_agent의 현재 상태를 보기 위해 repl_agent에 접속하는데, 이 때 사용하는 통신 포트이다. 기본 포트 번호는 33333이다.
- **복제 지연 시간 log 파일의 크기(line 수)** : 복제 성능 모니터링 목적으로 복제 지연시간 로그를 남기는데, 이때 기록할 파일의 크기를 정의한다.
- **복제의 재시작 여부** : 복제 서버와 복제 에이전트 사이의 연결이 단절되면 복제 에이전트가 재접속 시도 여부를 의미한다. 네트워크가 단절되면 복제 에이전트가 종료되는데, y로 설정하면 복제 에이전트를 재시동하여 복제 기능을 계속 실행하고, n로 설정하면 그대로 복제 기능이 종료한다.

마스터 데이터베이스 설정 및 백업

다음 단계로는 마스터 데이터베이스의 설정을 변경하고 슬레이브 데이터베이스와 동기화를 위해 백업 볼륨을 생성해야 한다. 마스터 데이터베이스가 복제에 필요한 트랜잭션 로그를 생성하도록 하기 위해서 **\$CUBRID/conf/cubrid.conf** 파일을 수정하여 **replication** 파라미터를 **yes**로 변경한 후 마스터 데이터베이스 서버를 재시작한다. 이미 이 파라미터가 **yes**로 설정되어 있다면 마스터 데이터베이스를 재시작할 필요는 없다. **\$CUBRID/conf/cubrid.conf** 파일에서 아래와 같이 **replication**을 **yes**로 설정한다.

```
[@sample master]
replication=yes
```

위의 예는 **replication** 파라미터가 sample_master 데이터베이스에만 적용되는 예이다. 위와 같이 **[@<dbname>]** 밑에 파라미터를 지정하면 해당 <dbname> 예만 파라미터가 적용된다.

변경된 파라미터를 서버에 반영하기 위해 마스터 데이터베이스 서버를 재시작한다.

```
cubrid server stop sample_master      -- 서버 중지
cubrid server start sample_master     -- 서버 시작
```

이제 현 상태의 마스터 데이터베이스와 슬레이브 데이터베이스 간 동기화 작업을 위해 마스터 데이터베이스를 백업 받는다.

```
cubrid backupdb sample_master
```

이때 **-r** 옵션을 사용하여 보관 로그를 삭제하게 되면 경우에 따라 복제가 시작되지 않을 수 있다. CUBRID는 온라인 복제를 지원하므로, 슬레이브 데이터베이스를 동기화 하는 동안 마스터 데이터베이스에 대량으로 쓰기가 발생하여 백업 받은 이후 변경 분이 이미 보관 로그 파일로 넘어가는 경우가 발생할 수 있는데, 이때 복제 시스템은 보관 로그 파일을 읽어 이를 슬레이브에 반영한다. 따라서 슬레이브 데이터베이스에 반영되지 않은 로그가 트랜잭션 로그 보관 로그에 있는 경우를 대비하여 보관 로그를 삭제하지 않는 모드로 백업을 수행해야 한다.

백업 후 다음과 같은 명령어로 마스터 데이터베이스의 백업 볼륨과 정보 파일을 확인한다.

```
ls sample_master_bk*
```

슬레이브 데이터베이스와 동기화를 위해 백업볼륨들과 백업볼륨 정보 파일을 ftp나 파일 복사를 통해 슬레이브 호스트로 전송한다.

슬레이브 데이터베이스 추가

설명

마스터 데이터베이스의 백업 볼륨을 슬레이브 호스트로 복사한 후에 슬레이브 데이터베이스를 추가하기 위해 **repl_make_slavedb** 유틸리티를 수행한다.

repl_make_slavedb는 마스터 데이터베이스명, 슬레이브 데이터베이스명, 복제 계정의 사용자명 및 암호를 입력 받는다.

구문

```
repl_make_slavedb master_db_name slave_db_name -u userid -p passwd
```

다음은 **repl_make_slavedb**를 수행하면 출력되는 도움말 화면이다.

```
#####
#
#          CUBRID 복제 환경 구성 : 슬레이브 DB   구성          #
#
#   복제 환경을 구성하는 작업은 다음과 같은 순서로 진행되어야 합니다.  #
#   1. 복제에이전트 구성 (repl_make_slavedb 스크립트 구동)          #
#   2. 마스터 DB 전체 백업 (cubrid backupdb 유틸리티구동)          #
#   3. 마스터 DB 백업본 복사 (마스터 DB 호스트->슬레이브 DB 호스트) #
#   4. 슬레이브 DB 구축 (repl_make_slavedb 스크립트 구동)          #
#   5. 복제서버 구동 (cubrid repl_server 유틸리티 구동)            #
#   6. 복제에이전트 구동 (cubrid repl_agent 유틸리티 구동)          #
#
#   NOTE1: 슬레이브 DB 구성 전에 반드시 repl_make_distdb 스크립트를 #
#   이용하여 배포DB를 생성해야 합니다.                             #
#   NOTE2: 마스터 DB에 대하여 전체 백업을 수행해야 합니다.         #
#   백업받는 시점은 슬레이브 DB 구축시점 이전이면 언제나          #
#   상관없으나, 온라인백업을 하는 경우 가급적 슬레이브 DB를      #
#   구축하기 바로 전에 수행하시면 초기 복제에 소요되는             #
#   시간을 절약할 수 있습니다.                                     #
#   필요한 백업 파일은 다음과 같습니다.                             #
#   - master_db_name.bk_vinf                                         #
#   - master_db_name.bk0v???                                         #
#   NOTE3: 현재 이 스크립트는 반드시 가져온 백업 파일이 있는       #
#   디렉토리에서 수행되어야 합니다.                                 #
#
#   Ctrl-C 등으로 비정상적으로 스크립트를 중단하는 경우에는         #
#   cubrid server stop/cubrid deletedb 등의 유틸리티를 이용하여   #
#   슬레이브 DB를 삭제한 후 다시 수행하십시오.                     #
#####
```

repl_make_slavedb 유틸리티는 모두 5단계로 이루어지며 그 중 단계 1, 3, 5 에서 필요한 정보를 입력해 주어야 한다.

```
STEP 1 : 마스터 DB의 백업본을 복구하기 위한 사전 작업을 진행합니다.
a. 구축할 슬레이브 DB의 볼륨이 저장될 디렉토리 경로를 입력하십시오.
   현재 디렉토리이면 엔터 키를 입력하십시오. >>
b. 구축할 슬레이브 DB의 로그 볼륨이 저장될 디렉토리 경로를 입력하십시오.
   현재 디렉토리이면 엔터 키를 입력하십시오. >>

STEP 2 : 슬레이브 DB 백업본을 복구합니다.
- 백업 파일과 백업볼륨 정보 파일이 반드시 현재 디렉토리에 존재해야 합니다.
- 슬레이브 DB를 복구하고 있습니다. 잠시만 기다려 주시기 바랍니다...
- 슬레이브 DB 가 복구되었습니다.

STEP 3 : 복구된 최종 로그 번호와 슬레이브 DB 정보를 배포 DB에 기록합니다.
a. 배포 DB 이름을 입력하십시오. >>
```

```

- 배포 DB가 같은 호스트에 없습니다.
- 배포 DB가 위치한 호스트의 IP 주소를 입력하십시오. >>
b. 배포 DB에 접속하기 위한 DBA 계정의 암호를 입력하십시오. >>
STEP 4 : 슬레이브 DB를 생성하고 구동합니다.
STEP 5 : 슬레이브 DB 구축 후 사후 처리 단계
- 이 작업을 위해서는 마스터 DB의 DBA 계정이 필요합니다.
- 마스터 DB의 DBA 계정을 입력하십시오. >>

```

- **(단계 1) 사전 작업** : 슬레이브 데이터베이스의 데이터 볼륨이 위치할 디렉토리 경로와 로그 볼륨이 위치할 디렉토리 경로를 입력한다. 이때 반드시 절대 경로를 입력해 주어야 한다.
- **(단계 2) 슬레이브 데이터베이스 동기화** : 마스터 데이터베이스의 백업 볼륨을 복구하여 슬레이브 데이터베이스로 구축하여 양쪽 데이터베이스를 동기화한다.
- **(단계 3) 동기화된 슬레이브 DB 정보를 배포 DB에 기록** : 슬레이브 데이터베이스에 반영된 최종 로그 번호를 배포 DB에 기록하여 복제 에이전트가 사용할 수 있도록 한다.
- **(단계 4) 슬레이브 DB 구동** : 슬레이브 데이터베이스 서버를 구동한다.
- **(단계 5) 사후 처리 단계** : 슬레이브 데이터베이스 내의 모든 클래스의 소유자를 입력 받은 복제 계정으로 변경하여 불필요한 쓰기 연산이 일어나지 않도록 조치하고 사용자가 정의한 모든 트리거를 제거하여 복제 시 오류가 발생하지 않도록 한다. 이 작업을 위해서는 마스터 데이터베이스의 DBA 계정이 필요하다.

복제 파라미터 설정과 복제 그룹을 설정할 필요가 없으면 아래 6, 7 단계는 (q-중지)로 통과할 수 있다.

```

STEP 6 : 파라미터 설정 단계
1. perf_poll_interval      - 복제 지연시간을 측정하는 단위(초)
2. size_of_log_buffer      - 복제 에이전트의 로그버퍼 크기(페이지)
3. size_of_cache_buffer    - 복제 에이전트의 복제로그버퍼 크기(페이지)
4. size_of_copylog         - 복제로그의 페이지 수
5. index_replication       - 인덱스 복제 여부
6. for_recovery            - 마스터 교체용 복제 여부
7. log_apply_interval      - 복제 수행 주기(초)
8. restart_interval        - 슬레이브 재접속 주기(초)
-- 변경하고자 하는 파라미터 번호 (q - 중지) q

```

- **(단계 6) 파라미터 설정 단계** : 슬레이브 데이터베이스의 복제 환경을 구성하기 위한 파라미터로 각 항목에 대해 사용자 정의할 수 있다. 변경하지 않을 경우 기본값이 설정된다. 자세한 것은 [복제 파라미터 설정](#)을 참조한다.

```

STEP 7 : 복제 그룹 설정 단계
- 복제 대상 클래스를 선정합니다.

MASTER_HOST :
-----
$master_db_name의 모든 클래스가 복제 대상으로 지정되어 있습니다.
복제 그룹을 다시 설정하시겠습니까? (y 또는 n) >>
1. 복제 그룹을 초기화하고 클래스를 새로 추가
2. 전체 클래스 중에서 복제 제외 클래스 지정
==> 작업 번호 입력 (q - quit) >>

```

- **(단계 7) 복제 그룹 설정 단계** : 마스터 데이터베이스와 동일하게 슬레이브 데이터베이스 구성하는 것이 기본값이다. 이 기본 구성에 복제할 클래스들을 추가 또는 제외 등을 통해 변경할 수 있는 방법을 제공한다. 자세한 것은 [복제 그룹 설정](#)을 참고한다.

주의 사항

repl_make_slavedb 유틸리티로 슬레이브 데이터베이스를 새로 생성한 경우에는 반드시 기존에 사용하던 복제로그, 추적로그 등이 남아 있는지를 확인하고 삭제해야 정상적으로 복제가 진행된다. 예를 들어 **repl_make_distdb** 유틸리티 수행 시 복제로그 및 추적로그의 파일 경로를 **/home/replication/log**로 설정한 경우 해당 디렉토리 아래에 **<master_db_name>.copy**, **<master_db_name>.copy.ar0**, **<dist_db_name>.tail** 등의 파일이 있는지 확인해야 한다.

복제 서버 및 에이전트 구동

설명

복제를 시작하기 위해서는 복제서버와 복제에이전트를 구동해야 한다. 복제서버는 마스터 데이터베이스가 수행되는 호스트에서 트랜잭션 로그를 읽어 복제 에이전트에게 전달하는 작업을 한다.

구문 1

```
cubrid repl_server command
command:
start master-database-name server-network-port [-a max-agent-num] [-e error-file]
stop master-database-name
status
```

복제서버 수행시 파라미터로 마스터 데이터베이스명과 복제에이전트와의 통신을 위한 TCP 포트번호를 필수로 입력해야 하며 TCP 포트 번호는 반드시 **repl_make_distdb** 유틸리티 수행 시 입력한 복제서버의 TCP 포트번호와 일치해야 한다. 옵션으로 서비스 가능한 복제에이전트의 수를 입력할 수 있다.

복제서버가 정상적으로 수행되고 있는지를 확인하기 위해서는 **cubrid repl_server status**를 사용한다. 보다 자세한 설명은 [CUBRID 유틸리티 사용법\(구문\)](#)을 참조한다.

마스터 호스트에서 복제서버를 수행한 후 최종적으로 슬레이브 호스트에서 복제 에이전트를 수행하면 복제 구성이 완료되고 복제 서비스가 시작된다.

구문 2

```
cubrid repl_agent command
command:
start dist-database-name [dba-password]
stop dist-database-name
status
```

복제에이전트는 배포 DB명과 배포 DB의 **DBA** 계정 암호를 입력한다. 배포 DB의 **DBA** 계정 암호가 없는 경우에는 입력하지 않아도 된다.

복제에이전트가 수행되면 **repl_make_distdb**에서 설정한 복제로그 경로, 에러로그 경로, 추적로그 경로에 각각 복제로그, 에러로그, 추적로그가 생성된다.

복제로그는 **<master_db_name>.copy**로 추적로그는 **<dist_db_name>.trail**로, 에러로그는 **<dist_db_name>.err**로 생성된다. 이 파일이 손상되는 경우에는 복제가 진행될 수 없으므로 주의하여 관리해야 한다.

cubrid repl_agent status로 복제 에이전트의 상태를 확인할 수 있다.

슬레이브 데이터베이스 삭제

슬레이브 데이터베이스를 완전히 삭제하기 위해서는 복제 에이전트를 중지한 후 배포 DB를 삭제한다.

- (단계 1) 복제에이전트 중지 : **cubrid repl_agent stop**을 사용하며 배포DB명을 파라미터로 지정한다.

```
cubrid repl_agent stop sample_dist
```

- (단계 2) 배포DB 삭제 : 배포DB 서버를 중지하고 배포DB를 삭제한다.

```
cubrid server stop sample_dist  
cubrid deletedb sample_dist
```

- (단계 3) 복제 관련 로그 삭제 : 복제 수행시 만들어진 *<master_db_name>.copy*,
<master_db_name>.copy.ar0, *<dist_db_name>.trail*, *<dist_db_name>.err* 파일을 삭제한다.

만약, 일시적으로 복제 작업을 중단하는 경우에는 복제 에이전트와 복제 서버만 중지하고 추후 복제를 다시 시작할 때 복제 서버를 먼저 시작한 후 복제 에이전트를 시작한다.

복제 그룹 설정

설명

마스터 데이터베이스로부터 특정 클래스들만 복제하여 슬레이브 데이터베이스를 구성하고자 할 경우, **repl_make_group** 유틸리티를 이용하여 복제 그룹을 구성한다. 기본 복제 구성은 마스터 데이터베이스에서 기본키가 정의된 모든 클래스를 대상으로 하며, 슬레이브 데이터베이스가 생성되어 있어야 한다.

구문

```
repl_make_group master_db_name dist_db_name [option]
```

repl_make_group의 옵션은 다음과 같다.

옵션	설명
-p passwd	배포 데이터베이스에 대한 DBA 암호.
-f file_name	복제 대상 클래스의 목록 파일. 파일은 공백 또는 쉼표로 구분된 클래스명으로 구성.
-a class_name_list	그룹 복제할 클래스 추가. 추가할 클래스는 공백 또는 쉼표로 구분.
-d class_name_list	그룹 복제할 클래스 삭제. 삭제할 클래스는 공백 또는 쉼표로 구분.
-i	그룹 복제 초기화 여부 결정. Y/N 으로 기본값은 N.

repl_make_group 유틸리티를 통해 클래스 별로 복제 여부를 지정하기 위해서는 다음과 같은 방법으로 설정한다.

- (단계 1) 모든 클래스가 복제 대상으로 지정되어 있을 경우 : 슬레이브 데이터베이스를 기본으로 구성한 후, 처음 **repl_make_group** 유틸리티를 통해 복제 그룹을 구성하고자 할 때는 아래와 같이 두 가지 옵션이 제공된다.

```
MASTER_HOST :
-----
$master_db_name의 모든 클래스가 복제 대상으로 지정되어 있습니다.
복제 그룹을 다시 설정하시겠습니까? (y 또는 n) >>
1. 복제 그룹을 초기화하고 클래스를 새로 추가
2. 전체 클래스 중에서 복제 제외 클래스 지정
==> 작업 번호 입력 (q - quit) >>
```

- (단계 2) 부분 클래스가 복제 대상으로 지정되어 있을 경우 : 부분 클래스가 슬레이브 데이터베이스로 구성되어 있을 경우 설정할 수 있는 옵션은 아래와 같이 5 가지 방법이 제공된다.

```
MASTER_HOST : 192.168.2.77
-----
$master_db_name의 모든 클래스가 복제 대상으로 지정되어 있습니다.
복제 그룹을 다시 설정하시겠습니까? (y 또는 n) >>
1. 복제 그룹을 초기화하고 클래스를 새로 추가
2. 전체 클래스 중에서 복제 제외 클래스 지정
3. 현재 상태에서 새로운 클래스 추가
4. 현재 상태에서 복제 제외 클래스 지정
5. 전체 클래스를 복제 대상으로 지정
```

주의 사항

- 복제 운영 중 복제 그룹 유틸리티를 통해 복제 대상을 추가한 경우, 스냅샷 동기화(**repl_make_snapshot**) 유틸리티를 통해 마스터 데이터베이스의 해당 클래스에 대한 데이터를 동기화시켜 주어야 한다.
- 복제 대상 클래스가 다른 클래스를 참조하고 있다면 반드시 참조 클래스도 같이 복제 대상으로 지정한다.
- 입력 클래스는 복수로 지정할 수 있으며, 공백 또는 쉼표로 구분해 준다.

복제 파라미터 설정

설명

이미 설정된 복제 관련 파라미터는 슬레이브 데이터베이스 단위로 변경할 수 있다. 변경 가능한 파라미터는 8가지로 **repl_change_param** 유틸리티를 통해 지원한다.

구문

```
repl_change_param master_db_name slave_db_name dist_db_name [option]
```

repl_change_param 의 옵션은 다음과 같다.

옵션	설명
-p <i>passwd</i>	배포 데이터베이스에 대한 DBA 암호.
-n <i>parameter_name</i>	설정할 수 있는 파라미터 이름. 소문자로 입력.

-v parameter_value **-n** 옵션에서 정의한 파라미터에 대한 값

--help 도움말 메시지 출력

repl change param masterdb slavedb distdb

1. perf_poll_interval - 복제 지연시간을 측정하는 단위(초)
2. size_of_log_buffer - 복제 에이전트의 로그버퍼 크기(페이지)
3. size_of_cache_buffer - 복제 에이전트의 복제 로그버퍼 크기(페이지)
4. size_of_copylog - 복제로그의 페이지 수
5. index_replication - 인덱스 복제 여부
6. for_recovery - 마스터 교체용 복제 여부
7. log_apply_interval - 복제 수행 주기(초)
8. restart_interval - 슬레이브 재접속 주기(초)

슬레이브 데이터베이스의 복제 환경을 구성하기 위한 파라미터로 각 항목에 대해 사용자 정의할 수 있다.
변경하지 않을 경우 기본값이 설정된다.

1. perf_poll_interval
 - 허용 범위 : 10 ~ 60
 - 복제 지연 시간 측정을 설정하는 파라미터로 단위는 초(sec)이다. 기본값은 **10**이다.
2. size_of_log_buffer
 - 허용 범위 : 100 ~ 1000
 - repl_agent가 트랜잭션 로그를 임시로 저장하는 버퍼의 수를 지정한다. 한 버퍼의 크기는 데이터베이스 페이지 크기와 동일하다. 기본값은 **500**이다.
3. size_of_cache_buffer
 - 허용 범위 : 100 ~ 500
 - repl_agent가 복제 로그를 임시로 저장하는 버퍼의 수를 지정한다. 한 버퍼의 크기는 데이터베이스 페이지 크기와 동일하다. 기본값은 **100**이다.
4. size_of_copylog
 - 허용 범위 : 1000 ~ 10000
 - 복제 로그의 페이지 수를 정의한다. 만약 지정된 페이지 수 이상인 경우 이미 처리 완료된 페이지는 truncate된다. 기본값은 **5000**이다.
5. index_replication
 - 허용값 : Y/N
 - 인덱스를 자동 복제하고자 하는 경우 설정한다. Y는 "설정함"이고 N은 "설정하지 않음"이다. Y/N는 대소문자를 구별하지 않는다. 기본 값은 **N**이다.
6. for_recovery
 - 허용값 : Y/N
 - 마스터 데이터베이스에 장애 발생시 마스터 데이터베이스로 대체할 목적으로 복제를 구성할 경우 설정한다. 이 값이 Y로 설정되면, **index_replication** 값도 Y로 설정되어 동작한다. 즉 마스터의 모든 클래스가 복제된다. Y는 "설정함"이고 N은 "설정하지 않음"이다. Y/N는 대소문자를 구별하지 않는다.

- 한번 마스터 데이터베이스 교체 대비 서버로 설정(for_recovery=Y)한 후, 다시 **for_recovery** 파라미터의 값을 N으로 변경하여 교체 대비 서버 목적에서 해제할 수 있다. 해제한 후 다시 교체 대비 서버로 설정하고자 한다면 처음부터 새로 복제를 구성해야 한다. 그리고 하나의 슬레이브 데이터베이스는 하나의 마스터 데이터베이스만이 **for_recovery** 파라미터를 설정할 수 있다. 기본 값은 **N**이다.

7. log_apply_interval

- 허용 범위 : 0 ~ 600
- 마스터 데이터베이스의 변경 내용을 슬레이브 데이터베이스에 반영하는 간격을 설정하는 파라미터로 단위는 초(sec)이다. 0으로 설정할 경우는 실시간 반영을 의미한다. 기본값은 **0**이다.

8. restart_interval

- 허용 범위 : 1 ~ 60
- 슬레이브 데이터베이스에 접속이 안될 경우 재접속 하기 위한 간격을 설정하는 파라미터로 단위는 초(sec)이다. 기본값은 **100**이다.

예제

```
$ repl change param masterdb slavedb distdb admin
-- 변경하고자 하는 파라미터 번호 (q - 중지) > > 1
> > perf_poll_interval 값 (10 ~ 60) > > 10
-- 변경하고자 하는 파라미터 번호 (q - 중지) > > q
$ repl change param masterdb slavedb distdb admin -n size_of_cache_buffer -v 500
파라미터가 정상적으로 변경되었습니다.
```

스냅샷 동기화

설명

스냅샷 동기화(**repl_make_snapshot**)는 마스터 데이터베이스의 복제 대상 클래스 중 복제에 참여하지 않던 것을 운영 중에 복제에 추가하고자 할 때 사용하는 유틸리티이다. 적용 대상은 **repl_make_group** 유틸리티로 마스터 데이터베이스의 일부 클래스만 복제하는 슬레이브 데이터베이스이다.

구문

```
repl_make_snapshot -m masterdb_name -s slavedb_name -d distdb_name -cf classes_file_name
[-p dist_db_password]
```

- **-m masterdb_name** : 마스터 데이터베이스 이름
- **-s slavedb_name** : 슬레이브 데이터베이스 이름
- **-d distdb_name** : 배포 데이터베이스 이름
- **-cf classes_file_name** : 추가할 클래스 목록에 대한 파일 이름으로, 클래스는 공백 또는 쉼표(.)로 구분
- **-p dist_db_password** : 배포 데이터베이스에 대한 **DBA** 암호

마스터 데이터베이스 교체(수동)

마스터 데이터베이스에 장애가 생겨 복구가 불가능한 경우 슬레이브 데이터베이스를 마스터 데이터베이스로 교체할 수 있다. 마스터 데이터베이스로 교체하기 위해서는 다음과 같은 조건을 만족해야 한다.

- 마스터 데이터베이스의 모든 클래스에 기본 키가 설정되어 있어 슬레이브 데이터베이스로 복제가 되어야 한다.
- 마스터 데이터베이스의 최종 스키마 정보가 백업되어 있어 마스터 데이터베이스가 가진 사용자 계정, 설정된 인덱스와 트리거 정보를 얻을 수 있어야 한다.
- 슬레이브 데이터베이스의 복제용 계정을 삭제하려면 모든 작업이 완료된 후 수작업으로 삭제해야 한다.
- 마스터 데이터베이스에서 정의한 시리얼은 복구되지 않는다. 따라서 시리얼을 사용한 경우에는 모든 작업이 완료된 후 수작업으로 시리얼의 최대값을 조정해야 한다.

슬레이브 데이터베이스를 사용하여 마스터 데이터베이스를 교체하는 과정은 다음과 같다.

사전 단계: 마스터 데이터베이스 스키마 백업

마스터 데이터베이스의 사용자 계정, 클래스, 인덱스, 트리거 등 스키마가 변경될 때마다 **cubrid unloaddb** 유틸리티를 이용하여 항상 스키마 정보를 백업해야 한다.

```
cubrid unloaddb -s -C master_db_name
```

cubrid unloaddb 유틸리티 수행 결과로 master_db_name_schema와 master_db_name_trigger라는 파일이 생성되고 이 파일을 보관했다가 슬레이브 데이터베이스를 마스터 데이터베이스로 교체할 때 사용한다.

단계 1: 슬레이브 데이터베이스 백업 및 복사

cubrid backupdb 유틸리티를 이용한 전체 백업을 수행한 후 마스터 데이터베이스가 위치할 호스트로 백업 볼륨, 활성 로그 파일, 그리고 보관 로그 파일을 모두 복사한다.

```
cubrid backupdb slave_db_name
```

마스터 데이터베이스로 교체하기 위해서는 반드시 슬레이브 데이터베이스 전체를 백업해야 한다. 백업 볼륨은 물론 활성 로그 파일 및 보관 로그 파일도 모두 복사해야 전체를 복구할 수 있다.

단계 2: 슬레이브 데이터베이스 복구 후 이름 변경

마스터 호스트에서 백업 볼륨을 복구한 후 이름을 변경한다.

```
cubrid restoredb slave_db_name
cubrid renamedb slave_db_name master_db_name
```

만약, 슬레이브 호스트에서의 데이터베이스 볼륨 경로와 마스터 호스트에서의 데이터베이스 볼륨 경로가 다른 경우에는 **cubrid restoredb**의 **-n** 옵션을 이용하여 볼륨 경로를 이전 마스터 데이터베이스와 동일하게 지정한다. **cubrid restoredb** 유틸리티에 대한 상세 설명은 [데이터베이스 복구](#)를 참조한다.

"사전 단계: 마스터 데이터베이스 스키마 백업"에서 보관했던 스키마 정보 파일을 이용하여 장애 이전 마스터 데이터베이스와 동일한 사용자 계정 및 인덱스를 생성한다.

```
call add user('REPL', '') on class db root to auser;
```

```
call find user('REPL') on class db user to auser;
```

```
csql -S -u dba -p dba_passwd master db name -e -i master db name schema
```

다음 단계는 동일한 방식으로 마스터에 정의된 트리거 정보를 다시 반영하는 작업이다.

```
csql -S -u dba -p dba passwd master db name -e -i master db name trigger
```

시리얼 값이 복구되지 않으므로 시리얼의 현재 값을 수작업으로 조정한다.

[illegible]

복제 계정을 삭제한다.

```
csql -S -u dba -p dba_passwd master db name -c "drop user repl user"
```

자동 교체 방식은 수동 교체 방식에 비해 좀 더 빠르고 편리하게 슬레이브 데이터베이스를 마스터 데이터베이스로 교체할 수 있는 방법이다. 교체 작업은 **repl_check_sync**와 **repl_change_master** 유틸리티를 사용하여 진행한다. 슬레이브 데이터베이스의 복제 파라미터 옵션 중 **for_recovery**가 'Y'로 설정된 것만 자동 교체가 가능하다.

슬레이브 데이터베이스를 만들 때 모든 클래스의 소유권이 복제 사용자로 변경된다. 따라서

for_recovery에서도 권한 변경은 복제되지 않고, 사용자 추가 및 삭제 그리고 그룹 추가 및 삭제 등만

복제된다. 슬레이브 데이터베이스를 마스터 데이터베이스로 교체한 후 사용자별 권한은 **DBA**가 설정해야 한다. 트리거 역시 일괄적으로 슬레이브 데이터베이스에서 비활성화 되어 있으므로, 마스터 데이터베이스로 교체 후 **DBA**가 활성화시켜 주어야 한다.

슬레이브 데이터베이스의 동기화 상태 점검(repl_check_sync)

설명

마스터 데이터베이스에 장애가 발생하면 슬레이브 데이터베이스 중 하나를 마스터 데이터베이스로 교체해야 한다. 이 때 가장 우선 해야 할 일은 모든 슬레이브 데이터베이스 간의 동기화 상태와 동기화가 완료되었는지를 점검하는 것이다. **repl_check_sync** 유틸리티는 현재 구성된 슬레이브 데이터베이스의 동기화 상태를 점검한다.

구문

```
repl_check_sync dist_db_name.config [-p passwd]
```

- *dist_db_name.config*: 배포 데이터베이스 구성 로그 파일
- *-p passwd*: 배포 데이터베이스에 대한 **DBA** 암호.
- *dist_db_name.config* 파일 규격: 라인별로 **cubrid repl_agent** 호스트 기록(hostname port_id)

예제

```
192.168.2.100 5627
192.168.2.200 5627
```

실행 결과는 슬레이브 데이터베이스별로 최종 반영된 로그 번호를 출력한다.

결과:

agent_ip	status	log number	last_updated_time
1.1.1.1	A	1234/110	15:30:30
1.1.1.2	A	1234/123	15:29:30
1.1.1.3	A	1234/123	15:28:30
1.1.1.4	A	1234/123	15:27:30
1.1.1.5	A	1234/123	15:26:30

출력 결과 중 status는 다음과 같은 정보로 구성된다.

A : Active(복제 진행 중)

F : First(처음 복제 시작)

I : Idle(휴식 - 복제 내용 없음)

슬레이브 데이터베이스를 마스터 데이터베이스로 교체(repl_change_master)

설명

repl_check_sync 유틸리티를 통해 최종 동기화 상태를 확인한 후 슬레이브 데이터베이스를 마스터 데이터베이스로 교체하기 위해서 트리거, 계정 정보 등을 원본과 동일하게 변경해야 한다.

repl_change_master 유틸리티는 배포 데이터베이스의 마스터 데이터베이스에 대한 정보를 기준으로 지정된 슬레이브 데이터베이스를 마스터 데이터베이스로 변환시킨다.

구문

```
repl_change_master servers_info_file
```

- `servers_info_file` 파일 규격 : 라인별로 배포 데이터베이스 정보 기록(dist_name dist_host dist_passwd)

예제

```
distdb1 192.168.2.100 admin
distdb2 192.168.2.200 admin
```

복제 상태 관리

복제 상태 확인

복제가 진행되고 있는지 상태를 확인하기 위해서 **cubrid repl_server status** 나 **cubrid repl_agent status**를 사용한다.

- 복제 서버 혹은 복제 에이전트의 상태만 출력하여 확인한다.

```
$ cubrid repl_server status
```

또는

```
$ cubrid repl_agent status
@ cubrid replication status
repl_agent distdb (rel 8.1, pid 22203)
repl_server masterdb (rel 8.1, pid 12341)
```

- CUBRID의 모든 프로세스의 상태를 출력하여 복제 부분을 확인한다.

```
$ cubrid service status
@ cubrid master status
++ cubrid master is running.
@ cubrid server status
Server slavedb (rel 8.1, pid 11325)
Server distdb (rel 8.1, pid 31440)
Server masterdb (rel 8.1, pid 29191)
@ cubrid broker status
=====
NAME          PID  PORT  AS  JQ      REQ TPS AUTO  SES SQLL CONN
=====
* query editor 12149 30300  5   0        0 ---  ON  OFF ON:A AUTO
* broker1      12161 33300  5   0        0 ---  ON  OFF ON:A AUTO
@ cubrid manager server status
++ cubrid manager server is running.
@ cubrid replication status
repl_agent distdb (rel 8.1, pid 22203)
repl_server masterdb (rel 8.1, pid 12341)
```

복제 성능 모니터링

복제 성능 모니터링은 별도의 유틸리티를 제공하지 않고 배포 데이터베이스 생성 시 정의한 추적 로그 생성 디렉터리에 생성되는 성능 로그 파일 형태로 지원한다.

성능 로그는 트랜잭션이 마스터 데이터베이스에서 커밋된 시점부터 슬레이브 데이터베이스로 변영된 시점까지 소요된 시간을 기록하고 있다. 단, 마스터 서버와 슬레이브 서버 서로 간의 시스템 시각이 동기화된 경우에 유효하다. 서로 시간이 동기화되지 않으면 지연시간 로그에 왜곡이 발생한다. 지연 시간은 슬레이브 데이터베이스 별로 설정된 `perf_poll_interval` 파라미터에 의해 결정된다.

다음은 배포 데이터베이스 이름이 `distdb`라고 가정하고 `tail` 명령을 통해 성능 로그를 확인해 본 예이다.

```
$ tail -f distdb.perf
-----
No. master_db_name      tran_index      master_time      slave_time      delay
-----
001      mdb              8 2007/10/17 10:03:24 2007/10/17 10:03:24 0
002      mdb             -1 ----/--/-- --:--:-- 2007/10/17 10:04:07 0
003      mdb             10 2007/10/17 10:04:07 2007/10/17 10:04:07 0
004      mdb             11 2007/10/17 10:04:07 2007/10/17 10:04:07 0
005      mdb             -1 ----/--/-- --:--:-- 2007/10/17 10:04:07 0
```

중간에 `tran_index`가 -1로 출력되는 부분은 복제가 현재 시점까지 동기화되었다는 표시이다.

복제 파라미터

복제 관련 파라미터는 다음과 같다.

복제 파라미터

파라미터	기본 값	설명	설정 방식	관련 프로세스
replication	no	yes 로 설정된 경우 복제 로그 생성	cubrid.conf 파일 수정	<code>\$CUBRID/bin/cub_server</code>
error_log	NULL	복제 에이전트의 오류 로그를 저장할 디렉터리 경로	repl_make_distdb 유틸리티의 입력으로 지정	<code>\$CUBRID/bin/repl_agent</code>
trail_log	NULL	복제 에이전트의 추적 로그를 저장할 디렉터리 경로	repl_make_distdb 유틸리티의 입력으로 지정	<code>\$CUBRID/bin/repl_agent</code>
copylog_path	NULL	복제 에이전트의 복제 로그를 저장할 디렉터리 경로	repl_make_distdb 유틸리티의 입력으로 지정	<code>\$CUBRID/bin/repl_agent</code>
agent_port	NULL	<code>repl_check_sync</code> 로 <code>repl_agent</code> 의 정보를 알아올 때 사용되는 소켓 포트 번호	repl_make_distdb 유틸리티의 입력으로 지정	<code>\$CUBRID/bin/repl_agent</code>
perf_log_size	10000	.perf(복제 지연 시간을	repl_make_distdb	<code>\$CUBRID/bin/</code>

		기록하는 파일)의 크기 (라인 수)	유틸리티의 입력으 로 지정	repl_agent
commit_interval_msecs	500	repl_agent 가 슬레이브 에 복제 정보를 commit 하는 주기(단위: msec). repl_agent 는 슬레이브 데이터베이스를 복제할 때 각 트랜잭션의 정보 를 적용하지만 commit 은 하지 않는다. 이 옵션 에 설정된 시간 간격으 로 commit 을 수행한다.	repl_make_distdb \$CUBRID/bin/ 유틸리티의 입력으 로 지정	repl_agent
retry_connect	n	네트워크가 단절되어 repl_agent 가 종료되면, repl_agent 를 재시작할 지 여부	repl_make_distdb \$CUBRID/bin/ 유틸리티의 입력으 로 지정	repl_agent

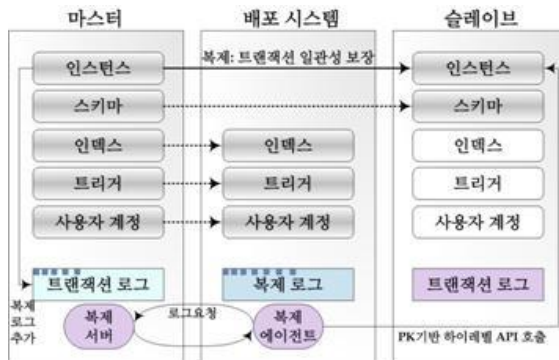
복제 내부 구현

복제 구현 원칙 및 아키텍처

CUBRID의 복제 시스템 설계 및 구현 원칙은 다음과 같다.

- 트랜잭션의 일관성이 보장되어야 한다.
- 슬레이브 데이터베이스의 다양한 활용을 위해 슬레이브의 스키마 독립성이 제공되어야 한다. 즉, 슬레이브 데이터베이스에 별도의 테이블이나 인덱스를 구축할 수 있어야 한다.
- 복제로 인해 마스터 데이터베이스에 미치는 영향을 최소화 해야 한다.
- 실시간 복제가 가능해야 한다.

따라서 CUBRID의 복제 시스템은 트랜잭션 로그 기반으로 동작하며 마스터 서버와는 별개로 복제 서버가 트랜잭션 로그를 복제 에이전트에게 전달하고 복제 에이전트가 트랜잭션 로그를 분석하여 그 결과를 슬레이브 데이터베이스에 전달하는 구조로 설계되었다.



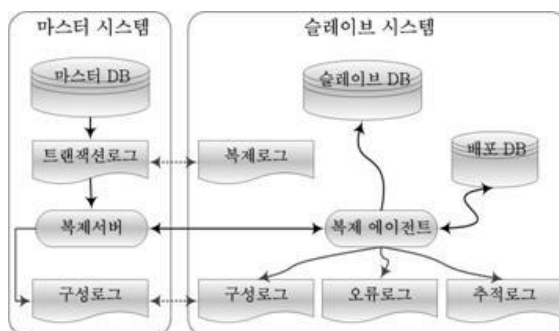
인스턴스와 스키마 정보는 슬레이브 시스템에 실시간으로 변경 내역이 반영된다. 인덱스, 트리거, 사용자 계정은 슬레이브의 스키마 독립성을 위해 백업 데이터베이스에 변경 내용만 기록하고 슬레이브 데이터베이스에는 반영되지 않는다. 마스터 데이터베이스 시스템에 장애가 생겨 슬레이브 데이터베이스를 마스터 데이터베이스로 교체할 경우 백업 데이터베이스에 기록된 마스터 데이터베이스의 최종 데이터, 스키마, 인덱스는 자동으로 동기화 된다. 나머지 트리거, 사용자 계정은 자동으로 동기화 되지 않는다.

복제 서버 및 에이전트 구조

복제 서버는 다수의 복제 에이전트로부터 트랜잭션 로그의 전달 요청을 받고 트랜잭션 로그를 복제 에이전트로 전달한다. 복제 서버는 요청을 처리하는 스레드, 트랜잭션 로그를 디스크로부터 읽는 스레드로 구성된다.

복제 에이전트는 복제 서버를 통해 전달받은 트랜잭션 로그를 로컬 디스크에 저장하고 이를 분석하여 슬레이브 데이터베이스에 반영한다. 복제 에이전트는 트랜잭션 로그를 받아 버퍼에 저장하는 스레드, 버퍼에 저장된 로그 데이터를 복제 로그로 쓰는 스레드, 그리고 슬레이브 데이터베이스에 변경 항목을 반영하는 스레드로 구성된다.

마스터 데이터베이스 하나를 N개의 슬레이브 데이터베이스로 동시에 복제할 수 있다.



배포 데이터베이스 스키마

배포 데이터베이스는 복제에 필요한 메타 데이터를 관리하기 위한 데이터베이스이다. 마스터 및 슬레이브 데이터베이스 정보, 추적 정보, 복제 파라미터, 슬레이브 데이터베이스를 자동으로 마스터 데이터베이스로 교체하는 기능 제공을 위한 정보로 구성된다.

클래스 이름: db_info

db_info는 master_info 및 slave_info의 수퍼 클래스로서 공통으로 필요한 항목을 관리한다.

속성 이름	타입	설명	비고
dbid	integer	데이터베이스 고유 식별자	기본키
dbname	varchar(126)	데이터베이스 이름	
master_ip	varchar(50)	데이터베이스 서버의 IP 주소	
portnum	integer	복제 서버의 TCP 포트 번호	

클래스 이름: master_info

master_info 클래스에서는 복제하려고 하는 원본 데이터베이스의 데이터베이스 이름, 복제 서버에 접속하기 위한 IP 주소 및 TCP 포트 번호, 복제 로그 파일의 디렉터리 경로 등의 정보를 관리한다. repl_make_distdb 유틸리티를 수행하여 이 값을 지정한다.

속성 이름	타입	설명	비고
dbid	integer	데이터베이스 고유 식별자	db_info 상속
dbname	varchar(126)	데이터베이스 이름	db_info 상속
master_ip	varchar(50)	복제 서버의 IP 주소	db_info 상속
portnum	integer	cubrid.conf 의 cubrid_port_id 사용	db_info 상속
copylog_path	varchar(256)	복제 로그를 저장할 파일 경로	
size_of_log_buffer	integer	repl_agent 가 트랜잭션 로그를 임시로 저장하는 버퍼의 수를 지정. 한 버퍼의 크기는 데이터베이스 페이지 크기와 동일.	디폴트 값 : 500
size_of_cache_buffer	integer	repl_agent 가 복제 로그를 임시로 저장하는 버퍼의 수를 지정. 한 버퍼의 크기는 데이터베이스 페이지 크기와 동일.	디폴트 값 : 100
size_of_copylog	integer	복제 로그의 페이지 수를 정의. 만약 지정된 페이지 수 이상인 경우 이미 처리 완료된 페이지는	디폴트 값 : 5000

truncate 됨.			
start_pageid	integer	복제 로그의 첫 페이지 번호	사용 안 함
first_pageid	integer	활성 복제 로그의 첫 페이지 번호	
last_pageid	integer	복제 로그의 마지막 페이지 번호	

클래스 이름: slave_info

slave_info 클래스에서는 복제 대상 슬레이브 데이터베이스의 데이터베이스 이름, 복제 계정 및 암호 등을 관리한다. **repl_make_slavedb** 유틸리티를 수행하여 이 값을 지정한다.

속성 이름	타입	설명	비고
dbid	integer	데이터베이스 고유 식별자	db_info 상속
dbname	varchar(126)	데이터베이스 이름	db_info 상속
master_ip	varchar(50)	데이터베이스 서버의 IP 주소	db_info 상속
portnum	integer	복제 서버의 TCP 포트 번호	db_info 상속
userid	varchar(32)	슬레이브 데이터베이스의 복제 계정	
passwd	varchar(32)	슬레이브 데이터베이스의 복제 계정 암호	
trails	set_of(trail_info)	슬레이브 데이터베이스에 반영된 최종 로그 번호	repl_make_slavedb 유틸리티 수행 시 초기화됨

클래스 이름: trail_info

trail_info 클래스에서는 복제할 마스터와 슬레이브 관계 정보 및 복제를 시작할 로그 번호 등을 관리한다. **repl_make_slavedb** 유틸리티를 수행하여 이 값을 지정한다.

속성 이름	타입	설명	비고
master_dbid	integer	마스터 데이터베이스의 식별자	
slave_dbid	integer	슬레이브 데이터베이스의 식별자	
final_pageid	integer	슬레이브 데이터베이스에 최종으로 반영된 로그 페이지 번호	
final_offset	integer	슬레이브 데이터베이스에 최종으로 반영된 로그 페이지 위치	
all_repl	char(1)	모든 클래스의 복제 여부. Y 또는 y : 설정	현재는 for_recovery 와

		N 또는 n : 설정 안 함	같은 방식으로 동작. 추후 구분 예정.
repl_count	number(15,0)	복제한 트랜잭션 개수	갱신된 정보는 repl_check_sync 명령어를 통해 확인할 수 있음.
status	char(1)	현재 repl_agent 의 상태 정보. F : 복제를 구성하고 에이전트를 한번도 실행하지 않은 상태 A : 복제 실행 중 I : idle 상태	갱신된 정보는 repl_check_sync 명령어를 통해 확인할 수 있음.
error_msg	string	사용 안 함	
stop_time	timestamp	사용 안 함	
perf_poll_interval	integer	복제 지연 시간 측정을 설정하는 파라 디폴트 값 : 10 미터(단위 : 초). 허용 범위 : 10 초에서 60 초	
log_apply_interval	integer	사용 안 함	
for_recovery	integer	마스터 데이터베이스에 장애 발생 시 마스터 데이터베이스로 대체할 목적으로 복제를 구성할 경우 설정한다. 이 값이 y 이면 index_replication 값도 y 로 설정되며 마스터의 모든 클래스가 복제된다. Y 또는 y : 설정 N 또는 n : 설정 안 함	현재는 all_repl 과 같은 방식으로 동작.
index_replication	char(1)	인덱스를 자동으로 복제하고자 할 때 설정. Y 또는 y : 설정 N 또는 n : 설정 안 함	디폴트 값 : n
restart_interval	integer	repl_agent 가 배포 데이터베이스에 접속하여 정보를 반영하는 간격(단위: 초).	디폴트 값 : 100

복제 제약 사항

- 복제 기능은 Linux 계열에서만 지원한다.

- 마스터 데이터베이스와 슬레이브 데이터베이스는 반드시 동일 플랫폼으로 구성해야 한다. 예를 들어 마스터 데이터베이스는 Linux, 슬레이브 데이터베이스는 Solaris로 구축할 수 없다.
- 기본 키가 설정된 테이블에 대해서만 복제할 수 있다.
- 객체 타입을 속성으로 갖는 테이블의 경우에는 복제 시 오류가 발생할 수 있다. 객체 타입을 복제에서 사용하려면 외래키 ON CACHE OBJECT를 이용할 수 있다. 자세한 설명은 [테이블 생성의 외래 키\(foreign key\) 제약](#)을 참조한다.
- 마스터 데이터베이스의 모든 테이블에 기본키가 설정되어 있어야만 마스터 데이터베이스를 슬레이브 데이터베이스로 교체할 수 있다.
- 조건 1과 조건 2를 동시에 만족하는 **UPDATE** 문을 실행할 수 없다. 이 경우 에러는 아래와 같다.
 - 조건 1: unique 제약이 있는 속성을 **UPDATE**
 - 조건 2: 여러 레코드를 **UPDATE**

```
// 영문
Current version of replication does not allow changing multiple rows with a single UPDATE
statement which can violate the UNIQUE constraint.

// 한글
현재 버전의 복제는 하나의 UPDATE 질의로 여러 레코드를 변경시키는 것이 UNIQUE 제약을 위반하는 경우
허용되지 않습니다.
```

복제 주의 사항

- 마스터 데이터베이스의 변경 사항은 대부분 실시간으로 슬레이브 데이터베이스에 반영되나, 한번에 다량의 데이터를 갱신하는 긴 트랜잭션의 경우 복제 지연 시간이 증가할 수 있다.
- 복제가 지연될 경우 복제 서버가 마스터 데이터베이스의 트랜잭션 보관 로그에서 데이터 변경 항목을 읽어야 하므로 가급적 5개에서 10개 정도의 트랜잭션 보관 로그는 삭제하지 않는 것이 좋다.
- **repl_make_slavedb**를 수행하여 슬레이브 데이터베이스를 새로 구축하는 경우에는 복제 에이전트 수행 전에 반드시 이전 복제 구성에서 사용했던 복제 로그, 추적 로그와 같은 로그 파일을 삭제해야 한다.
- **TIMESTAMP**는 슬레이브 데이터베이스에 반영된 시각이 아니라 마스터와 동일한 값으로 복제된다.
- 마스터 데이터베이스 서버가 장애로 중지된 경우에는 복제 서버나 복제 에이전트는 중단되지 않고 장애 발생 이전에 변경된 데이터를 슬레이브 시스템에 복제한다. 그러나 슬레이브 데이터베이스 서버가 장애로 중지된 경우에는 복제 에이전트도 같이 중지된다. 따라서 슬레이브 데이터베이스 서버가 재시작한 후 수동으로 복제 에이전트를 재시작해야 한다.
- 복제가 진행되면서 생기는 복제 로그 및 첫 번째 복제 보관 로그, 추적 로그는 반드시 손상되지 않도록 유의해야 한다. 복제 에이전트 수행 시 **-ar** 옵션을 주지 않으면 첫번째 복제 보관 로그인 **<master_db_name>.copy.ar0**만 생성하고 그 이후로는 복제 보관 로그를 생성하지 않는다. 복제 보관 로그는 이미 반영된 트랜잭션 로그를 보관하기 위한 파일로, 더이상 복제 에이전트가 사용하지 않으므로 보관할 필요가 없으나, 첫 번째 복제 보관 로그는 복제에 장애가 발생한 경우 복구를 위해 필요하므로 삭제하지 않도록 주의한다.
- 배포 데이터베이스에 접속하여 수작업으로 데이터 항목을 수정하지 않을 것을 권장한다.

CUBRID HA

개요

High Availability(HA)란, 하드웨어, 소프트웨어, 네트워크 등에 장애가 발생하여도 지속적인 서비스를 제공해주는 기능이다. 이 기능은 하루 24시간 1년 내내 서비스를 제공해야 하는 네트워킹 컴퓨팅 분야에서 필수적인 요소이다. HA시스템은 두 대 이상의 서버 시스템으로 구성하여 시스템 구성 요소 중의 한 요소에 장애가 발생하여도 중단 없는 서비스를 제공할 수 있다.

High Availability 기능을 CUBRID에 적용한 것이 CUBRID HA 기능이며, CUBRID HA 기능은 여러 서버 시스템에서 데이터베이스를 항상 동기화된 상태로 유지하여 서비스를 제공한다. 또한 서비스를 수행중인 시스템에 예상치 못한 장애가 발생하였을 경우에는 자동으로 다른 시스템이 서비스를 수행하도록 하여 서비스 중단 시간을 최소화한다.

CUBRID의 HA 기능은 shared-nothing 구조이며, 마스터 데이터베이스 서버로부터 슬레이브 데이터베이스 서버로의 데이터 동기화를 위해 다음 두 단계를 수행한다.

- 마스터 데이터베이스 서버에서 생성되는 트랜잭션 로그를 실시간으로 다른 node에 복제하는 트랜잭션 로그 다중화 단계
- 실시간으로 복제되는 트랜잭션 로그를 분석하여 슬레이브 데이터베이스 서버로 데이터를 반영하는 트랜잭션 로그 반영 단계

CUBRID HA 기능은 이 두 단계를 수행하여 마스터 데이터베이스와 슬레이브 데이터베이스 사이에 항상 동기화된 데이터를 유지할 수 있도록 한다. 따라서, 서비스를 제공 중이던 마스터 데이터베이스 서버에 예상치 못한 장애가 발생하였을 경우 슬레이브 데이터베이스 서버가 마스터 데이터베이스 서버를 대신하여 중단 없는 서비스를 제공할 수 있다. CUBRID HA 기능은 시스템과 CUBRID의 상태를 실시간으로 감시하고 장애 발생 시 자동으로 절체(failover)를 수행하기 위해 Heartbeat을 사용한다.

참고 CUBRID 2008 R2.0, R2.1에서는 Linux Heartbeat 솔루션 기반의 HA 기능을 지원했고, CUBRID 2008 R2.2부터는 자체 개발한 CUBRID Heartbeat 기반의 HA 기능을 지원한다.

용어 정의

마스터 데이터베이스

복제의 대상이 되는 원본 데이터베이스이며 마스터 데이터베이스는 읽기, 쓰기 등 모든 데이터베이스 연산이 허용된다.

슬레이브 데이터베이스

마스터 데이터베이스와 동일한 내용의 복제된 데이터베이스(replica)이며, 마스터 데이터베이스의 변경이 자동적으로 반영되는 데이터베이스이다. 마스터 데이터베이스와 달리 슬레이브 데이터베이스는 읽기 연산만 사용될 수 있다.

Active 서버

'primary 서버'라고도 하며, 사용자에게 서비스를 제공하는 서버이다. active 서버는 마스터 데이터베이스를 사용한 읽기, 쓰기 등 모든 서비스를 제공한다.

Standby 서버

'secondary', 'passive' 혹은 'failover 서버'라고 하며, active 서버가 장애로 인해 서비스를 할 수 없을 때, active 서버를 대신하여 서비스를 제공할 수 있다. standby 서버는 슬레이브 데이터베이스를 사용한 읽기 서비스를 제공한다.

Failover

active 서버 또는 active 서버가 존재하는 시스템에 장애가 발생하여 서비스를 제공할 수 없는 상태가 검출되면, 자동으로 standby 서버를 active 서버로 절체하여 서비스를 계속 제공하는 기능이다.

Failback

failover 이후 active 서버가 장애 이전의 상태로 복구되면, 복구된 active 서버를 통하여 서비스가 수행되도록 원래의 상태로 다시 절체되는 기능이다.

Role change

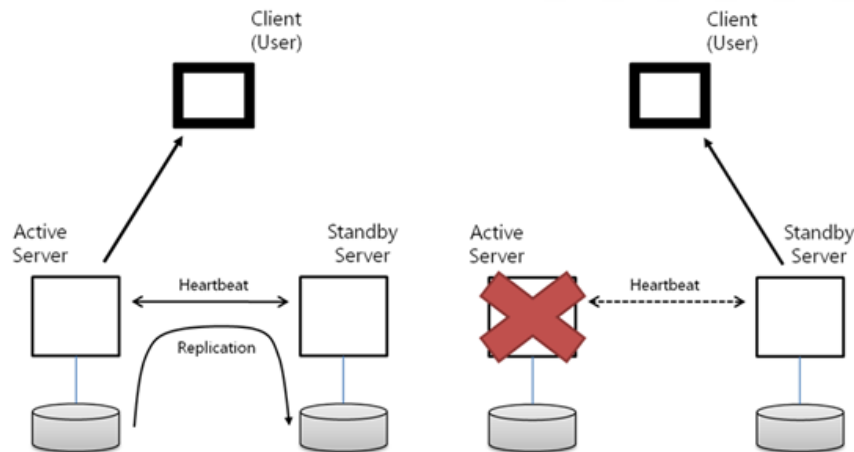
이전 active 서버에서 장애가 복구되어도 현재의 상태로 서비스를 계속 제공하는 기능이다.

Heartbeat

HA 기능을 제공하기 위한 핵심 구성 요소이다. CUBRID Heartbeat 기능은 cub_master 프로세스 내에 포함되며, 다른 노드의 cub_master 프로세스와 heartbeat 메시지를 주고 받으며 노드 장애를 감지하고, 장애가 감지되면 standby 서버로 failover를 수행한다. 또한, HA 관련 프로세스(cub_server, copylogdb, applylogdb)의 가용 상태를 주기적으로 모니터링한다.

서버 이중화(Duplex, Duplicate)

HA기능을 제공하기 위해 서버 HW를 중복으로 구성하여 시스템을 구축하는 것이다. 장애 발생시 standby 서버가 active 서버의 기능을 수행하는 방식(Active-Standby, 아래 그림)과 장애 발생 서버의 역할을 추가로 대행하면서 서비스 기능을 수행하도록 이중 체계를 구축하는 방식(Active-Active)이 있다.



데이터베이스 서버 다중화 구조

하나의 데이터베이스에 장애가 발생하여도 서비스를 계속 제공할 수 있도록 여러 대의 데이터베이스 서버를 구성한 구조이다. 서비스를 제공 중이던 active 데이터베이스 서버에 장애가 발생하면, 동일한 데이터를 가지고 있는 standby 데이터베이스 서버가 서비스를 제공할 수 있다.

브로커 다중화 구조

브로커를 다중화 하여 특정 브로커에 장애가 발생하여도 다른 브로커를 통한 서비스를 계속 제공할 수 있도록 구성한 구조이다. 또한 각각의 브로커는 아래에서 설명하는 것과 같이 다른 특성을 가질 수 있다.

- **Read-only broker** : read only 브로커는 읽기 연산만을 수행하는 브로커이며 standby 서버와의 연결을 통해 서비스를 제공한다. standby 서버가 존재하지 않는 경우 active 서버로 읽기 요청을 보낼 수 있다. 즉, 데이터베이스 서버와의 연결을 시도하는 순서는 standby 서버 우선이며 모든 standby 서버와의 연결 실패 시, active 서버와 연결될 수 있다.
- **Slave-only broker** : read-only broker와 달리 slave-only broker는 standby 서버로만 요청을 보낼 수 있다. standby 서버가 존재하지 않는 경우 active 서버로 연결을 시도하지 않는다.

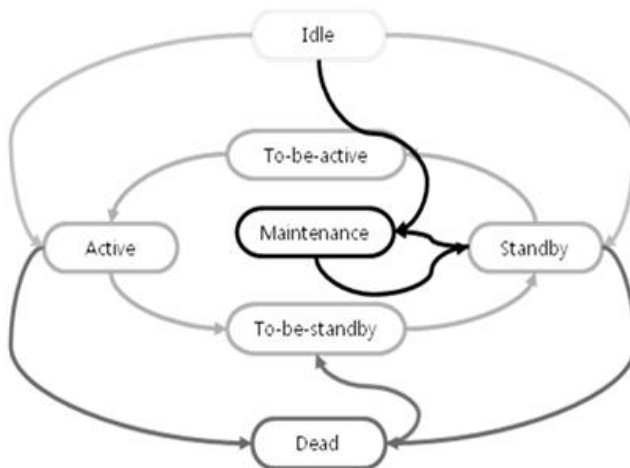
트랜잭션 로그 다중화(Transaction Log Multiplication)

active 서버에서 생성되는 트랜잭션 로그가 하나 이상의 standby 서버에 실시간으로 전송하여 모든 서버에 동일한 로그가 기록되도록 하는 기능이다.

CUBRID 데이터베이스 서버의 HA 모드

- **active** : 일반적인 읽기 및 쓰기 요청에 대한 서비스를 제공하며, 복제에 필요한 트랜잭션 로그를 생성하는 상태이다..

- **to-be active** : standby 상태의 서버를 active 모드로 변경하면 to-be-active 단계를 거쳐 active가 된다. to-be-active상태이면 유입되는 요청을 중지(suspend)하고 반영되지 않은 복제 로그를 반영한 후 active 상태로 변경된다.
- **standby** : 읽기 요청에 대한 서비스만 제공하는 상태로 쓰기 요청은 거부(deny)한다.
- **to-be standby** : 서버를 standby로 모드로 변경하면 to-be-standby 단계를 거쳐 standby가 된다. to-be-standby 모드이면 유입되는 요청을 거부(deny)하고 수행 중인 트랜잭션이 완료되면 standby 상태로 변경된다.
- **maintenance** : 데이터베이스의 운영 작업(스키마 변경, 설정 변경 등)을 하기 위한 모드이다. 일시적으로 해당 데이터베이스 서버를 HA 구성에서 제외시킨 다음 데이터베이스 서버를 maintenance 모드로 실행시켜서 필요한 운영 작업을 할 수 있다. 상태 동작 방식은 다음과 같다.
 - 로컬 호스트의 클라이언트만 접속할 수 있지만, 트랜잭션 로그를 복사 또는 적용하는 copylogdb와 applylogdb 유틸리티는 접속할 수 없다.
 - 쓰기 연산을 통하여 데이터베이스를 변경할 수 있지만, 변경에 대한 복제 로그를 생성하지 않는다.



- failover 시 데이터베이스 서버의 상태 변화
 - active 서버: active -> dead
 - standby 서버: standby -> to-be-active -> active

HA 제약 사항

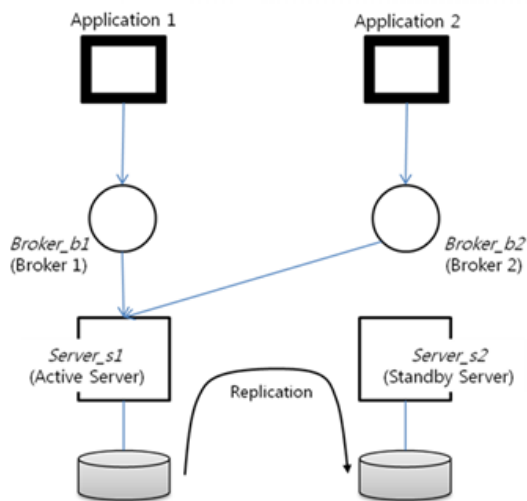
- Linux 계열에서만 CUBRID HA 기능을 사용할 수 있다.
- active 서버와 standby 서버는 반드시 동일 플랫폼으로 구성해야 한다.
- 기본 키(primary key)가 설정된 테이블에 대해서만 복제가 가능하다.
- 다음의 경우 마스터 데이터베이스와 슬레이브 데이터베이스의 데이터가 일치하지 않을 수 있다.
- HA 구성에서 테이블에 트리거(trigger)가 설정된 경우 : 마스터 데이터베이스에서 이미 수행된 트리거가 슬레이브 데이터베이스에서 중복 수행된다.

- HA 구성에서 테이블에 메소드(method)가 사용된 경우 : 메소드 사용 시 복제 로그가 생성되지 않는다.
- HA 구성에서 CUBRID 매니저를 이용하여 작업이 수행된 경우 : CUBRID 매니저에서 특정 작업 수행 시 복제 로그가 생성되지 않는다.
- HA 구성에서 독립 실행 모드(standalone)에서 작업이 수행된 경우 : 독립 실행 모드는 서버 프로세스가 구동되어 있지 않은 경우에 DB 작업을 수행할 수 있는 모드이기 때문에, 독립 실행 모드에서 수행한 작업에 관한 로그가 슬레이브 데이터베이스로 반영되지 않는다.

HA 환경 설정

HA 구성을 위해서는 active 서버 및 standby 서버 각각에서 CUBRID heartbeat 기능이 활성화되어야 한다. 브로커와 데이터베이스 서버를 하나의 장비에서 구성할 수 있으며, 이를 각각 분리된 시스템에 구성할 수도 있다.

HA 기능을 사용하기 위해서는 다음과 같이 설정한다.



노드 그룹 식별자 생성

active 및 standby 서버에서 동일한 사용자 계정(예: ha_user1)을 각각 생성하고, 해당 계정에서 CUBRID를 실행한다. 생성된 UNIX 사용자 계정이 HA 모드로 구동될 서버 노드 그룹 식별자가 된다.

HA 구동 스크립트 준비

HA 구동 스크립트는 **CUBRID/share/init.d/cubrid-ha**에 위치하며, 이를 active 및 standby 서버(/etc/init.d/)에 각각 준비한다.

하나의 호스트에서 여러 사용자 계정으로 HA를 구성하고 이를 시스템 서비스에 등록하려면, HA 구동 스크립트의 이름을 **cubrid-ha-{unix user id}**로 변경하고 각 스크립트를 **/etc/init.d/** 디렉터리에 복사한다. 시스템 서비스에 등록하려면 root 권한이 필요하다.

```
[root@server_s1 init.d]# cd /home/ha user1/CUBRID/share/init.d/
[root@server_s1 init.d]# cp cubrid-ha /etc/init.d/
```

데이터베이스 복사

active 서버에 DB(예: tdb01)를 새로 생성하고, 이를 standby 서버로 복사한다. 데이터 볼륨 복사, 백업/복구, **unloaddb/loaddb** 중 하나의 방법을 이용할 수 있다.

서버 설정(cubrid.conf)

cubrid.conf 설정 파일에 HA 관련 파라미터를 추가해야 한다. 이때, HA 관련 파라미터는 반드시 **[common]** 섹션에 설정해야 하며, HA 모드로 구동하지 않을 데이터베이스에 대해서만 **[@ <database>]** 섹션에서 **ha_mode**의 값을 no로 설정한다. 만약, **[common]** 섹션에 **ha_mode**의 값이 no이고, **[@ <database>]** 섹션에 **ha_mode**의 값이 yes인 경우에는 에러가 출력된다.

- **ha_mode** : HA 기능을 설정하기 위한 파라미터이며, 디폴트 값은 **off**이다. 이 파라미터 값을 **on**으로 변경한다.
- **ha_port_id** : UDP 포트를 설정하는 파라미터이며, 이 포트를 통해 각각의 cub_master 프로세스가 heartbeat 메시지를 주고 받으며 노드 장애를 감지한다.
- **ha_node_list** : HA 모드로 구동되는 노드 그룹 식별자 및 그룹에 속하는 멤버 노드의 호스트명을 명시하는 파라미터이다. 파라미터 값으로 명시한 멤버 노드의 호스트명 및 현재 노드의 호스트명은 **/etc/hosts**에 등록되어 있어야 한다. 아래와 같이 ha_user1이라는 노드 그룹의 멤버로 active 서버의 호스트명(server_s1)과 standby 서버의 호스트명(server_s2)을 명시한다.

```
#cubrid.conf
[common]
...
ha_mode = on
ha_port_id = 41523
ha_node_list = ha_user1@server_s1:server_s2
```

HA 구동 스크립트 설정

active 및 standby 데이터베이스 서버에 각각 준비된 HA 구동 스크립트 파일에서 아래 설정을 추가한다.

- **CUBRID_USER** : 노드 그룹 식별자, 즉 CUBRID를 설치 및 실행한 UNIX 사용자 계정을 명시한다.
- **DB_LIST** : HA 모드로 구동할 데이터베이스 이름을 명시한다. 한 개 이상의 데이터베이스 이름을 명시하는 경우, 공백으로 구분한다.

```
#cubrid-ha
CUBRID_USER = ha user1
DB_LIST = 'tdb01'
# DB_LIST = 'tdb01 tdb02 tdb03'
```

브로커 설정

브로커의 기본 동작 모드는 읽기 및 쓰기 연산을 요청하는 모드이며, 필요 시 **cubrid_broker.conf**의 **ACCESS_MODE** 파라미터를 이용하여 Read only 브로커나 Slave only 브로커로 설정할 수 있다.

```
#cubrid broker.conf
ACCESS_MODE = RW|RO|SO

RW := Read-Write broker (기본값)
RO := Read-Only broker
SO := Slave-Only broker
```

구동 중인 브로커의 동작 모드는 **cubrid broker status** 유틸리티의 **-f** 인자를 사용하여 확인할 수 있다.

브로커 상태에 대한 자세한 내용은 [브로커 상태 확인](#)을 참고한다.

```
% broker1 - cub cas [4430,40821] /home/CUBRID/log/broker/broker1.access /home/CUBRID/
JOB_QUEUE:0, AUTO_ADD_APPL_SERVER:ON, SQL_LOG_MODE:ALL:100000
LONG_TRANSACTION_TIME:60, LONG_QUERY_TIME:60, SESSION_TIMEOUT:300
KEEP_CONNECTION:AUTO, ACCESS_MODE:RW
-----
ID   PID   QPS   LQS PSIZE STATUS          LAST ACCESS TIME   DB      HOST   LAST
CONNECT TIME          CLIENT IP
-----
1 26946   0     0 51168 IDLE           2009/11/06 16:06:41 -        -
-        10.0.1.101
2 26947   0     0 51172 IDLE           2009/11/06 16:06:41 -        -
-        10.0.1.101
3 26948   0     0 51172 IDLE           2009/11/06 16:06:41 -        -
-        10.0.1.101
4 26949   0     0 51172 IDLE           2009/11/06 16:06:41 -        -
-        10.0.1.101
5 26950   0     0 51172 IDLE           2009/11/06 16:06:41 -        -
-        10.0.1.101
```

데이터베이스 호스트 정보 설정

HA 구성에서 사용할 Active 서버 및 Standby 서버의 호스트 정보를 브로커 및 서버 시스템 내 데이터베이스 위치 정보 파일(**databases.txt**)에 추가해야 한다. Active 서버와 Standby 서버는 콜론(:)으로 구분하며, Standby 서버는 한 개 이상 추가 가능하고 각 호스트들 간의 구분 또한 콜론(:)으로 구분한다. 이처럼 동일한 데이터베이스에 대해 여러 개의 호스트를 지정하는 환경에서는, 클라이언트 서버 모드(CS 모드)로 **csql**, **backupdb**, **applylogdb** 등의 유틸리티 수행 시 대상 데이터베이스 이름 뒤에 반드시 호스트 이름(@host)을 지정해야 한다. 독립 모드(SA 모드)로 유틸리티를 수행하는 경우에는 대상 데이터베이스 이름 뒤에 호스트 이름을 지정할 수 없다.

```
#databases.txt
#db-name      vol-path      db-host      log-path
tdb01         /home/db/db2   server s1:server s2  /home/db/db2
tdb02         /home/db/db1   server_s1:server_s2  /home/db/db1
```

HA 모드 구동

cubrid-ha start

cubrid-ha start는 HA 모드로 데이터베이스 서버를 구동하는 명령어이며, cub_master, cub_server, copylogdb, applylogdb 프로세스를 순서대로 구동하며 복제 로그(copylog)가 저장되는 경로를 생성한다.

cubrid-ha start 명령이 수행되는 순서에 따라 active 서버와 standby 서버가 결정된다.

먼저 구동된 서버 노드(server_s1)의 데이터베이스가 마스터 데이터베이스가 된다.

```
[root@server_s1 ~]# service cubrid-ha start
Starting cubrid-ha: [ OK ]
```

나중에 구동된 서버 노드(server_s2)의 데이터베이스가 슬레이브 데이터베이스가 된다.

```
[root@server_s2 ~]# service cubrid-ha start
Starting cubrid-ha: [ OK ]
```

cubrid-ha status

cubrid-ha status는 HA 모드로 구동 중인 데이터베이스 서버의 상태 정보를 확인하는 명령어이다. 이 명령어가 수행된 HA 노드의 상태 정보 및 HA 관련 프로세스의 상태 정보가 출력된다.

```
[root@server_s2 ~]# service cubrid-ha status
HA-Node Info (current server_s2, state slave)
  Node server_s2 (priority 2, state slave)
  Node server_s1 (priority 1, state master)

HA-Process Info (master 30519, state slave)
  Applylogdb tdb01@localhost:/home1/cubrid1/DB/tdb01_server_s1 (pid 30796, state
registered)
  Copylogdb tdb01@server_s1:/home1/cubrid1/DB/tdb01_server_s1 (pid 30788, state
registered)
  Server tdb01 (pid 30551, state registered)

++ cubrid heartbeat list: success
Status cubrid-ha: [ OK ]
```

cubrid-ha stop

cubrid-ha stop은 HA 모드로 구동 중인 데이터베이스 서버를 중지하는 명령어이다. applylogdb, copylogdb, cub_server, cub_master 프로세스를 순서대로 중지하며, 관련 프로세스가 모두 종료되므로 주의해야 한다.

한쪽 노드(server_s1)에서만 수행되면 다른 노드(server_s1)로 failover하고, 양쪽 노드(server_s1, server_s2)에서 각각 수행되면 서비스가 중단된다.

```
[root@server_s1 ~]# service cubrid-ha stop
Stopping cubrid-ha: [ OK ]
```

서비스를 중단하기 위해서는 나머지 노드(server_s2)에서 이 명령을 수행한다.

```
[root@server_s2 ~]# service cubrid-ha stop
Stopping cubrid-ha: [ OK ]
```

cubrid-ha deact

cubrid-ha deact는 서비스 중인 노드에서 배치 작업을 수행하기 위해 서비스 노드를 다른 노드로 failover할 때 사용되는 명령어로서, 서비스 중인 노드(server_s1)의 상태를 unknown으로 변경하고 standby 서버 노드(server_s2)로 자동 failover한다.

```
[root@server_s1 ~]# service cubrid-ha deact
deactivate cubrid-ha:                                [ OK ]

[root@server_s1 ~]# service cubrid-ha status

HA-Node Info (current server s1, state unknown)

HA-Process Info (master 13396, state unknown)

Status cubrid-ha:                                    [ OK ]
```

cubrid-ha act

cubrid-ha act는 **cubrid-ha deact**가 수행된 노드에서 배치 작업을 완료한 이후, 해당 노드를 서비스 노드로 복구하기 위하여 사용되는 명령어로서, 해당 노드(server_s1)의 상태를 unknown에서 active로 변경한다.

```
root@server_s1 ~]# service cubrid-ha act
activate cubrid-ha:                                [ OK ]
```

cubrid heartbeat deregister

cubrid heartbeat deregister는 현재 노드에서 실행 중인 특정 프로세스를 종료하고, 관리 프로세스 목록에서 이를 삭제하기 위해 사용되는 명령어이며, 반드시 대상 프로세스 ID가 명시되어야 한다. 이때, 대상 프로세스가 종료되더라도 standby 서버로 failover되지 않는다.

아래는 데이터베이스 리스트에 tdb01과 tdb02가 포함되어 HA 모드로 동작되는 경우에, 현재 노드에서 실행 중인 tdb01 서버 프로세스를 종료하고 프로세스 목록에서 삭제하는 예이다.

```
[ha user1@server_s1 ~]$ cubrid heartbeat deregister 4087
@ cubrid heartbeat deregister: 4087
HA-Process Info (master 4072, state master)
  Applylogdb tdb01@localhost:/home1/cubrid1/DB/tdb01 server s2 (pid 4307, state
registered)
  Applylogdb tdb02@localhost:/home1/cubrid1/DB/tdb02 server s2 (pid 4313, state
registered)
  Copylogdb tdb02@server_s2:/home1/cubrid1/DB/tdb01_server_s2 (pid 4311, state registered)
  Copylogdb tdb01@server_s2:/home1/cubrid1/DB/tdb01_server_s2 (pid 4305, state registered)
  Server tdb02 (pid 4195, state registered_and_active)
  Server tdb01 (pid 4087, state deregistered)
```

HA 유틸리티

HA 기능을 위해 사용되는 CUBRID 유틸리티는 다음과 같다. 아래 유틸리티는 **ha_mode** 파라미터가 **on**으로 설정된 서버에서만 사용할 수 있다.

데이터베이스 서버의 작동 모드 출력/설정

설명

cubrid changemode 유틸리티는 데이터베이스 서버의 상태를 출력하거나 변경한다. 서버의 작동 모드가 바뀔 때마다 서버 에러 로그에 이력이 출력된다. **-m** 옵션을 지정하지 않으면 현재의 작동 모드가 출력된다.

구문

```
cubrid changemode [option] <database_name>@<hostname>
option:
-m, --mode=<MODE> : 변경할 모드를 지정한다. 사용할 수 있는 값은 active, standby, maintenance이다.
```

예제 1

```
[ha_user1@server_s1 ~]$ cubrid changemode tdb01@server_s1
The server 'tdb01@server_s1's current HA running mode is active.

[ha_user1@server_s1 ~]$ cubrid changemode tdb01@server_s2
The server 'tdb01@server_s2's current HA running mode is standby.
```

예제 2

;database 명령어를 수행하여 CSQL 세션 내에서 현재 접속 중인 데이터베이스 서버의 HA 모드를 확인할 수 있다.

```
csql> ;database
demodb@localhost (active)
```

데이터베이스 서버의 트랜잭션 로그 저장

설명

cubrid copylogdb 유틸리티는 원격 데이터베이스 서버에서 생성되는 트랜잭션 로그를 지정한 경로에 저장한다.

트랜잭션 로그의 전송 방식은 다음 세 가지이며, 사용자는 운영 정책에 따라 적합한 트랜잭션 로그 전송 방식을 선택하여 사용한다.

- **동기식(Synchronous)** : 데이터베이스 서버는 copylog 프로세스에 트랜잭션 로그를 모두 전송하고, 전송된 트랜잭션 로그가 디스크에 저장(write)될 때까지 기다린 후 커밋한다. 즉, 로그 페이지 전송과 로그 페이지 저장을 모두 보장하는 방식이다. **cubrid copylog -m**의 값을 **sync**로 지정한다.
- **반동기식(Semi-Synchronous)** : 데이터베이스 서버는 copylog 프로세스에 트랜잭션 로그를 모두 전송하고, 응답을 받으면 커밋한다. 즉, 로그 페이지 전송은 보장하나 로그 페이지 저장은 보장하지 않는 방식이다. **cubrid copylog -m**의 값을 **semisync**로 지정한다.
- **비동기식(Asynchronous)** : 데이터베이스 서버는 copylog 프로세스에 트랜잭션 로그를 전송한 직후 커밋한다. 즉, 로그 페이지 전송 완료를 보장하지 않는 방식이다. **cubrid copylog -m**의 값을 **async**로 지정한다.

구문

```
cubrid copylogdb [option] <database_name>@<hostname>
option:
-L, --log-path=<PATH> : 복사한 트랜잭션 로그를 저장할 파일의 경로이다.
-m, --mode=<MODE> : 트랜잭션 로그 페이지를 복사해 오는 방식을 지정한다. 지정 가능한 값은 sync, semisync 및
async이며, 모드에 따라 페이지 전송 작업 및 페이지 저장(write) 작업을 다르게 보장한다.
```

저장한 트랜잭션 로그를 데이터베이스에 반영

설명

cubrid applylogdb 유틸리티는 복사된 트랜잭션 로그 파일을 지정한 경로에서 읽어서 분석 후 로컬 데이터베이스 서버에 반영한다.

구문

```
cubrid applylogdb [option] <database_name>@<hostname>
option:
-L, --log-path=<PATH>: 읽을 트랜잭션 로그 파일의 경로이다.
--max-mem-size=<SIZE>: 프로세스의 최대 메모리 크기이다. 메모리 크기의 단위는 MB이고, 최대 1000MB까지
가능하다.
```

참고 현재 메모리 사용량에 따른 CAS 프로세스 재시작 조건은 [브로커별](#)

[파라미터](#)에서 **APPL_SERVER_MAX_SIZE** 파라미터의 설명을 참고한다. **applylogdb** 프로세스 재시작 조건은 **--max-mem-size** 옵션을 이용하여 설정할 수 있다. **applylogdb** 프로세스의 현재 메모리 사이즈가 **-max-mem-size** 옵션 값 또는 프로세스 시작 시 메모리 사이즈의 2배 값 중 큰 값에 도달하는 경우, **applylogdb** 프로세스가 재시작된다.

HA 관련 JDBC 환경 설정

기본 환경

- JDK 1.5 이상
- CUBRID 2008 R2.0 이상
- CUBRID JDBC Driver 2008 R2.0 이상

HA 환경에서 JDBC 연결 문자열

설명

JDBC에서 HA 기능을 사용하려면, 브로커 장애 시 연결할 브로커 연결 정보를 URL string에 추가로 지정하여야 한다. HA를 위해 지정되는 속성은 장애 시 연결(failover)할 하나 이상의 standby 브로커의 호스트 정보(**alhosts**), 또는 active 서버의 장애 복구 시 active 브로커로 연결(failback)을 시도하는 주기(**rctime**)이다. JDBC에서 연결 설정을 위한 상세한 설명은 [연결 설정](#)을 참고한다.

구문

```
jdbc:cubrid:<host>:<port>:<db-name>:[user-id]:[password]:?[<property> [& <property>]]

host :
hostname | ip_address

property :
althosts= <alternative_hosts> | rctime= <second> | charset= <character_set>

alternative_hosts :
<standby_broker1_host>:<port> [, <standby_broker2_host>:<port>]
```

- **althosts** : 기본으로 지정한 브로커에 연결할 수 없는 경우, 그 다음으로 연결을 시도(failover)할 브로커 정보를 나타낸다. failover할 브로커를 여러 개 지정할 수 있고, **althosts**에 나열한 순서대로 연결을 시도한다.
- **rctime** : 장애가 발생했던 active 브로커에 연결을 시도하는 주기이다. 이 값은 active 브로커에 장애가 발생하여 standby 브로커로 failover한 경우에 사용된다. 장애 발생 후 **althosts**에 명시한 브로커로 접속하여(failover) 트랜잭션을 종료한 후, **rctime**만큼 시간이 경과할 때마다 기존의 active 브로커에 연결을 시도한다(failback). 이 값이 설정되어 있지 않으면, 600초로 설정된다.

예제

```
--connection URL string when user name and password omitted
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::

--connection URL string when charset property specified
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::charset=utf-8

--connection URL string when a property(althosts) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::althosts=127.0.0.2:31000,127.0.0.3:31000

--connection URL string when properties(althosts,rctime) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::althosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600

--connection URL string when properties(althosts,rctime, charset) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::althosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600
&charset=utf-8
```

HA 관련 시스템 카탈로그

클래스 이름: db_ha_apply_info

applylogdb 유틸리티가 복제 로그를 반영할 때마다 그 진행 상태를 저장하기 위한 테이블이다. 이 테이블은 **applylogdb** 유틸리티가 커밋하는 시점마다 갱신되며, *_counter 컬럼에는 수행 연산의 누적 카운트 값이 저장된다. 각 컬럼의 의미는 다음과 같다.

컬럼명	컬럼 타입	의미
db_name	VARCHAR(255)	로그에 저장된 DB 이름
db_creation_time	DATETIME	반영하는 로그에 대한 원본 DB의 생성 시각
copied_log_path	VARCHAR(4096)	반영하는 로그 파일의 경로
page_id	INTEGER	슬레이브 DB에 커밋된 복제 로그의 page

offset	INTEGER	슬레이브 DB 에 커밋된 복제 로그의 offset
log_record_time	DATETIME	슬레이브 DB 에 커밋된 복제 로그에 포함된 timestamp, 즉 해당 로그 레코드 생성 시간
last_access_time	DATETIME	applylogdb 가 슬레이브 DB 에 commit 한 시각
insert_counter	BIGINT	applylogdb 가 insert 한 횟수
update_counter	BIGINT	applylogdb 가 update 한 횟수
delete_counter	BIGINT	applylogdb 가 delete 한 횟수
schema_counter	BIGINT	applylogdb 가 schema 를 변경한 횟수
commit_counter	BIGINT	applylogdb 가 commit 한 횟수
fail_counter	BIGINT	applylogdb 가 insert/update/delete/commit/schema 변경 중 실패 횟수
required_page_id	INTEGER	applylogdb 가 읽을 수 있는 최소 pageid
start_time	DATETIME	applylogdb 프로세스가 슬레이브 DB 에 접속한 시간
status	INTEGER	반영 진행 상태 (0: IDLE, 1: BUSY)

성능 튜닝

성능에 영향을 미칠 수 있는 시스템 파라미터의 설정 정보를 제공한다. 시스템 파라미터는 시스템의 전체적인 성능과 동작을 결정한다. 이 장에서는 데이터베이스 서버, 브로커, CUBRID 매니저 서버에 적용하는 설정 파일의 사용법과 개별 파라미터의 의미를 설명한다.

이 장에서는 설명하는 주요 내용은 다음과 같다.

- 데이터베이스 서버 설정
- 브로커 설정
- CUBRID 매니저 서버 설정

데이터베이스 서버 설정

데이터베이스 서버 설정이 미치는 범위

CUBRID는 데이터베이스 서버, 브로커, CUBRID 매니저로 구성되며, 각 구성요소에 대한 설정 파일이 존재한다. 데이터베이스 서버의 시스템 파라미터 설정 파일은 **cubrid.conf**이며 **\$CUBRID/conf** 디렉터리에 위치한다. **cubrid.conf**에 설정되는 시스템 파라미터는 데이터베이스 시스템의 전체적인 성능과 동작에 영향을 준다. 따라서, 데이터베이스 서버 설정을 이해하는 작업은 매우 중요하다.

CUBRID 데이터베이스 서버는 클라이언트/서버 구조로 구성된다. 구체적으로 서버 라이브러리와 링크되어 있는 데이터베이스 서버 프로세스와 클라이언트 라이브러리와 링크되어 있는 브로커 프로세스로 나뉜다. 서버 프로세스는 데이터베이스 저장 구조를 관리하고 동시성과 트랜잭션 기능을 제공하며, 클라이언트 프로세스는 쿼리 실행을 위한 준비 단계와 객체 관리 및 스키마 관리 기능을 수행한다.

cubrid.conf 파일에서 설정할 수 있는 데이터베이스 서버의 시스템 파라미터는 적용되는 범위에 따라 클라이언트 파라미터, 서버 파라미터, 클라이언트/서버 파라미터로 구분할 수 있다. 클라이언트 파라미터는 브로커와 같은 클라이언트 프로세스에만 적용되는 파라미터이며, 서버 파라미터는 서버 프로세스의 동작에 영향을 주는 파라미터이다. 클라이언트/서버 파라미터는 서버와 클라이언트에 적용된다.

cubrid.conf 파일의 위치와 적용

- 데이터베이스 서버 프로세스는 **\$CUBRID/conf/cubrid.conf** 파일만 참조한다. 데이터베이스별 설정은 **cubrid.conf** 파일 내에서 섹션으로 구분한다.
- 클라이언트 프로세스는 1) **\$CUBRID/conf/cubrid.conf** 파일을 참조한 후에, 2) 현재 디렉터리(**\$PWD**)에 있는 **cubrid.conf** 파일을 추가로 참조한다. 현재 디렉터리 파일(**\$PWD/cubrid.conf**) 파일의 설정이 **\$CUBRID/conf/cubrid.conf** 파일의 설정을 덮어쓴다. 즉, **\$PWD/cubrid.conf**와 **\$CUBRID/conf/cubrid.conf**에서 동일한 파라미터에 대한 설정이 존재하면, **\$PWD/cubrid.conf**에 있는 설정이 최우선으로 적용된다.

cubrid.conf 설정 파일과 기본 제공 파라미터

CUBRID는 데이터베이스 서버, 브로커, CUBRID 매니저로 구성된다. 각 구성 요소를 수행하기 위한 설정 파일 이름은 다음과 같고, 모두 **\$CUBRID/conf** 디렉터리에 위치한다.

- 데이터베이스 서버 설정 파일 : **cubrid.conf**
- 브로커 설정 파일 : **cubrid_broker.conf**
- CUBRID 매니저 서버 설정 파일 : **cm.conf**

cubrid.conf 파일은 CUBRID 데이터베이스 서버에 대한 시스템 파라미터를 지정하는 설정 파일로 데이터베이스 시스템의 전체적인 성능과 동작을 결정한다. **cubrid.conf** 파일은 시스템 설치에 필요한 몇 가지 중요한 파라미터가 디폴트로 설정된 상태로 제공된다.

데이터베이스 서버 시스템 파라미터

다음은 **cubrid.conf** 설정 파일에 사용 가능한 데이터베이스 서버 시스템 파라미터이다. 적용 구분의 클라이언트 파라미터, 서버 파라미터의 의미는 [데이터베이스 서버 설정이 미치는 범위](#)를 참조한다.

파라미터 이름	적용 구분	타입	디폴트 값
cubrid_port_id	클라이언트 파라미터	int	1523
communication_histogram	클라이언트 파라미터	bool	no
db_hosts	클라이언트 파라미터	string	NULL
max_clients	서버 파라미터	int	50
ansi_quotes	클라이언트 파라미터	bool	yes
block_ddl_statement	클라이언트 파라미터	bool	no
block_nowhere_statement	클라이언트 파라미터	bool	no
compat_numeric_division_scale	클라이언트/서버 파라미터	bool	no
intl_mbs_support	클라이언트 파라미터	bool	no
oracle_style_empty_string	클라이언트 파라미터	bool	no
only_full_group_by	클라이언트 파라미터	bool	no
pipes_as_concat	클라이언트 파라미터	bool	yes
data_buffer_pages	서버 파라미터	int	25000
dont_reuse_heap_file	서버 파라미터	bool	no
index_scan_oid_buffer_pages	서버 파라미터	float	4.0
sort_buffer_pages	서버 파라미터	int	16
temp_file_memory_size_in_pages	서버 파라미터	int	4
thread_stack_size	서버 파라미터	int	102400
garbage_collection	클라이언트 파라미터	bool	no
temp_file_max_size_in_pages	서버 파라미터	int	-1
temp_volume_path	서버 파라미터	string	NULL
unfill_factor	서버 파라미터	float	0.1
volume_extension_path	서버 파라미터	string	NULL
call_stack_dump_activation_list	클라이언트/서버 파라미터	string	NULL
call_stack_dump_deactivation_list	클라이언트/서버 파라미터	string	NULL
call_stack_dump_on_error	클라이언트/서버 파라미터	bool	no

error_log	클라이언트/서버 파라미터	string	cub_client.err, cub_server.err
error_log_level	클라이언트/서버 파라미터	string	syntax
error_log_warning	클라이언트/서버 파라미터	string	no
error_log_size	클라이언트/서버 파라미터	int	8000000
auto_restart_server	서버 파라미터	bool	yes
deadlock_detection_interval_in_secs	서버 파라미터	int	1
file_lock	서버 파라미터	bool	yes
isolation_level	클라이언트 파라미터	int	3
lock_escalation	서버 파라미터	int	100000
lock_timeout_in_secs	클라이언트 파라미터	int	-1
lock_timeout_message_type	서버 파라미터	int	0
background_archiving	서버 파라미터	bool	yes
log_max_archives	서버 파라미터	int	INT_MAX
page_flush_interval_in_msecs	서버 파라미터	int	0
adaptive_flush_control	서버 파라미터	bool	true
max_flush_pages_per_second	서버 파라미터	int	10000
sync_on_nflush	서버 파라미터	int	200
pthread_scope_process	서버 파라미터	bool	yes
backup_volume_max_size_bytes	서버 파라미터	int	-1
checkpoint_interval_in_mins	서버 파라미터	int	720
checkpoint_every_npages	서버 파라미터	int	10000
log_buffer_pages	서버 파라미터	int	50
media_failure_support	서버 파라미터	bool	yes
insert_execution_mode	클라이언트 파라미터	int	1
max_plan_cache_entries	클라이언트/서버 파라미터	int	1000
max_query_cache_entries	서버 파라미터	int	-1
query_cache_mode	서버 파라미터	int	0
query_cache_size_in_pages	서버 파라미터	int	-1
replication	서버 파라미터	bool	no

index_scan_in_oid_order	클라이언트 파라미터	bool	no
single_byte_compare	서버 파라미터	bool	no
compactdb_page_reclaim_only	서버 파라미터	int	0
csql_history_num	클라이언트 파라미터	int	50
java_stored_procedure	서버 파라미터	bool	no
async_commit	서버 파라미터	bool	no
group_commit_interval_in_msecs	서버 파라미터	int	0
index_unfill_factor	서버 파라미터	float	0.20

파라미터의 섹션별 분류

cubrid.conf에 지정된 파라미터는 다음과 같이 세 가지 섹션으로 제공된다.

- CUBRID 서비스를 시작할 때 사용 : [service] 섹션
- 전체 데이터베이스에 공통으로 적용 : [common] 섹션
- 각 데이터베이스에 개별적으로 적용 : [@<database>] 섹션

여기서 <database>는 파라미터를 개별적으로 적용할 데이터베이스 이름이며, [common]에 설정된 파라미터가 [@<database>]에 설정된 파라미터와 동일한 경우 [@<database>]에 설정된 파라미터가 최종 적용된다.

기본 제공 파라미터

CUBRID 설치 시 생성되는 기본 데이터베이스 환경 설정 파일(**cubrid.conf**)에는 데이터베이스 서버 파라미터 중 반드시 변경해야 할 일부 파라미터가 기본적으로 포함된다. 디폴트로 포함되지 않는 파라미터의 설정값을 변경하기 원할 경우 직접 추가/편집해서 사용하면 된다.

다음은 **cubrid.conf** 파일 내용이다.

```
# Copyright (C) 2008 Search Solution Corporation. All rights reserved by Search Solution.
#
# $Id$
#
# cubrid.conf#

# For complete information on parameters, see the CUBRID
# Database Administration Guide chapter on System Parameters

# Service section - a section for 'cubrid service' command
[service]

# The list of processes to be started automatically by 'cubrid service start' command
# Any combinations are available with server, broker and manager.
service=server,broker,manager

# The list of database servers in all by 'cubrid service start' command.
# This property is effective only when the above 'service' property contains 'server'
keyword.
#server=foo,bar
```

```

# Common section - properties for all databases
# This section will be applied before other database specific sections.
[common]

# Number of data buffer pages
# data buffer pages (25,000 pages) * DB page size (4KB) = 100M
data_buffer_pages=25000

# Number of sort buffer pages
# sort buffer pages (16 pages) * DB page size (4KB) * number of threads
sort_buffer_pages=16

# Number of log buffer pages.
# log buffer pages (50 pages) * DB page size (4KB) = 200KB
log_buffer_pages=50

# Maximum number of locks acquired on individual instances of a
# class before the locks on the instances are escalated to a class lock
lock_escalation=100000

# Minimal amount of time to wait for a lock (seconds).
# A negative value, indicates to wait indefinitely until the lock is
# granted or until the transaction is rolled back as a result of a deadlock.
# A value of 0, indicates not to wait for a lock.
lock_timeout_in_secs=-1

# Interval between attempts at deadlock detection (seconds).
# An approximate interval to attempt to run the deadlock detector.
deadlock_detection_interval_in_secs=1

# Checkpoint when the specified time has passed (minutes).
# Checkpoint will be done also when log has grown by specified pages.
checkpoint_interval_in_mins=720

# Transaction isolation level.
# Six levels of isolation are provided, represented by:
# "TRAN SERIALIZABLE"
# "TRAN REP CLASS REP INSTANCE"
# "TRAN REP CLASS COMMIT INSTANCE"
# "TRAN REP CLASS UNCOMMIT INSTANCE"
# "TRAN COMMIT CLASS COMMIT INSTANCE"
# "TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE"
# For other aliases, or for more information on the levels, see the
# System Parameters chapter in the Database Administration Guide.
isolation_level="TRAN REP CLASS UNCOMMIT INSTANCE"

# TCP port id for the CUBRID programs (used by all clients).
cubrid_port_id=1523

# The maximum number of concurrent client connections the server will accept.
# This value also means the total # of concurrent transactions.
max_clients=50

# Restart the server process automatically
auto_restart_server=yes

# Become a master server for replication.
replication=no

# Enable Java Stored Procedure
java_stored_procedure=no

```

접속 관련 파라미터

다음은 데이터베이스 서버와 관련된 파라미터로 각 파라미터의 타입과 설정 가능한 값의 범위는 다음과 같다.

파라미터 이름	타입	디폴트 값	최소값	최대값
---------	----	-------	-----	-----

cubrid_port_id	int	1523	1	
db_hosts	string	NULL		
max_clients	int	50	10	1024

cubrid_port_id

cubrid_port_id는 마스터 프로세스가 사용하는 포트를 설정하기 위한 파라미터로 디폴트 값은 **1523**이다. CUBRID를 설치한 서버에서 이미 1523 포트를 사용하고 있거나, 방화벽에 의해 1523 포트가 차단된 경우에는 마스터 프로세스가 정상적으로 구동할 수 없으므로, 마스터 서버와 연결할 수 없다는 에러 메시지가 나타날 수 있다. 이와 같이 포트 충돌이 발생하는 경우, 관리자는 서버 환경을 고려하여 **cubrid_port_id**의 설정값을 변경해야 한다.

db_hosts

db_hosts는 클라이언트에서 연결할 수 있는 데이터베이스 서버 호스트의 목록 및 연결 순서를 지정하기 위한 파라미터이다. 서버 호스트 목록은 한 개 이상의 서버 호스트 이름을 나열하며, 각 호스트는 이름 사이에 공백 또는 콜론(:) 기호를 사용하여 구분한다. 이 때, 중복되거나 존재하지 않는 호스트 이름은 무시된다.

다음은 **db_hosts** 파라미터의 설정값을 보여주는 예제로 **host1**, **host2**, **host3**의 순서대로 연결이 시도된다.

```
db_hosts="host1:host2:host3"
```

한편, 클라이언트는 서버 연결을 위하여 데이터베이스 위치 정보 파일(**databases.txt**)을 참조하여 지정된 서버 호스트에 1차적으로 연결을 시도한다. 연결이 실패하면 데이터베이스 설정 파일(**cubrid.conf**)의 **db_hosts** 파라미터의 설정값을 참조하여 2차적으로 지정된 서버 호스트 중 첫 번째 서버 호스트에 연결을 시도한다.

max_clients

max_clients는 데이터베이스 서버에 동시 연결을 허용하는 클라이언트(일반적으로 브로커 프로세스)의 최대 수를 지정하기 위한 파라미터이다. 즉, **max_clients** 파라미터는 동시에 수행하고 있는 트랜잭션의 수를 의미한다. 이 파라미터의 디폴트 값은 **50**이다. 하지만, 서버를 사용하는 동시 사용자 수를 고려하여 적절한 값으로 개수를 제한하는 것이 필요하다.

메모리 관련 파라미터

다음은 데이터베이스 서버 또는 클라이언트에서 사용하는 메모리와 관련된 파라미터로 각 파라미터의 타입과 설정 가능한 값의 범위는 다음과 같다.

파라미터 이름	타입	디폴트 값	최소값	최대값
---------	----	-------	-----	-----

data_buffer_pages	int	25000	1	
index_scan_oid_buffer_pages	float	4.0	0.05	16.0
sort_buffer_pages	int	16	1	
temp_file_memory_size_in_pages	int	4	0	20
thread_stacksize	int	102400	65536	
garbage_collection	bool	no		

data_buffer_pages

data_buffer_pages는 데이터베이스 서버가 메모리 내에 캐시하는 데이터 페이지의 개수를 설정하기 위한 파라미터이다. **data_buffer_pages** 파라미터의 값이 클수록 버퍼에 캐시되는 데이터 페이지가 많아지므로 디스크 I/O 비용을 줄일 수 있다는 장점이 있다. 반면, 이 파라미터의 값을 너무 크게 설정하면 과도하게 시스템 메모리가 점유되므로 운영체제에 의해 버퍼 풀이 스와핑(swapping)되는 현상이 발생할 수 있다. **data_buffer_pages** 파라미터는 필요한 메모리 크기가 시스템 메모리의 2/3 이내가 되도록 설정할 것을 권장한다. 디폴트 값은 **25,000**페이지이다.

- 필요한 메모리 크기 = 데이터 버퍼 페이지수(**data_buffer_pages**) * 데이터베이스 페이지 크기(page size)
- 데이터 버퍼 페이지 수 = **data_buffer_pages** 파라미터의 설정값
- 데이터베이스 페이지 크기 = 데이터베이스 생성 시 **cubrid createdb** 유틸리티의 **-s** 옵션에 의해 지정된 페이지 크기값

index_scan_oid_buffer_pages

index_scan_oid_buffer_pages는 인덱스 스캔을 수행할 때 OID 리스트의 임시 저장을 위한 버퍼 페이지 개수를 설정하기 위한 파라미터로 디폴트 값은 **4.0**이다. 최소값은 0.05이며, 최대값은 16.0이다.

index_scan_oid_buffer_pages 파라미터 값과 데이터베이스 생성 시 설정한 단위 페이지의 크기에 비례하여 OID 버퍼의 크기가 결정되고, 이러한 OID버퍼의 크기가 클수록 인덱스 스캔 비용이 증가하는 경향을 보인다. 이를 고려하여 **index_scan_oid_buffer_pages** 파라미터 값을 조정할 수 있다.

sort_buffer_pages

sort_buffer_pages는 정렬을 수행하는 질의를 처리할 때 사용되는 버퍼 페이지의 개수를 설정하기 위한 파라미터로 디폴트 값은 **16**이며, 최소값은 1이다. 서버는 각 클라이언트 요청에 대하여 하나의 정렬 버퍼를 할당하며, 정렬을 완료한 후에는 할당되었던 버퍼 메모리를 해제한다.

temp_file_memory_size_in_pages

temp_file_memory_size_in_pages는 질의에 관한 임시 결과를 캐시하는 버퍼 페이지 개수를 설정하기 위한 파라미터로 디폴트 값은 4이며, 최대값은 20까지 허용된다.

- 필요한 메모리 크기 = 임시 메모리 버퍼 페이지 수(**temp_file_memory_size_in_pages**) * 데이터베이스 페이지 크기(page size)
- 임시 메모리 버퍼 페이지 수 = **temp_file_memory_size_in_pages** 파라미터 설정값
- 데이터베이스 페이지 크기 = 데이터베이스 생성 시 **cubrid createdb** 유틸리티의 **-s** 옵션에 의해 지정된 페이지 크기 값

thread_stacksize

thread_stacksize는 스레드의 스택 크기를 설정하기 위한 파라미터로 디폴트 값은 **100*1024**이다.

thread_stacksize 파라미터의 설정값은 운영체제가 허용하는 스택 크기를 초과할 수 없다.

garbage_collection

garbage_collection은 클라이언트에서 사용되지 않는 쓰레기(garbage) 메모리를 해제할 것인지를 설정하는 파라미터로 디폴트 값은 **no**이다.

디스크 관련 파라미터

다음은 데이터베이스 볼륨 정의 및 파일 저장을 위한 디스크 관련 파라미터로 각 파라미터의 타입과 설정 가능한 값의 범위는 다음과 같다.

파라미터 이름	타입	디폴트 값	최소값	최대값
temp_file_max_size_in_pages	int	-1		
temp_volume_path	string	NULL		
unfill_factor	float	0.1	0.0	0.3
volume_extension_path	string	NULL		
dont_reuse_heap_file	bool	no		

temp_file_max_size_in_pages

temp_file_max_size_in_pages는 복잡한 질의문이나 정렬 수행을 위하여 사용되는 임시 볼륨을 디스크에 저장하기 위하여 최대로 할당할 수 있는 페이지 개수를 설정하는 파라미터로 디폴트 값은 **-1**이다. 디폴트 값으로 설정되면 **temp_volume_path** 파라미터에서 지정된 디스크 공간 이내에서 무제한적으로 임시 볼륨이 생성 및 저장되고, 0으로 설정되면 임시 볼륨이 생성되지 않으므로 관리자가 직접 **cubrid addvoldb** 유틸리티를 이용하여 임시 볼륨을 생성해야 한다.

temp_volume_path

temp_volume_path는 복잡한 질의문이나 정렬 수행을 위하여 사용되는 임시 볼륨이 생성되는 디렉터리를 지정하는 파라미터로 디폴트 값은 데이터베이스 생성 시에 설정된 볼륨 위치이다.

unfill_factor

unfill_factor는 데이터 갱신에 대비하여 힙(heap) 페이지로 할당되는 디스크 공간의 비율을 정의하기 위한 파라미터로 디폴트 값은 **0.1**로 10%의 여유 공간이 설정된다. 원칙적으로, 테이블의 데이터는 물리적인 순서대로 삽입되지만, 데이터가 원래 크기보다 큰 데이터로 갱신되어 해당 페이지의 저장 공간이 부족하면 갱신된 데이터는 다른 페이지에 재배치되어야 하므로 성능이 저하될 수 있다. 이를 방지하기 위하여 **unfill_factor** 파라미터를 통해 힙 페이지 공간 비율을 설정할 수 있고, 최대값은 0.3(30%)까지 허용된다. 한편, 데이터 갱신이 거의 발생하지 않는 데이터베이스에서는 이 파라미터를 0.0으로 설정하여 데이터 갱신을 위한 힙 페이지 공간을 할당하지 않을 수 있고, **unfill_factor** 파라미터의 값이 음수거나 최대값보다 크게 설정되는 경우에는 디폴트 값(**0.1**)이 적용된다.

volume_extension_path

volume_extension_path는 자동으로 확장되는 볼륨이 생성되는 디렉터리를 지정하기 위한 파라미터로 디폴트 값은 데이터베이스 생성 시 설정된 볼륨 위치이다.

dont_reuse_heap_file

dont_reuse_heap_file은 테이블 삭제(DROP TABLE)로 인해 삭제된 힙 파일을 새로운 테이블 생성(CREATE TABLE)시 재사용하지 않도록 설정하는 파라미터로, 0으로 설정되면 삭제된 힙 파일을 재사용하고, 1로 설정되면 삭제된 힙 파일을 새로운 테이블 생성 시 재사용하지 않는다. 디폴트 값은 **0**이다.

오류 메시지 관련 파라미터

다음은 CUBRID에 의해 기록되는 오류 메시지의 처리에 관한 파라미터로 각 파라미터의 타입과 설정 가능한 값의 범위는 다음과 같다.

파라미터 이름	타입	디폴트 값
call_stack_dump_activation_list	string	NULL
call_stack_dump_deactivation_list	string	NULL
call_stack_dump_on_error	bool	no
error_log	string	cub_client.err, cub_server.err
error_log_level	string	syntax
error_log_warning	string	no

error_log_size	string	8000000
----------------	--------	---------

call_stack_dump_activation_list

call_stack_dump_activation_list는 모든 오류에 대해 콜-스택을 덤프하지 않기로 설정한 상태에서, 예외적으로 콜-스택을 덤프할 특정 오류 번호를 지정하기 위한 파라미터이다. 따라서, **call_stack_dump_activation_list** 파라미터는 **call_stack_dump_on_error**의 값이 **no**인 경우에만 효력이 있다. 다음은 -115, -116번의 오류 번호를 제외한 나머지 오류에 대해서 콜-스택 덤프가 수행되지 않도록 파라미터를 설정한 예제이다.

```
call_stack_dump_on_error= no
call_stack_dump_activation_list=-115,-116
```

call_stack_dump_deactivation_list

call_stack_dump_deactivation_list는 모든 오류에 대해 콜-스택 덤프를 설정한 상태에서, 예외적으로 콜-스택을 덤프하지 않는 특정 오류 번호를 지정하기 위한 파라미터이다. 따라서, **call_stack_dump_deactivation_list** 파라미터는 **call_stack_dump_on_error**의 값이 **yes**인 경우에만 효력이 있다. 다음은 -115, -116번의 오류 번호를 제외한 나머지 오류에 대해서 콜-스택 덤프를 수행하기 위해 파라미터를 설정한 예제이다.

```
call_stack_dump_on_error= yes
call_stack_dump_deactivation_list=-115,-116
```

call_stack_dump_on_error

call_stack_dump_on_error는 데이터베이스 서버에서 오류가 발생했을 때 콜-스택을 덤프할지 결정하기 위한 파라미터이다. no로 설정되면 모든 오류에 대해서 콜-스택을 덤프하지 않고, yes로 설정되면 모든 오류에 대해서 콜-스택을 덤프한다. 디폴트 값은 **no**이다.

error_log

error_log는 데이터베이스 서버에 오류가 발생하는 경우, 에러 로그가 저장되는 파일 이름을 지정하기 위한 서버/클라이언트 파라미터이다. 에러 로그가 저장되는 파일명의 작성 규칙은 `<database_name>_<date>_<time>.err`이다. 한편 시스템이 데이터베이스 서버 정보를 찾을 수 없는 오류에 대해서는 에러 로그 파일명의 작성 규칙을 따를 수 없다. 따라서, **cubrid.err** 파일에 오류 로그를 기록한다. **cubrid.err** 에러 로그 파일은 **\$CUBRID/log/server** 디렉터리에 저장된다.

error_log_level

error_log_level은 에러 심각성(severity) 수준에 따라 에러 로그 파일에 저장할 에러 메시지를 지정할 수 있는 서버 파라미터이다. 에러 심각성 수준은 가장 낮은 수준인 **NOTIFICATION**부터 가장 심각한 수준인 **FATAL**까지 총 5단계로 구성되며, 그에 따른 에러 메시지 포함 관계는 **FATAL < ERROR < SYNTAX**

C **WARNING** C **NOTIFICATION**이다. 디폴트 값은 **SYNTAX**이며, 이 경우 **FATAL, ERROR, SYNTAX**에 해당하는 에러 메시지만 에러 로그 파일에 기록된다.

error_log_warning

error_log_warning은 에러 심각성(severity) 수준이 **WARNING**인 에러 메시지의 출력 여부를 설정할 수 있는 서버 파라미터이다. 디폴트 값은 **no**이므로, **error_log_level**의 값이 **NOTIFICATION**으로 설정된 경우에도 **WARNING** 메시지를 제외한 나머지 수준의 에러 메시지만 저장될 것이다. 따라서, **WARNING** 메시지가 에러 로그 파일에 저장되도록 하려면, **error_log_warning**의 값을 **yes**로 설정해야 한다.

error_log_size

error_log_size는 에러 로그 파일에서 기록되는 최대 라인 수를 지정하는 파라미터로 디폴트 값은 **8,000,000**이다. 에러 로그 파일의 라인 수가 이 파라미터의 설정값에 도달하면 `<database_name>_<date>_<time>.err.bak` 파일이 생성된다.

동시성/잠금 파라미터

다음은 데이터베이스 서버의 동시성 제어 및 잠금에 관한 파라미터로 각 파라미터의 타입과 설정 가능한 값의 범위는 다음과 같다.

파라미터 이름	타입	디폴트 값	최소값	최대값
deadlock_detection_interval_in_secs	int	1	1	
isolation_level	int	3	1	6
lock_escalation	int	100000	5	
lock_timeout_in_secs	int	-1	-1	
lock_timeout_message_type	int	0	0	2

deadlock_detection_interval_in_secs

deadlock_detection_interval_in_secs는 중단된 트랜잭션에 대해 교착 상태 여부를 탐지하는 주기를 초 단위로 설정하기 위한 파라미터이다. CUBRID 시스템은 교착 상태에 있는 트랜잭션 중 하나를 롤백시켜 교착 상태를 해결한다. 디폴트 값은 1초이다. 탐지 주기가 길면 오랜 시간동안 교착 상태를 탐지할 수 없으므로 주의한다.

isolation_level

isolation_level은 트랜잭션의 격리 수준을 설정하기 위한 파라미터로 격리 수준이 높을수록 트랜잭션의 동시성이 적고 다른 동시성 트랜잭션에 의해 간섭받지 않는다. **isolation_level** 파라미터는 격리 수준을 의미하는 1에서 6까지의 정수값 또는 문자 스트링으로 설정하며, 디폴트 값은

TRAN_REP_CLASS_UNCOMMIT_INSTANCE이다. 각 격리 수준 및 파라미터 설정값에 대한 자세한 내용은 [격리 수준 설정](#)과 다음 표를 참조한다.

격리 수준	isolation_level 파라미터 설정값
SERIALIZABLE	"TRAN_SERIALIZABLE" or 6
REPEATABLE READ CLASS with REPEATABLE READ INSTANCES	"TRAN_REP_CLASS_REP_INSTANCE" or "TRAN_REP_READ" or 5
REPEATABLE READ CLASS with READ COMMITTED INSTANCES (or CURSOR STABILITY)	"TRAN_REP_CLASS_COMMIT_INSTANCE" or "TRAN_READ_COMMITTED" or "TRAN_CURSOR_STABILITY" or 4
REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES	"TRAN_REP_CLASS_UNCOMMIT_INSTANCE" or "TRAN_READ_UNCOMMITTED" or 3
READ COMMITTED CLASS with READ COMMITTED INSTANCES	"TRAN_COMMIT_CLASS_COMMIT_INSTANCE" or 2
READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES	"TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE" or 1

- **TRAN_SERIALIZABLE** : 가장 높은 수준의 일관성을 보장하는 격리 수준이며, [SERIALIZABLE](#)을 참고한다.
- **TRAN_REP_CLASS_REP_INSTANCE** : 유령 읽기(phantom read)가 발생할 수 있는 격리 수준이며, [REPEATABLE READ CLASS with REPEATABLE READ INSTANCES](#)를 참고한다.
- **TRAN_REP_CLASS_COMMIT_INSTANCE** : 반복 불가능한 읽기(unrepeatable read)가 발생할 수 있는 격리 수준이며, [REPEATABLE READ CLASS with READ COMMITTED INSTANCES](#)를 참고한다.
- **TRAN_REP_CLASS_UNCOMMIT_INSTANCE** : 더티 읽기(dirty read)가 발생할 수 있는 격리 수준이며, [REPEATABLE READ CLASS with READ UNCOMMITTED INSTANCES](#)를 참고한다.
- **TRAN_COMMIT_CLASS_COMMIT_INSTANCE** : 반복 불가능한 읽기(unrepeatable read)가 발생할 수 있고, 데이터 조회 중에 다른 트랜잭션에 의한 테이블 스키마의 변경이 허용되는 격리 수준이며, [READ COMMITTED CLASS with READ COMMITTED INSTANCES](#)를 참고한다.
- **TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE** : 더티 읽기(dirty read)가 발생할 수 있고, 데이터 조회 중에 다른 트랜잭션에 의한 테이블 스키마의 변경이 허용되는 격리 수준이며, [READ COMMITTED CLASS with READ UNCOMMITTED INSTANCES](#)를 참고한다.

lock_escalation

lock_escalation은 행에 대한 잠금이 테이블 잠금으로 확대되기 전에 개별 행에 허용되는 최대 잠금의 개수를 설정하기 위한 파라미터로 디폴트 값은 **100,000**이다. **lock_escalation** 파라미터의 설정값이 작으면, 메모리 잠금 관리에 의한 오버헤드가 적은 반면 동시성은 줄어든다. 반대로 설정값이 크면 메모리 잠금 관리에 의한 오버헤드가 큰 반면 동시성이 향상된다.

lock_timeout_in_secs

lock_timeout_in_secs는 잠금 대기 시간을 지정하기 위한 클라이언트 파라미터로 지정된 시간 이내에 잠금이 허용되지 않으면 해당 트랜잭션이 취소되고 오류가 반환된다. 디폴트 값인 **-1**로 설정하면 잠금이 허용될 때까지의 대기 시간이 무제한이고, 0으로 설정하면 잠금을 대기하지 않는다.

lock_timeout_message_type

lock_timeout_message_type은 잠금 타임아웃 발생 시 반환되는 메시지에 포함되는 정보 수준을 설정하는 파라미터이다. 디폴트 값인 **0**으로 설정하면 잠금 소유 정보가 메시지에 포함되지 않고, 1로 설정하면 잠금 소유 정보가 하나만 포함되며, 2로 설정하면 잠금 소유 정보가 모두 포함된다.

- **lock_timeout_message_type** = 0인 경우

```
ERROR: Your transaction (index 3, cub_user@cdfs006.cub|15668) timed out waiting
on X LOCK lock on instance 0|636|34 of class participant. You are waiting for
user(s) to finish.
```

- **lock_timeout_message_type** = 1인 경우

```
ERROR: Your transaction (index 3, cub_user@cdfs006.cub|15668) timed out waiting
on X LOCK lock on instance 0|636|34 of class participant. You are waiting for user(s)
cub_user@cdfs006.cub|15615 to finish.
```

- **lock_timeout_message_type** = 2인 경우

```
ERROR: Your transaction (index 3, cub_user@cdfs006.cub|15668) timed out waiting
on X LOCK lock on instance 0|636|34 of class participant. You are waiting for user(s)
cub_user@cdfs006.cub|15615, cub_user@cdfs006.cub|15596 to finish.
```

로깅 관련 파라미터

다음은 CUBRID 데이터베이스의 백업과 복구에 이용되는 로그에 관련된 파라미터로 각 파라미터의 타입과 설정 가능한 값의 범위는 다음과 같다.

파라미터 이름	타입	디폴트 값	최소값	최대값
log_buffer_pages	int	50	3	
media_failure_support	bool	yes		
log_max_archives	int	INT_MAX	0	
background_archiving	bool	yes		
page_flush_interval_in_msecs	int	0	-1	
checkpoint_interval_in_mins	int	720	1	
checkpoint_every_npages	int	10000	10	
adaptive_flush_control	bool	true		
max_flush_pages_per_second	int	10000	1	INT_MAX
sync_on_nflush	int	200	1	INT_MAX

log_buffer_pages

log_buffer_pages는 메모리에 캐시되는 로그 버퍼의 페이지 수를 설정하는 파라미터로 디폴트 값은 **50**이다.

log_buffer_pages 파라미터의 설정값이 크면 데이터베이스 수정 연산이 많고, 길고 큰 트랜잭션이 많은 환경에서는 디스크 I/O가 감소되어 성능이 향상될 수 있다. CUBRID가 설치된 시스템의 메모리 크기 및 작업 연산의 크기를 고려하여 적당한 값으로 설정할 것을 권장한다.

- 필요한 메모리 크기 = 로그 버퍼 페이지수(**log_buffer_pages**) * 데이터베이스 페이지 크기(**database_page_size**)
- 로그 버퍼 페이지수 = **log_buffer_pages** 파라미터 설정값
- 데이터베이스 페이지 크기 = 데이터베이스 생성 시 **cubrid createdb** 유틸리티의 **-s** 옵션에 의해 지정된 페이지 크기 값

media_failure_support

media_failure_support는 저장 매체의 장애에 대비하여 보관 로그를 보존할지 여부를 설정하기 위한 파라미터이다. 디폴트 값인 **yes**로 설정하면 활성 로그가 꼭 찬 후 트랜잭션에 변경 사항이 있는 경우 활성 로그 전체를 보관 로그로 복사하여 보존한다. 파라미터 값을 **no**로 설정하면, 활성 로그가 꼭 찬 후 생성된 보관 로그가 자동으로 삭제된다.

이 파라미터를 **no**로 설정하면 **background_archiving** 파라미터도 비활성화된다.

log_max_archives

log_max_archives는 **media_failure_support**가 **yes**일 때, 보존할 보관 로그 파일의 최대 개수를 설정하기 위한 파라미터이다. 최소값은 0이며, 디폴트 값은 **INT_MAX**이다.

예를 들어, cubrid.conf에 **log_max_archives**의 값을 3으로 설정하면, 최근 3개의 보관 로그 파일만 유지하고 네 번째 보관 로그가 생성될 때 이전 생성된 보관 로그 파일을 자동으로 삭제한다. 이때 삭제되는 보관 로그의 정보는 *_lginf 파일에 기록된다.

다만, 활성화된 트랜잭션이 기존 보관 로그를 여전히 참조하고 있다면, 해당 보관 로그는 삭제되지 않는다. 즉, 어떤 트랜잭션이 첫 번째 보관 로그가 생성되는 시점에서 시작되어 다섯 번째 보관 로그가 생성되는 시점까지도 종료되지 않았다면 첫 번째 보관 로그는 삭제되지 않는다.

background_archiving

background_archiving은 **media_failure_support**의 값이 **yes**일 때, 특정 시점마다 주기적으로 임시 보관 로그를 생성하도록 하는 파라미터로서, 보관 로그 작업으로 인한 디스크 I/O 부하를 분산시키고자 할 때 유용하다. 디폴트 값은 **yes**이다.

page_flush_interval_in_msecs

page_flush_interval_in_msecs는 데이터 버퍼에 존재하는 더티 페이지를 디스크로 저장(flush)하는 주기를 밀리초(msec) 단위로 설정하는 파라미터이며, 디폴트 값은 **0**이다. 이 파라미터를 최소값인 -1로 설정하는 경우, 더티 페이지는 체크포인트 또는 페이지가 스왑(swap)되는 시점에만 디스크로 저장된다.

이는 I/O 부하, 버퍼 동시성과 관련있는 파라미터로서, 서비스 환경의 워크로드를 고려하여 파라미터 값을 설정해야 한다.

checkpoint_interval_in_mins

checkpoint_interval_in_mins는 체크포인트가 수행되는 주기를 분 단위로 설정하는 파라미터이며, 디폴트값은 **720**이다.

체크포인트는 데이터 버퍼에 존재하는 로그 파일(더티 페이지)을 디스크로 저장(flush)하는 작업이며, 데이터베이스 장애 발생 시 최근 체크포인트 시점까지 데이터를 복구할 수 있다. 다만, 체크포인트 작업으로 인해 디스크로 저장되는 로그 파일의 양이 많을 경우 디스크 I/O가 발생하여 DB 운영에 영향을 끼칠 수 있으므로 체크포인트 주기를 적절하게 설정해야 한다.

체크포인트 주기 설정과 관련된 파라미터는 **checkpoint_interval_in_mins**과

checkpoint_every_npages이며, **checkpoint_interval_in_mins** 파라미터의 설정값이 경과된 시점 또는 로그 페이지 수가 **checkpoint_every_npages** 파라미터의 설정값에 도달하는 시점마다 체크포인트 작업이 주기적으로 수행된다.

checkpoint_every_npages

checkpoint_every_npages는 체크포인트가 수행되는 주기를 로그 페이지 단위로 설정하는 파라미터이며, 디폴트값은 **10,000**이다. 특히, 특정 시간대에 **INSERT/UPDATE**가 집중되는 서비스 환경에서는 **checkpoint_every_npages** 파라미터의 설정값을 작게 설정하여 체크포인트 시점에 I/O 부하를 분산할 수 있다.

adaptive_flush_control

adaptive_flush_control는 현 시점의 내려쓰기(flush) 작업 진행 상태에 따라 50ms마다 내려쓰기할 용량(flush capacity)을 자동 조정하는 파라미터이며, 디폴트 값은 **yes**이다. 즉, 특정 시점에 **INSERT** 또는 **UPDATE** 연산이 집중되어 내려쓰기한 페이지 수가 **max_flush_pages_per_second** 파라미터 값에 도달하면 이 용량을 증가시키고, 이에 도달하지 못하면 이 용량을 감소시킨다. 이처럼 워크로드에 따라 주기적으로 내려쓰기 용량을 조정하여 I/O 부하를 분산할 수 있다.

max_flush_pages_per_second

max_flush_pages_per_second는 버퍼로부터 디스크로 내려쓰기(flush) 작업을 수행할 때, 내려쓰기할 최대 용량 (flush capacity)을 설정하기 위한 파라미터이며, 디폴트 값은 **10000**이다. 즉, 이 파라미터 설정을 통해 1초당 내려쓰기할 최대 용량을 제어하여, 특정 시점에 I/O 부하가 집중되는 현상을 방지할 수 있다.

만약, 특정 시점에 **INSERT** 또는 **UPDATE** 연산이 집중되어 이 파라미터에 의해 설정된 최대 용량에 도달하면, 로그 페이지만 내려쓰기를 수행하고 데이터 페이지는 더 이상 디스크로 내려쓰지 않는다. 따라서, 이 파라미터는 서비스 환경의 워크로드를 고려하여 적절한 값을 설정해야 한다.

sync_on_nflush

sync_on_nflush는 버퍼로부터 데이터 페이지 및 로그 페이지를 내려쓰기한 후, 운영 시스템의 FILE I/O와 동기화를 수행하는 주기를 페이지 단위로 설정하는 파라미터이며, 디폴트 값은 **200**이다. 즉, 200페이지만큼 내려쓰기 작업이 수행될 때마다 CUBRID 서버는 운영 체제의 FILE I/O와 동기화를 수행한다. I/O 부하와 관련된 파라미터이다.

트랜잭션 처리 관련 파라미터

다음은 트랜잭션의 커밋 성능 향상을 위한 파라미터로 각 파라미터의 타입과 설정 가능한 값의 범위는 다음과 같다.

파라미터 이름	타입	디폴트 값	최소값	최대값
async_commit	bool	no		
group_commit_interval_in_msecs	int	0	0	

async_commit

async_commit은 비동기식 커밋 기능을 활성화시키는 파라미터로 디폴트 값인 **no**로 설정하면 비동기식 커밋을 수행하지 않고, **yes**로 설정하면 비동기식 커밋을 수행한다. 비동기식 커밋이란 커밋 로그가 디스크에 플러시되기 이전에 클라이언트에게 커밋을 완료 처리하고, 로그 플러시 스레드(LFT)가 로그 플러시를 백그라운드에서 수행하여 커밋 작업의 성능을 향상시키는 기능이다. 로그 플러시가 수행되기 전에 데이터베이스 서버에 장애가 발생하면 이미 커밋 완료된 트랜잭션을 복구할 수 없으므로 주의한다.

group_commit_interval_in_msecs

group_commit_interval_in_msecs은 그룹 커밋을 수행하는 간격을 밀리세컨드(msec) 단위로 지정하는 파라미터로 디폴트 값인 **0**으로 설정되면 그룹 커밋을 수행하지 않는다. 그룹 커밋이란 지정된 시간동안 발생한 여러 번의 커밋을 그룹으로 취합하여 커밋 로그가 동시에 디스크에 플러시되도록 하여 커밋 작업의 성능을 향상시키는 기능이다.

구문/타입 관련 파라미터

다음은 CUBRID에서 지원하는 SQL 구문 및 데이터 타입에 관한 파라미터로 각 파라미터의 타입과 설정 가능한 값의 범위는 다음과 같다.

파라미터 이름	타입	디폴트 값
ansi_quotes	bool	yes
block_ddl_statement	bool	no
block_nowhere_statement	bool	no
compat_numeric_division_scale	bool	no
intl_mbs_support	bool	no
oracle_style_empty_string	bool	no
only_full_group_by	bool	no
pipes_as_concat	bool	yes

ansi_quotes

ansi_quotes는 식별자 처리를 위한 기호 또는 문자열을 감싸는 기호에 관한 파라미터로 디폴트 값은 **yes**이다. 이 파라미터 값이 **yes**이면 큰따옴표가 식별자 처리 기호로 해석되고 작은따옴표가 문자열 처리 기호로 해석된다. 한편, **no**이면 큰 따옴표가 문자 스트링 처리 기호로 해석된다.

block_ddl_statement

block_ddl_statement는 클라이언트가 수행하는 데이터 정의문(Data Definition Language, DDL)을 제한하기 위한 파라미터로 **no**로 설정하면 해당 클라이언트의 데이터 정의문 수행을 허용하며, **yes**로 설정하면 해당 클라이언트의 데이터 정의문 수행을 허용하지 않는다. 디폴트 값은 **no**이다.

block_nowhere_statement

block_nowhere_statement는 클라이언트가 수행하는 조건절(**WHERE**)이 없는 **UPDATE/DELETE** 문을 제한하기 위한 파라미터로 **no**로 설정하면 해당 클라이언트의 조건절이 없는 **UPDATE/DELETE** 문을 허용하며, **yes**로 설정하면 해당 클라이언트의 조건절이 없는 **UPDATE/DELETE** 문의 수행을 허용하지 않는다. 디폴트 값은 **no**이다.

compat_numeric_division_scale

compat_numeric_division_scale은 나눗셈 연산의 결과 값(몫)에 대하여 소수점 이하 자릿수를 몇 자리까지 표시할 것인가를 지정하기 위한 파라미터로 **no**로 설정하면 몫의 소수점 이하 자릿수가 9개가 되고, **yes**로 설정하면 몫의 소수점 이하 자릿수가 피연산자의 소수점 이하 자릿수에 따라 결정된다. 디폴트 값은 **no**이다.

intl_mbs_support

intl_mbs_support는 멀티바이트 문자 세트(Multibyte Character Set)의 지원 여부를 지정하기 위한 파라미터이며, 디폴트 값은 **no**이다. 한글과 같은 멀티바이트 문자로 테이블 이름 또는 컬럼 이름을 생성하는 경우, 이 파라미터 값을 **yes**로 설정해야 한다. 단, 멀티바이트 문자 세트를 지원하기 위한 연산 비용이 크므로, 성능 향상을 위해 **intl_mbs_support** 파라미터를 **no**로 설정하고 테이블 이름이나 컬럼 이름을 영어로 사용할 것을 권장한다.

oracle_style_empty_string

oracle_style_empty_string은 다른 DBMS(Database Management System)와의 호환성을 향상시키기 위한 파라미터로 빈 문자열(empty string)을 Oracle DBMS와 마찬가지로 **NULL**로 처리할 것인지 지정한다. **oracle_style_empty_string** 파라미터를 **no**로 설정하면 빈 문자열을 유효한 문자열로 처리하고, **yes**로 설정하면 Oracle DBMS와 마찬가지로 빈 문자열을 **NULL**로 처리한다.

only_full_group_by

only_full_group_by는 **GROUP BY** 절 사용에 관한 확장된 문법의 사용 여부를 설정하는 파라미터이다.

이 파라미터 값이 **no**이면 확장된 문법이 적용되므로 **GROUP BY** 절에 명시되지 않은 컬럼을 **SELECT** 컬럼 리스트에 명시할 수 있고, 이 값이 **yes**이면 **GROUP BY** 절에 명시된 컬럼만 **SELECT** 컬럼 리스트에 명시할 수 있다.

디폴트 값은 **no**이므로, SQL 표준에 따라 질의를 수행하려면 **only_full_group_by** 파라미터 값을 **yes**로 설정한다. 이 경우에는 확장된 문법이 적용되지 않으므로 실행 결과로 아래와 같은 에러가 출력된다.

```
ERROR: Attributes exposed in aggregate queries must also appear in the group by clause.
```

pipes_as_concat

pipes_as_concat은 이중 파이프 기호(II)의 사용에 관한 파라미터로서, 디폴트 값은 **yes**이다. 이 파라미터 값이 **yes**이면 이중 파이프 기호가 문자열의 병합 연산자로 해석되고, **no**이면 부울린(boolean) 연산자인 **OR**로 해석된다.

쿼리 캐시 관련 파라미터

다음은 동일한 **SELECT** 문에 대해 캐시된 수행 결과를 제공하는 쿼리 캐시 기능과 관련된 파라미터로 각 파라미터의 타입과 설정 가능한 값의 범위는 다음과 같다.

파라미터 이름	타입	디폴트 값	최소값	최대값
max_plan_cache_entries	int	1,000		
max_query_cache_entries	int	-1		

query_cache_mode	int	0	0	2
query_cache_size_in_pages	int	-1		

max_plan_cache_entries

max_plan_cache_entries는 메모리에 캐시하는 쿼리 플랜의 최대 개수를 설정하는 파라미터이다.

max_plan_cache_entries 파라미터가 -1이나 0으로 설정되면 작성된 쿼리 플랜을 메모리 캐시에 저장하지 않는 것이며, 1 이상의 정수값이 설정되면 설정된 개수만큼의 쿼리 플랜을 메모리 캐시한다. 또한, 이 파라미터의 값이 1 이상의 정수값으로 설정되어야 동일한 쿼리에 대한 결과를 캐시하는 쿼리 캐시 기능을 적용할 수 있다.

max_query_cache_entries

max_query_cache_entries는 쿼리 캐시 기능을 적용할 쿼리의 최대 개수를 지정하는 파라미터로 -1이나 0으로 설정하면 쿼리 캐시 기능을 비활성화시키는 것이고, 1 이상의 정수값으로 설정하면 설정된 개수만큼의 쿼리에 대해 수행 결과를 캐시한다. 쿼리 캐시 기능은 조회 데이터가 변하지 않고, 동일한 쿼리가 반복적으로 입력되는 경우에 성능 향상을 기대할 수 있다. 쿼리 캐시 기능은 쿼리 플랜 캐시 기능에 종속적이므로 쿼리 플랜 캐시 기능을 활성화시키는 **max_plan_cache_entries** 파라미터 값이 1 이상의 정수로 설정되어야 쿼리 캐시 관련 파라미터의 설정이 적용된다는 점에 주의한다.

query_cache_mode

query_cache_mode는 두 가지의 쿼리 캐시 모드 중 하나를 지정하기 위한 파라미터로 제 1 쿼리 캐시 모드는 모든 쿼리에 대해 쿼리 캐시 기능을 적용하는 것이고, 제 2 쿼리 캐시 모드는 /*+ QUERY_CACHE(1) */이라는 힌트가 주어진 쿼리에 대해서만 쿼리 캐시 기능을 적용하는 것이다. 이 파라미터를 디폴트 값인 0으로 설정하면 쿼리 캐시 기능이 비활성화되고, 1로 설정하면 제 1 쿼리 캐시 모드로 수행되며, 2로 설정하면 제 2 쿼리 캐시 모드로 수행된다. 쿼리 캐시 기능을 활성화하려면 1) **max_plan_cache_entries**를 1 이상의 값으로 설정해야 하고, 2) **max_query_cache_entries**를 1 이상의 값으로 설정해야 하며, 3) **query_cache_mode**를 1 이상의 값으로 설정해야 한다. 이 중 어느 하나의 파라미터라도 조건을 만족하지 않으면 쿼리 캐시 기능은 비활성화되므로 주의한다.

```
// 최대 1,000개의 쿼리에 대해 플랜 캐시 기능을 수행하고, 최대 100개의 쿼리에 대해 쿼리 캐시 기능을
수행하는 예제이다.
max_plan_cache_entries=1000
max_query_cache_entries=100
query_cache_mode=1
// 플랜 캐시 기능이 비활성화되어 있으므로, 쿼리 캐시와 관련된 두 파라미터의 설정값은 의미없다.
max_plan_cache_entries=-1
max_query_cache_entries=100
query_cache_mode=1
// 최대 1,000개의 쿼리에 대해 플랜 캐시 기능을 수행하고, 쿼리 캐시 기능은 비활성화된다.
max_plan_cache_entries=1000
max_query_cache_entries=100
query_cache_mode=0
```

query_cache_size_in_pages

query_cache_size_in_pages는 캐시할 수행 결과 페이지를 지정하기 위한 파라미터로 지정된 페이지 이내의 수행 결과를 가지는 경우에만 쿼리 캐시 기능을 적용한다. 디폴트 값인 **-1**로 설정하면 수행 결과의 페이지 크기에 대한 제약 없이 모든 쿼리에 대해 쿼리 캐시를 수행한다.

유틸리티 관련 파라미터

다음은 CUBRID에서 사용되는 유틸리티와 관련된 파라미터로 각 파라미터의 타입과 설정 가능한 값의 범위는 다음과 같다.

파라미터 이름	타입	디폴트 값	최소값	최대값
compactdb_page_reclaim_only	int	0		
csql_history_num	int	50	1	200
communication_histogram	string	no		
backup_volume_max_size_bytes	int	-1	1024*32	

compactdb_page_reclaim_only

compactdb_page_reclaim_only는 **compactdb** 유틸리티와 관련된 파라미터로 이미 할당된 저장 영역의 OID를 재사용하기 위하여 이미 삭제된 객체의 저장 영역을 정리하는 유틸리티이다. **compactdb** 유틸리티에 의해 저장 영역이 재정렬되는 작업은 3단계로 구분할 수 있으며, **compactdb_page_reclaim_only** 파라미터를 통해 재정렬 작업의 단위를 선택할 수 있다. 디폴트 값인 **0**으로 설정하면 1, 2, 3단계를 모두 수행하므로 데이터 단위, 테이블 단위, 파일 단위로 저장 영역을 재정렬한다. 1로 설정하면 1단계를 생략하므로 테이블 및 파일 단위로 저장 영역을 재정렬하고, 2로 설정하면 1, 2단계를 생략하므로 파일 단위로만 저장 영역을 재정렬한다.

- 1단계 : 데이터 단위로 저장 영역을 재정렬한다.
- 2단계 : 테이블 단위로 저장 영역을 재정렬한다.
- 3단계 : 파일 단위(heap file)로 저장 영역을 재정렬한다.

csql_history_num

csql_history_num은 CSQL 인터프리터와 관련된 파라미터로 CSQL 인터프리터 내에서 히스토리 내역으로 저장되는 SQL 문의 개수를 설정하는 파라미터이다. 디폴트 값은 **50**이다.

communication_histogram

communication_histogram은 데이터베이스 서버의 통계 정보를 확인하는 유틸리티인 **cubrid statdump** (자세한 내용은 [데이터베이스 서버 실행 통계 정보 출력](#) 참조)와 csql 인터프리터의 [세션 명령어](#) **"/;h"**와 관련된 파라미터이며, 디폴트 값은 **no**이다.

backup_volume_max_size_bytes

backup_volume_max_size_bytes는 **cubrid backupdb** 유틸리티에 의해 생성되는 백업 볼륨의 최대 크기를 바이트 단위로 설정하는 파라미터이다. 디폴트 값인 **-1**로 설정하면 생성되는 백업 볼륨의 크기가 무제한으로 허용되며, 파라미터가 설정되지 않으면 저장 매체의 한도 용량까지 백업 볼륨의 크기가 허용된다.

기타 파라미터

다음은 기타 파라미터 정보로 각 파라미터의 타입과 설정 가능한 값의 범위는 다음과 같다.

파라미터 이름	타입	디폴트 값	최소값	최대값
service	string			
server	string			
replication	bool	no		
index_scan_in_oid_order	bool	no		
single_byte_compare	bool	no		
insert_execution_mode	int	1	1	7
java_stored_procedure	bool	no		
pthread_scope_process	bool	yes		
auto_restart_server	bool	yes		
index_unfill_factor	float	0.20	0	0.35

service

service는 CUBRID 서비스 시작 시 자동으로 시작하는 프로세스를 등록하는 파라미터로 **server**, **broker**, **manager**의 세 종류 프로세스가 있다. 일반적으로 **service=server,broker,manager**와 같이 세 종류 프로세스를 모두 등록한다.

- 파라미터 값으로 **server**가 설정되면 **@server** 파라미터에서 지정한 데이터베이스 프로세스를 시작한다.
- 파라미터 값으로 **broker**가 설정되면 브로커 프로세스를 시작한다.
- 파라미터 값으로 **manager**가 설정되면 매니저 프로세스를 시작한다.

server

server는 CUBRID 서비스 시작 시 자동으로 시작하는 데이터베이스 서버 프로세스를 등록하는 파라미터이다.

replication

replication은 데이터베이스 복제 기능을 활성화시키는 파라미터이다. 디폴트 값인 **no**로 설정하면 복제 기능이 비활성화되고, **yes**로 설정하면 복제 기능이 활성화된다. 복제 기능이 활성화되면 해당 데이터베이스는 복제 로그를 생성하는 복제 마스터 서버의 역할을 수행한다.

index_scan_in_oid_order

index_scan_in_oid_order는 인덱스를 스캔한 후 검색 결과 데이터를 가져오는 순서를 OID 순으로 지정하기 위한 파라미터이다. 디폴트 값인 **no**로 설정하면 데이터 순서대로 결과를 가져오고, **yes**로 설정하면 OID 순서대로 결과를 가져온다.

single_byte_compare

single_byte_compare은 스트링 비교(string compare)를 1바이트 단위로 수행하기 위한 파라미터이다. 디폴트 값인 **no**로 설정하면 2바이트 단위로 스트링 비교를 수행한다. **yes**로 설정하면 1바이트 단위로 스트링 비교를 수행하므로 유니코드(UTF-8) 환경에서 저장된 데이터에 대해 정상적으로 스트링 비교/검색을 수행할 수 있다.

insert_execution_mode

insert_execution_mode는 **INSERT**가 서버 측에서 바로 수행되도록 하는 파라미터로서, 최소값은 1, 최대값은 31이다. 일반적으로 쿼리는 클라이언트에서 수행되고, 커밋 후에 서버에 반영된다. 그러나 **insert_execution_mode** 파라미터를 설정하면, 설정된 실행 모드에 해당하는 **INSERT** 문은 서버 측에서 바로 수행되고, 나머지 실행 모드에 해당하는 **INSERT** 문은 클라이언트에서 수행된다. 이는 **INSERT**한 데이터의 더티 읽기(dirty read)가 요구되는 환경 또는 클라이언트 메모리 용량이 제한적인 환경에서 서버 측 **INSERT** 문을 수행하기 위해 설정할 수 있다.

지원되는 **INSERT** 문의 실행 모드는 5가지이며, 각 실행 모드에 대응하는 정수 값의 조합을 통해 파라미터를 설정할 수 있다.

- **INSERT_SELECT** : **INSERT** 문에서 **SELECT** 문을 이용하는 경우이며, 정수값 1에 대응된다.

```
INSERT INTO code2(s_name, f_name) SELECT s_name, f_name from code;
```

- **INSERT_VALUES** : 보통 사용하는 **INSERT** 문에 해당하며, 정수값 2에 대응된다.

```
INSERT INTO code2(s_name, f_name) VALUES ('S', 'Silver');
```

- **INSERT_DEFAULT** : 디폴트 값이 정의된 컬럼을 포함하는 테이블에 대한 **INSERT** 문에서 해당 컬럼의 값을 별도로 명시하지 않는 경우이며, 정수값 4에 대응된다.

```
CREATE TABLE code2(s_name char(1) DEFAULT ' ', f_name varchar(40));  
INSERT INTO code2(f_name) DEFAULT VALUES;
```

- **INSERT_REPLACE** : **REPLACE** 문을 수행하는 경우이며, 정수값 8에 해당한다.

```
CREATE TABLE code2(s_name char(1) NOT NULL UNIQUE, f_name varchar(40));  
REPLACE INTO code2 VALUES ('S', 'Silver');
```

- **INSERT_ON_DUP_KEY_UPDATE** : **INSERT** 문에서 **ON DUPLICATE KEY UPDATE** 절을 명시하는 경우이며, 정수값 16에 해당한다.

```
CREATE TABLE code2(s_name char(1) NOT NULL UNIQUE, f_name varchar(40));
INSERT INTO code2 VALUES ('S', 'Silver') ON DUPLICATE KEY UPDATE f_name='Silver';
```

위의 각 실행 모드에 대응되는 정수 값의 합계를 파라미터 값으로 설정하여, 서버 측에서 바로 수행될 **INSERT** 문의 종류를 지정할 수 있다.

- 예 1) **INSERT_SELECT**와 **INSERT_VALUES**를 서버 측에서 실행하고자 하는 경우, **insert_execution_mode=3**으로 설정한다. ($1 + 2 = 3$)
- 예 2) **INSERT_SELECT**, **INSERT_DEFAULT**, **INSERT_REPLACE**, **INSERT_ON_DUP_KEY_UPDATE**를 서버 측에서 실행하고자 하는 경우, **insert_execution_mode=29**로 설정한다. ($1 + 4 + 8 + 16 = 29$)

java_stored_procedure

java_stored_procedure는 Java 가상 머신(Java Virtual Machine, JVM)을 실행하여 Java 저장 프로시저(Java stored procedure)를 사용하게 하기 위한 파라미터이다. 디폴트 값인 **no**로 설정하며 JVM이 실행되지 않고, **yes**로 설정하면 JVM이 실행되어 Java 저장 프로시저(Java stored procedure)를 사용할 수 있다. 따라서, Java 저장 프로시저를 사용할 계획이 있는 경우에는 파라미터를 **yes**로 설정해야 한다.

pthread_scope_process

pthread_scope_process는 스레드의 경쟁 범위를 설정하는 파라미터로 AIX 시스템에서만 적용된다. **no**로 설정하면 경쟁 범위가 **PTHREAD_SCOPE_SYSTEM**이 되고, **yes**로 설정하면 **PTHREAD_SCOPE_PROCESS**가 된다. 디폴트 값은 **yes**이다.

auto_restart_server

auto_restart_server는 데이터베이스 서버 프로세스에 심각한 오류가 발생해서 프로세스가 중단될 경우에 자동으로 재시작할 것인가를 지정하는 파라미터이다. **auto_restart_server**를 **yes**로 설정하면 서버 프로세스는 정상적인 종료 절차(CUBRID 서버의 **STOP** 명령)가 아닌 잘못 중단된 경우에 자동으로 재시작한다.

index_unfill_factor

최초 인덱스 생성 후 **INSERT**나 **UPDATE**를 실행할 때 인덱스 페이지가 꽉 차서 여유 공간이 없으면 인덱스 페이지 노드의 분할(split)이 발생하게 되는데, 이는 오퍼레이션 시간을 증가시켜 성능에 영향을 미친다.

index_unfill_factor는 인덱스를 생성할 때 각 인덱스 페이지 노드의 여유 공간을 확보하는 비율을 지정하는 파라미터이다. **index_unfill_factor** 설정값은 인덱스를 처음 생성할 때만 적용되며, 페이지에 지정된 빈 공간의 비율을 동적으로 유지하지 않는다. 값의 범위는 0부터 0.35까지이고 디폴트값은 **0.20**이다.

인덱스를 생성할 때 인덱스의 페이지 노드에 여유 공간이 없이(**index_unfill_factor=0**) 생성한다면, 추가로 삽입할 때마다 매번 인덱스 페이지 노드의 분할이 발생하여 성능에 영향을 끼치게 된다.

index_unfill_factor 값이 크면 인덱스 생성 시 노드 여유 공간을 많이 확보한다. 따라서 최초 인덱스 생성 후 노드 여유 공간이 꽉 찰 때까지 상대적으로 긴 시간 동안 인덱스 노드의 분할이 발생하지 않으므로, 상대적으로 성능이 나을 수 있다.

이 값이 작으면 인덱스 생성 시 노드 여유 공간이 작기 때문에, 인덱스 노드의 여유 공간이 금방 꽉 차게 될 가능성이 높으므로, 상대적으로 **INSERT**나 **UPDATE**에 의한 인덱스 노드 분할의 발생 가능성이 높다.

데이터베이스 서버 설정값 변경

환경 설정 파일 편집

\$CUBRID/conf 디렉터리에 있는 데이터베이스 설정 파일(**cubrid.conf**)을 직접 편집하여 데이터베이스 파라미터를 추가 및 삭제할 수 있으며, 파라미터의 설정값을 변경할 수 있다.

설정 파일에서 파라미터를 설정할 때 파라미터 구문 규칙은 다음과 같다.

- 파라미터 이름은 대/소문자를 구분하지 않는다.
- 파라미터 이름과 설정값은 동일한 라인에 입력되어야 한다.
- 파라미터 값을 설정하기 위해 등호(=)를 사용하며, 등호 기호의 양 옆에는 공백 문자를 사용할 수 있다.
- 파라미터 값이 문자열인 경우 따옴표 없이 문자열만 입력한다. 다만, 해당 문자열에 공백 문자가 포함된 경우에는 따옴표를 사용한다.

SQL 문을 사용

설명

SQL 문을 이용하여 CSQL 인터프리터나 CUBRID 매니저의 질의 편집기에서 데이터베이스 파라미터의 값을 설정할 수 있다. 다만, 클라이언트 파라미터만 갱신할 수 있음에 유의한다.

구문

```
SET SYSTEM PARAMETERS 'parameter_name=value [{; name=value}...]'
```

*parameter_name*은 설정값 변경이 가능한 클라이언트 파라미터 이름이고, *value*는 해당 파라미터의 값을 의미한다. 세미콜론(;)으로 구분하여 여러 개의 파라미터 값을 변경할 수 있다.

예제

다음은 인덱스 스캔 작업의 결과를 OID 순으로 가져오고, CSQL 인터프리터에서 히스토리 내역으로 저장하는 쿼리 개수를 70개로 설정하는 예제이다.

```
SET SYSTEM PARAMETERS 'r;csql_history_num=70; index_scan_in_oid_order=1'
```

CSQL 인터프리터의 세션 명령어 사용

설명

CSQL 인터프리터 내에서 세션 명령어(**SEt**)를 이용하여 데이터베이스 파라미터의 값을 설정할 수 있다. 다만, 클라이언트 파라미터만 갱신할 수 있음에 유의한다.

예제

다음은 데이터 정의문 수행이 허용되지 않도록 **block_ddl_statement** 파라미터를 1로 설정하는 예제이다.

```
csql> ;se block ddl statement=1
=== Set Param Input ===
block_ddl_statement=1
```

브로커 설정

cubrid_broker.conf 설정 파일과 기본 제공 파라미터

브로커 시스템 파라미터

다음은 **cubrid_broker.conf** 설정 파일에 사용할 수 있는 브로커 파라미터이다. 각 파라미터에 대한 설명은 [브로커별 파라미터](#)에서 **파라미터 설명**을 참조한다.

파라미터 이름	타입	디폴트 값
MASTER_SHM_ID	int	30001
ADMIN_LOG_FILE	string	log/broker/cubrid_broker.log
SERVICE	string	ON
BROKER_PORT	int	30000(최대값 : 65535)
MIN_NUM_APPL_SERVER	int	5
MAX_NUM_APPL_SERVER	int	40
APPL_SERVER_SHM_ID	int	30000
APPL_SERVER_MAX_SIZE	int	Windows : 40 (32 bit), 80 (64 bit) Linux : 0
LOG_DIR	string	log/broker/sql_log
ERROR_LOG_DIR	string	log/broker/error_log
SQL_LOG	string	ON
TIME_TO_KILL	int	120
SESSION_TIMEOUT	int	300
KEEP_CONNECTION	string	AUTO
ACCESS_LIST	string	-
ACCESS_LOG	string	ON
APPL_SERVER_PORT	int	BROKER_PORT+1
APPL_SERVER	string	CAS
LOG_BACKUP	string	OFF
SQL_LOG_MAX_SIZE	int	100000
MAX_STRING_LENGTH	int	-1

SOURCE_ENV	string	cubrid.env
STATEMENT_POOLING	string	ON
CCI_PCONNECT	string	OFF
SELECT_AUTO_COMMIT	string	OFF
LONG_QUERY_TIME	int	60
LONG_TRANSACTION_TIME	int	60
ACCESS_MODE	string	RW

기본 제공 파라미터

CUBRID 설치 시 생성되는 기본 브로커 설정 파일인 **cubrid_broker.conf**에는 브로커 파라미터 중에서 반드시 변경해야 할 일부 파라미터가 디폴트로 포함된다. 디폴트로 포함되지 않는 파라미터의 설정값을 변경하기 원할 경우 직접 추가/편집해서 사용하면 된다.

다음은 설치 시 기본으로 제공되는 **cubrid_broker.conf** 파일 내용이다.

```
[broker]
MASTER_SHM_ID      =30001
ADMIN_LOG_FILE      =log/broker/cubrid_broker.log

[%query editor]
SERVICE            =ON
BROKER_PORT         =30000
MIN_NUM_APPL_SERVER =5
MAX_NUM_APPL_SERVER =40
APPL_SERVER_SHM_ID  =30000
LOG_DIR             =log/broker/sql_log
ERROR_LOG_DIR       =log/broker/error_log
SQL_LOG             =ON
TIME_TO_KILL        =120
SESSION_TIMEOUT     =300
KEEP_CONNECTION     =AUTO

[%BROKER1]
SERVICE            =ON
BROKER_PORT         =33000
MIN_NUM_APPL_SERVER =5
MAX_NUM_APPL_SERVER =40
APPL_SERVER_SHM_ID  =33000
LOG_DIR             =log/broker/sql_log
ERROR_LOG_DIR       =log/broker/error_log
SQL_LOG             =ON
TIME_TO_KILL        =120
SESSION_TIMEOUT     =300
KEEP_CONNECTION     =AUTO
```

브로커 설정 파일 관련 환경 변수

CUBRID_BROKER_CONF_FILE 환경 변수를 사용하여 **cubrid_broker.conf** 파일의 위치를 지정할 수 있다.

서로 다른 구성으로 여러 개의 브로커를 실행할 때 사용한다.

공통 적용 파라미터

다음은 브로커 전체에 공통으로 적용되는 파라미터로 [broker] 아래에 작성된다.

MASTER_SHM_ID

MASTER_SHM_ID는 CUBRID 브로커를 관리하기 위해 사용되는 공유 메모리의 ID를 설정하는 파라미터로서, 이 값은 시스템 내에서 유일한 값이어야 한다. 디폴트 값은 **30001**로 설정된다.

ADMIN_LOG_FILE

ADMIN_LOG_FILE은 CUBRID 브로커의 구동에 관한 시간 기록을 저장하는 파일을 지정하기 위한 파라미터로서, 디폴트 값은 **log/broker/cubrid_broker.log** 파일이다.

브로커별 파라미터

다음은 브로커에 개별적으로 적용되는 파라미터로 [%*broker_name*] 아래에 각각 작성된다.

SERVICE

SERVICE는 해당 브로커의 구동 여부를 결정하기 위한 파라미터로, **ON** 또는 **OFF**의 값으로 설정된다. 디폴트 값은 **ON**이며, 이 파라미터가 **ON**으로 설정된 경우에만 해당 브로커를 구동할 수 있다.

BROKER_PORT

BROKER_PORT는 해당 브로커의 포트 번호를 지정하기 위한 파라미터로 시스템 내에서 유일한 값이면서 65535 이하의 값이어야 한다. **query_editor**의 브로커 포트는 디폴트 값이 **30000**으로 설정되며, **broker1**의 브로커 포트는 디폴트 값이 **33000**으로 설정된다.

MIN_NUM_APPL_SERVER

MIN_NUM_APPL_SERVER는 CUBRID 브로커가 생성할 수 있는 응용서버(CAS, CUBRID Application Server)의 최소 개수를 지정하는 파라미터로 디폴트 값은 **5**이다.

MAX_NUM_APPL_SERVER

MAX_NUM_APPL_SERVER는 CUBRID 브로커가 생성할 수 있는 응용서버(CAS)의 최대 개수를 지정하는 파라미터이며, 디폴트 값은 **40**이다.

APPL_SERVER_SHM_ID

APPL_SERVER_SHM_ID는 응용서버(CAS)가 이용하는 공유 메모리 ID를 지정하기 위한 파라미터로 시스템 내에서 유일한 값이어야 한다. 디폴트 값은 해당 브로커의 포트와 동일한 값이다.

APPL_SERVER_MAX_SIZE

APPL_SERVER_MAX_SIZE는 응용서버(CAS)가 처리하는 프로세스 메모리 사용량의 최대 크기를 지정하는 파라미터로 단위는 MB이다. 이 파라미터의 값은 응용서버(CAS)를 강제로 재구동하는 정책에 영향을 미치므로 서버 운영 환경을 고려하여 설정해야 한다. 특히, 값을 너무 작게 설정하면 응용서버가 빈번하게 재구동될 수 있으므로 주의한다. 또한, 이 파라미터는 Windows 버전과 Linux 버전의 디폴트 값이 다르므로 주의한다.

Windows 버전의 CUBRID는 32비트 버전에서는 **APPL_SERVER_MAX_SIZE**의 디폴트 값이 40(MB)이고, 64비트 버전에서는 80(MB)이다. 현재 프로세스의 크기가 **APPL_SERVER_MAX_SIZE**의 값을 초과하면, 브로커가 해당 응용 서버를 재구동한다.

Linux 버전의 CUBRID는 **APPL_SERVER_MAX_SIZE**의 디폴트 값이 0이고, 다음의 경우에 해당 응용 서버를 재구동한다.

- **APPL_SERVER_MAX_SIZE**의 값이 0 또는 음수인 경우: 현재 프로세스의 크기가 응용 서버의 초기 메모리의 2배가 될 때
- **APPL_SERVER_MAX_SIZE**의 값이 양수인 경우: **APPL_SERVER_MAX_SIZE**의 설정 값을 초과할 때

LOG_DIR

LOG_DIR은 SQL 로그가 저장되는 디렉터리를 지정하는 파라미터로, 디폴트 값은 **log/broker/sql_log**이다. SQL 로그가 기록되는 파일명은 *broker_name_id.sql.log*이다.

ERROR_LOG_DIR

ERROR_LOG_DIR은 브로커에 대한 에러 로그가 저장되는 디렉터리를 지정하는 파라미터로, 디폴트 값은 **log/broker/error_log**이다. 브로커 에러 로그 파일명은 *broker_name_id.err*이다.

SQL_LOG

SQL_LOG는 응용 클라이언트의 요청에 따라 응용서버(CAS)가 처리한 SQL 문에 대해 어떤 로그를 기록할 것인지 결정하는 파라미터로 디폴트 값은 **ON**이다. 이 파라미터가 **ON**으로 설정되면, 모든 로그를 기록한다. SQL 로그가 기록되는 파일명은 *broker_name_id.sql.log*이며, 설치 디렉터리의 **log/broker/sql_log** 디렉터리에 생성된다. 파라미터 값은 다음과 같다.

- **OFF** : 모든 로그를 기록하지 않음
- **ERROR** : 에러가 발생한 질의에 대한 로그만 기록
- **NOTICE** : 설정된 시간을 초과한 장기 실행 질의/트랜잭션의 로그, 에러가 발생한 질의에 대한 로그 기록
- **TIMEOUT** : 설정된 시간을 초과한 장기 실행 질의/트랜잭션의 로그 기록
- **ON/ALL** : 모든 로그 기록

TIME_TO_KILL

TIME_TO_KILL은 자동 추가된 응용서버 중 유휴 상태의 응용서버(CAS)를 제거하기 위한 기준 시간을 설정하는 파라미터로 디폴트 값은 **120(sec)**이다. 유휴 상태란 작업이 없이 쉬고 있는 상태를 말한다. 이 상태가 **TIME_TO_KILL** 시간 이상 유지되면 응용서버(CAS)의 제거/추가가 발생한다.

이 파라미터에 설정된 값은 자동 추가된 응용서버에만 적용되므로 **AUTO_ADD_APPL_SERVER** 파라미터가 **ON**인 경우에만 적용된다. **TIME_TO_KILL** 파라미터의 값을 너무 작게 설정하면 응용서버(CAS)의 제거/추가가 너무 빈번하게 발생할 수 있으므로 주의한다.

SESSION_TIMEOUT

SESSION_TIMEOUT은 해당 브로커의 세션을 종료하기 위한 타임아웃 값을 설정하는 파라미터로 디폴트 값은 **300(sec)**이다. 이 파라미터의 설정값을 초과하는 시간 동안 작업 요청에 응답이 없는 경우 해당 세션은 종료된다.

KEEP_CONNECTION

KEEP_CONNECTION은 응용서버(CAS)와 응용 클라이언트 사이의 연결 방식을 지정하는 파라미터로 **ON/OFF/AUTO** 중 하나로 설정된다. 이 파라미터가 **OFF**로 설정되면 클라이언트는 트랜잭션 단위로 응용서버와 연결하고, **ON**으로 설정되면 커넥션 단위로 응용서버와 연결한다. 또한 **AUTO**로 설정되면 응용서버의 개수가 클라이언트 개수보다 많은 경우 커넥션 단위로 연결하고, 응용서버의 개수가 클라이언트의 개수보다 적은 경우 트랜잭션 단위로 연결한다. 디폴트 값은 **AUTO**이다.

ACCESS_LIST

ACCESS_LIST는 CUBRID 브로커로 접근을 허용하는 응용 클라이언트의 IP 주소 리스트를 저장할 파일 이름을 지정하는 파라미터이다. 210.192.33.*와 210.194.34.*인 IP 주소의 접근을 허용하려면 이를 임의의 파일(ip_lists.txt)에 저장하여 이 파라미터의 값으로 파일명을 설정한다.

ACCESS_LOG

ACCESS_LOG는 해당 브로커의 접속 로그를 저장할 것인지 지정하는 파라미터로 디폴트 값은 **ON**이다. 브로커 접속 로그 파일명은 *broker_name.id.access*이고, **LOG_DIR** 파라미터에서 설정한 디렉터리에 저장된다.

APPL_SERVER_PORT

APPL_SERVER_PORT는 Windows 운영체제에서만 추가할 수 있는 파라미터로 응용 클라이언트와 통신하는 응용서버(CAS)의 통신 포트를 지정하는 파라미터이다. 디폴트 값은 **BROKER_PORT** 파라미터 값에 1을 더한 값으로 설정되며, 응용서버의 최대 개수가 **cubrid_broker_conf**의 **MAX_NUM_APPL_SERVER** 파라미터에 의해 제한되므로 설정할 수 있는 응용서버(CAS)의 통신 포트의 개수 역시 최대 **MAX_NUM_APPL_SERVER**

파라미터의 설정값으로 제한된다. Windows 운영체제에서 응용 클라이언트와 CUBRID 브로커 사이에 방화벽이 존재하는 경우에는 반드시 **BROKER_PORT** 및 **APPL_SERVER_PORT**에서 설정된 통신 포트를 열어야 한다.

APPL_SERVER

APPL_SERVER는 CUBRID 브로커가 생성하고 관리하는 응용서버의 종류를 지정하는 파라미터로 디폴트 값은 **CAS**이다.

LOG_BACKUP

LOG_BACKUP은 브로커의 접속 로그 파일과 에러 로그 파일을 백업할 것인지 지정하는 파라미터이다. 디폴트 값은 **OFF**이며, 이 파라미터가 **ON**으로 설정되면 CUBRID 브로커가 종료될 때 접속 로그와 에러 로그의 백업을 수행한다. 이 때, 접속 로그의 백업 파일명은 *broker_name_id.access*가 되고, 에러 로그의 백업 파일명은 *broker_name_id.error*가 된다.

SQL_LOG_MAX_SIZE

SQL_LOG_MAX_SIZE는 SQL 로그 파일의 최대 크기를 지정하는 파라미터로 디폴트 값은 **100,000**(KB)이다. **SQL_LOG** 파라미터가 **ON**으로 설정된 경우에 생성되는 SQL 로그 파일이 이 파라미터의 설정값에 도달하면 *broker_name_id.sql.log.bak*이 생성된다.

MAX_STRING_LENGTH

MAX_STRING_LENGTH는 bit, varbit, char, varchar, nchar, nchar varying인 데이터 타입에 대해서 최대 스트링 길이를 지정하는 파라미터이다. 디폴트 값인 **-1**로 설정되면 데이터베이스에서 정의된 스트링 길이가 그대로 사용되고, 파라미터의 값이 **100**으로 설정되면 임의의 속성이 varchar(1000)으로 정의되었어도 100으로 정의된 것처럼 동작한다.

SOURCE_ENV

SOURCE_ENV는 브로커 각각에 대해 개별적으로 운영체제 환경 변수를 설정할 수 있는 파일을 정하는 파라미터로, 파일 확장자는 반드시 **env**여야 한다. **cubrid.conf**에서 지정하는 모든 파라미터는 환경 변수를 통해서도 설정할 수 있다. 예를 들어, **cubrid.conf**에서 **lock_timeout_in_secs**는 환경 변수 **CUBRID_LOCK_TIMEOUT_IN_SECS**로 지정할 수 있다. 또 다른 예로, broker1에서만 데이터 정의문 수행을 차단하려면 **SOURCE_ENV**에서 지정한 파일에 **CUBRID_BLOCK_DDL_STATEMENT**를 **1**로 설정하면 된다. 환경변수가 있을 경우, **cubrid.conf**보다 우선한다. 디폴트 값은 **cubrid.env**이다.

STATEMENT_POOLING

STATEMENT_POOLING은 statement 풀링 기능의 사용 여부를 설정하는 파라미터로 기본값은 **ON**이다.

CUBRID는 트랜잭션이 커밋 또는 롤백되는 경우, 해당 클라이언트 세션에 존재하는 prepared statement의 핸들을 모두 close하는데, **STATEMENT_POOLING**의 값이 **ON**인 경우에는 prepared statement의 핸들을 지속적으로 풀에 유지하므로 이를 재사용할 수 있다. 따라서, prepared statement를 재사용하는 일반 응용 프로그램 또는 statement pooling이 구현된 DBCP와 같은 라이브러리가 적용된 환경에서는 반드시 디폴트 설정(**ON**)을 유지해야 한다.

STATEMENT_POOLING의 값이 **OFF**인 상태에서 트랜잭션 커밋 또는 종료 이후 해당 prepared statement를 실행하는 경우에는 다음과 같은 메시지가 출력된다.

```
Caused by: cubrid.jdbc.driver.CUBRIDException: Attempt to access a closed Statement.
```

CCI_PCONNECT

CCI_PCONNECT는 CCI connection 풀링 기능의 사용 여부를 설정하는 파라미터로 기본값은 **OFF**이다.

SELECT_AUTO_COMMIT

SELECT_AUTO_COMMIT은 CCI 또는 PHP에서 **SELECT** 문에 대한 자동 커밋 모드를 설정하는 파라미터로, 기본값은 **OFF**이다. 단, 자동 커밋은 prepared statement가 n개인 경우, 서버로부터 n개 질의문 전부에 대해 결과 셋을 가져오기(fetch)한 시점에만 자동 커밋이 수행되므로 주의한다. 아래는

SELECT_AUTO_COMMIT의 값이 **ON**일 때, 상황에 따른 자동 커밋 여부에 관한 예제이며, 보다 상세한 설명은 [cci_end_tran](#)을 참고한다.

```
SELECT 1 prepare
SELECT 1 execute // AUTO COMMIT 0

SELECT 1 prepare
SELECT 2 prepare
SELECT 1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
SELECT 2 execute // AUTO COMMIT 0

SELECT 1 prepare
SELECT 1 execute // AUTO COMMIT 0
INSERT 1 prepare
INSERT 1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed

INSERT 1 prepare
INSERT 1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
SELECT 1 prepare
SELECT 1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed

SELECT 1 prepare
INSERT 1 prepare
SELECT 1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
INSERT 1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed

INSERT 1 prepare
SELECT 1 prepare
INSERT 1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
SELECT 1 execute // AUTO COMMIT X -> An EXPLICIT COMMIT needed
```

LONG_QUERY_TIME

LONG_QUERY_TIME은 장기 실행 질의(long-duration query)로 판단될 질의 실행 시간을 설정하는 파라미터이다. 디폴트 값은 **60(초)**이고 소수점을 사용하여 밀리초(msec) 단위의 값을 설정할 수 있다. 예를 들어 500밀리초로 설정하려면 값을 0.5로 설정한다. 파라미터 값이 0으로 설정되면 장기 실행 질의를 판단하지 않는다.

LONG_TRANSACTION_TIME

LONG_TRANSACTION_TIME은 장기 실행 트랜잭션(long-duration transaction)으로 판단될 트랜잭션의 실행 시간을 설정하는 파라미터이다. 디폴트 값은 **60(초)**이고 소수점을 사용하여 밀리초(msec) 단위의 값을 설정할 수 있다. 예를 들어 500밀리초로 설정하려면 값을 0.5로 설정한다. 파라미터 값이 0으로 설정되면 장기 실행 트랜잭션을 판단하지 않는다.

CUBRID 매니저 서버 설정

cm.conf 설정 파일과 기본 제공 파라미터

CUBRID 매니저 서버 시스템 파라미터

다음은 CUBRID 매니저 서버를 실행하는데 필요한 파라미터로 설정 파일(**cm.conf**)에서 편집할 수 있다.

파라미터 이름	타입	디폴트 값
cm_port	int	8001
monitor_interval	int	5
allow_user_multi_connection	string	YES
auto_start_broker	string	YES
server_long_query_time	int	10

파라미터 작성 규칙

CUBRID 매니저 서버 설정 파일에 적용되는 파라미터 구문 규칙은 다음과 같다.

- 파라미터 이름은 대/소문자를 구분하지 않는다.
- 파라미터 이름과 설정값은 동일한 라인에 입력해야 한다.
- 파라미터 값을 설정하기 위해 공백 문자 및 등호 기호(=)를 사용할 수 있다.
- 파라미터 값이 문자열인 경우 따옴표 없이 문자열만 입력한다. 다만, 해당 문자열에 공백 문자가 포함된 경우에만 따옴표를 사용한다.

기본 제공 파라미터

CUBRID 설치 시 생성되는 기본 CUBRID 매니저 서버용 설정 파일(**cm.conf**)에는 매니저 서버 파라미터 중 반드시 변경해야 할 일부 파라미터가 기본적으로 포함된다. 디폴트로 포함되지 않는 파라미터의 설정값을 변경하기 원할 경우 직접 추가/편집해서 사용하면 된다.

다음은 설치 시 기본으로 제공되는 **cm.conf** 파일 내용이다.

```
# cm.conf
#      -- CUBRID database management tool server configuration file
#
#
# When server starts, it looks for the environment variable
# 'CUBRID MANAGER' and use it to locate this file. It is assumed that
# 'CUBRID MANAGER' is the root directory of all CUBRID Manager related files.
#
#
# Port number designation
#
cm_port 8001
```

```
#
# Monitoring interval setting
#
monitor interval 5

#
# Allowing Multiple connection with one CUBRID Manager user.
#
allow_user_multi_connection YES

#####
# diagnostics parameter
#####
#
# turn on/off diag
#
#execute_diag ON

#
# server long query time (sec)
#
server_long_query_time 10
```

파라미터 설명

cm_port

cm_port는 CUBRID 매니저 서버와 매니저 클라이언트 사이에 이용되는 포트를 설정하기 위한 파라미터이다.

실제 사용되는 포트는 **cm_port** 파라미터에 의해 설정된 포트와 포트 번호에 1을 더한 포트 두 개가 사용된다. 예를 들어, 디폴트 값인 **8001**이 지정된 경우, 실제 이용되는 포트 번호는 8001, 8002이다.

cm_port는 **cub_auto**가 사용하며, **cm_port**+1은 **cub_js**가 사용한다.

monitor_interval

monitor_interval은 CUBRID 매니저 서버의 **cub_auto** 프로세스에 대한 모니터링 주기를 초 단위로 설정하는 파라미터로 디폴트 값은 **5(sec)**이고, 최소값은 **1(sec)**이다. **monitor_interval**이 짧을수록 시스템 부하가 커진다.

allow_user_multi_connection

allow_user_multi_connection은 CUBRID 매니저 서버에 여러 개의 매니저 클라이언트가 접속할 수 있도록 사용자의 중복 접속을 허용하는 파라미터이다. 디폴트 값인 **YES**로 설정하면 사용자는 동일한 사용자 이름을 가지고 두 개 이상의 CUBRID 매니저 클라이언트에서 하나의 매니저 서버에 접속할 수 있다.

auto_start_broker

auto_start_broker는 CUBRID 매니저 서버가 구동될 때 CUBRID 브로커도 자동으로 구동되도록 설정하는 파라미터이다. 디폴트 값인 **YES**로 설정하면 CUBRID 매니저 서버와 함께 CUBRID 브로커도 구동된다.

server_long_query_time

server_long_query_time은 데이터베이스에 대한 진단 항목 중 하나인 슬로우 쿼리(slow query)를 판단하기 위한 기준 시간을 설정하는 파라미터이다. 디폴트 값은 **10(sec)**이며, 이 경우 실행 시간이 10초를 초과하는 질의를 슬로우 쿼리로 판단한다. **server_long_query_time** 파라미터는 CUBRID 매니저를 통한 데이터베이스

진단 기능이 활성화된 경우에 적용되므로, **execute_diag** 파라미터가 **ON**으로 설정된 상태에서만 효력이 있다.

API 레퍼런스

이번 장에서는 다음과 같은 API에 대해 설명한다.

- JDBC API
- ODBC API
- OLE DB API
- PHP API
- CCI API

JDBC API

JDBC 프로그래밍

CUBRID JDBC 드라이버

CUBRID JDBC 드라이버는 Java로 작성된 응용 프로그램에서 CUBRID 데이터베이스를 접속할 수 있도록 해준다. CUBRID JDBC 드라이버 v1.0은 JDBC 2.0 스펙을 기준으로 개발되었다. CUBRID JDBC 드라이버는 CUBRID CAS를 통하여 DBMS와 연결하는 Pure Java Driver for Database Middleware (type 3)에 해당한다. CUBRID JDBC 드라이버는 JDK 1.5에서 컴파일한 것을 기본으로 제공하며, JDK 1.4나 JDK 1.6을 사용하는 경우에는 릴리스 노트나 JDBC 환경 설정 부분을 참고하기 바란다.

CUBRID JDBC 드라이버 버전 확인

JDBC 드라이버 버전은 다음과 같은 방법으로 확인할 수 있다.

```
% jar -tf cubrid.jdbc.jar
META-INF/
META-INF/MANIFEST.MF
cubrid/
cubrid/jdbc/
cubrid/jdbc/driver/
cubrid/jdbc/jci/
cubrid/sql/
CUBRID-JDBC-8.1.4.1032
cubrid/jdbc/driver/CUBRIDBlob.class
...
```

CUBRID JDBC 드라이버 등록

JDBC 드라이버 등록은 **Class.forName(driver-class-name)** 명령을 사용하며, 아래는 CUBRID JDBC 드라이버를 등록하기 위해 cubrid.jdbc.driver.CUBRIDDriver 클래스를 로드하는 예제이다.

```
import java.sql.*;
import cubrid.jdbc.driver.*;

public class LoadDriver {
    public static void main(String[] Args) {
        try {
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
        } catch (Exception e) {
            System.err.println("Unable to load driver.");
            e.printStackTrace();
        }
    }
    ...
}
```

JDBC 인터페이스

다음은 CUBRID에서 지원하는 JDBC 표준 인터페이스를 및 확장 인터페이스를 정리한 목록이다. JDBC 2.0 스펙에 포함된 메소드 중 일부는 지원하지 않으므로 프로그램 작성 시 주의한다.

JDBC 인터페이스 지원 여부

JDBC 표준 인터페이스	CUBRID 확장 인터페이스	지원 여부
---------------	-----------------	-------

java.sql.Blob	java.sql.CUBRIDBlob	지원
java.sql.CallableStatement	java.sql.CUBRIDCallableStatement	
java.sql.Clob	java.sql.CUBRIDClob	
java.sql.Connection	java.sql.CUBRIDConnection	
java.sql.DatabaseMetaData	java.sql.CUBRIDDatabaseMetaData	
java.sql.Driver	java.sql.CUBRIDDriver	
java.sql.PreparedStatement	java.sql.CUBRIDPreparedStatement	
java.sql.ResultSet	java.sql.CUBRIDResultSet	
java.sql.ResultSetMetaData	java.sql.CUBRIDResultSetMetaData	
java.sql.Statement	java.sql.CUBRIDStatement	JDBC 3.0 의 getGeneratedKeys() 메소드 지원
java.sql.DriverManager		지원
Java.sql.SQLException	Java.sql.CUBRIDException	지원
java.sql.Array		미지원
java.sql.ParameterMetaData		
java.sql.Ref		
java.sql.Savepoint		
java.sql.SQLData		
java.sql.SQLInput		
java.sql.Struct		

연결 설정

DriverManager는 JDBC 드라이버를 관리하기 위한 기본적인 서비스이며, 데이터베이스 드라이버를 선택하고 새로운 데이터베이스 연결을 생성하는 기능을 한다. CUBRID JDBC 드라이버가 등록되어 있다면 **DriverManager.getConnection(db-url, user-id, password)** 메소드를 호출하여 데이터베이스에 접속한다. **getConnection** 메소드는 **Connection** 객체를 반환한다. 그리고 그것은 질의 실행과 명령문 실행 그리고 트랜잭션의 커밋 또는 롤백에 사용된다. 연결 설정을 위한 *db-url* 인자의 구성은 다음과 같다.

```
jdbc:cubrid:<host>:<port>:<db-name>:[user-id]:[password]:[?<property> [& <property>]]

<host> ::=
hostname | ip_address

<property> ::=
althosts= <alternative_hosts> | rctime= <second> | charset= <character_set>

<alternative_hosts> :
<standby_broker1_host>:<port> [, <standby_broker2_host>:<port>]
```

- *<host>* : CUBRID 브로커가 동작하고 있는 서버의 IP 주소 또는 호스트 이름
- *<port>* : CUBRID 브로커의 포트 번호(디폴트 값: 33000)
- *<db-name>* : 접속할 데이터베이스 이름

- `[user-id]` : 데이터베이스에 접속할 사용자 ID이다. 기본적으로 데이터베이스에는 **dba**와 **public** 두 개의 사용자가 존재한다. 빈 문자열("")을 입력하면 **public** 사용자로 데이터베이스에 접속한다.
- `[password]` : 데이터베이스에 접속할 사용자의 암호이다. 해당 사용자에게 암호가 설정되어 있지 않으면, 빈 문자열("")을 입력한다.
- `althosts` : HA 환경에서 장애 시 fail-over할 하나 이상의 standby 브로커의 호스트 IP와 접속 포트이다.
- `rctime` : HA 환경에서 장애 복구 시 active 브로커로 연결을 시도하는 주기를 초 단위로 입력한다. 이에 관한 상세한 설명은 [HA 관련 JDBC 환경 설정](#) 및 아래 예제를 참고한다.
- `charset` : 접속하고자 하는 DB의 문자 세트(charset)이다.

예제 1

```
--connection URL string when user name and password omitted
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::

--connection URL string when charset property specified
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::charset=utf-8

--connection URL string when a property(althosts) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::althosts=127.0.0.2:31000,127.0.0.3:31000

--connection URL string when properties(althosts,rctime) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::althosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600

--connection URL string when properties(althosts,rctime, charset) specified for HA
URL=jdbc:CUBRID:127.0.0.1:31000:db1:::althosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600
&charset=utf-8
```

예제 2

```
String url = "jdbc:cubrid:127.0.0.1:33000:demodb:::";
String userid = "";
String password = "";

try {
    Connection conn =
        DriverManager.getConnection(url,userid,password);
    // Do something with the Connection
    ...

} catch (SQLException e) {
    System.out.println("SQLException:" + e.getMessage());
    System.out.println("SQLState: " + e.getSQLState());
}
...
```

참고 트랜잭션 롤백 요청을 하는 rollback 메소드는 서버가 롤백 작업을 완료한 후 종료된다.

OID 와 컬렉션 사용

JDBC 스펙에 정의된 메소드 이외에 CUBRID JDBC 드라이버에서 추가로 OID, 컬렉션(set, multiset, sequence), GLO 등을 다루는 메소드를 제공한다.

이 메소드의 사용을 위해서는 기본적으로 import하는 CUBRID JDBC 드라이버 클래스 이외에 **cubrid.sql.***를 import해야 한다. 또한 표준 JDBC API에서 제공하는 **ResultSet** 클래스가 아닌 **CUBRIDResultSet** 클래스로 변환하여 결과를 받아야 한다.

```
import cubrid.jdbc.driver.* ;
import cubrid.sql.* ;
...
CUBRIDResultSet urs = (CUBRIDResultSet) stmt.executeQuery(
"SELECT city FROM location");
```

주의 CUBRID의 확장 API를 사용하면, **AUTOCOMMIT**을 TRUE로 설정하였더라도 자동으로 커밋되지 않는다. 따라서 항상 open한 연결에 대해 명시적으로 커밋을 해야 한다. CUBRID 확장 API는 OID, 컬렉션, GLO 등을 다루는 메소드이다.

OID 사용

OID를 사용할 때 다음의 규칙을 지켜야 한다.

- **CUBRIDOID**를 사용하기 위해서는 반드시 **cubrid.sql.***를 import 해야 한다. (a)
- **SELECT** 문에 클래스명을 주어 OID를 가져올 수 있다. 물론 다른 속성과 혼용해서 사용할 수도 있다. (b)
- 질의에 대한 **ResultSet**은 반드시 **CUBRIDResultSet**으로 받아야 한다. (c)
- **CUBRIDResultSet**에서 OID를 가져오는 메소드는 **getOID()**이다. (d)
- OID에서 값을 가져오기 위해서는 **getValues()** 메소드를 통해 가져올 수 있다. 그 결과는 **ResultSet**이다. (e)
- OID에 값을 대입하기 위해서는 **setValues()** 메소드를 통해서 적용할 수 있다. (f)
- 확장 API 사용시에는 연결에 대해 항상 **commit()**을 해주어야 한다. (g)

```
import java.sql.*;
import cubrid.sql.*; //a
import cubrid.jdbc.driver.*;

/*
CREATE TABLE oid_test(
    id INTEGER,
    name VARCHAR(10),
    age INTEGER
);

INSERT INTO oid test VALUES(1, 'Laura', 32);
INSERT INTO oid test VALUES(2, 'Daniel', 39);
INSERT INTO oid test VALUES(3, 'Stephen', 38);
*/

class OID Sample
{
    public static void main (String args [])
    {
        // Making a connection
        String url= "jdbc:cubrid:localhost:33000:demodb::";
        String user = "dba";
        String passwd = "";

        // SQL statement to get OID values
        String sql = "SELECT oid test from oid test"; //b
        // columns of the table
        String[] attr = { "id", "name", "age" } ;

        // Declaring variables for Connection and Statement
        Connection con = null;
        Statement stmt = null;
        CUBRIDResultSet rs = null;
        ResultSetMetaData rsmd = null;

        try {
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
```

```

    } catch (ClassNotFoundException e) {
        throw new IllegalStateException("Unable to load Cubrid driver", e);
    }

    try {
        con = DriverManager.getConnection(url, user, passwd);
        stmt = con.createStatement();
        rs = (CUBRIDResultSet)stmt.executeQuery(sql); //c
        rsmd = rs.getMetaData();

        // Printing columns
        int numOfColumn = rsmd.getColumnCount();
        for (int i = 1; i <= numOfColumn; i++) {
            String ColumnName = rsmd.getColumnName(i);
            String JdbcType = rsmd.getColumnTypeName(i);
            System.out.print(ColumnName);
            System.out.print("(" + JdbcType + ")");
            System.out.print(" | ");
        }
        System.out.print("\n");

        // Printing rows
        CUBRIDResultSet rsoid = null;
        int k = 1;

        while (rs.next()) {
            CUBRIDOID oid = rs.getOID(1); //d
            System.out.print("OID");
            System.out.print(" | ");
            rsoid = (CUBRIDResultSet)oid.getValues(attr); //e

            while (rsoid.next()) {
                for( int j=1; j <= attr.length; j++) {
                    System.out.print(rsoid.getObject(j));
                    System.out.print(" | ");
                }
                System.out.print("\n");

                // New values of the first row
                Object[] value = { 4, "Yu-ri", 19 };
                if (k == 1) oid.setValues(attr, value); //f

                k = 0;
            }
            con.commit(); //g
        } catch (CUBRIDException e) {
            e.printStackTrace();
        } catch (SQLException ex) {
            ex.printStackTrace();
        } finally {
            if(rs != null) try { rs.close(); } catch (SQLException e) {}
            if(stmt != null) try { stmt.close(); } catch (SQLException e) {}
            if(con != null) try { con.close(); } catch (SQLException e) {}
        }
    }
}

```

컬렉션 사용

아래 예제 1의 'a'에 해당하는 부분이 **CUBRIDResultSet**으로부터 컬렉션 타입의 데이터를 가져오는 부분으로 그 결과는 배열 형태로 반환한다.

예제 1

```

import java.sql.*;
import java.lang.*;
import cubrid.sql.*;
import cubrid.jdbc.driver.*;

// create class collection_test(
// settest set(integer),
// multisettest multiset(integer),
// listtest list(Integer)
// );
//

// insert into collection_test values({1,2,3},{1,2,3},{1,2,3});
// insert into collection_test values({2,3,4},{2,3,4},{2,3,4});
// insert into collection_test values({3,4,5},{3,4,5},{3,4,5});

class Collection_Sample
{
    public static void main (String args [])
    {
        String url= "jdbc:cubrid:210.216.33.250:43300:demodb:::";
        String user = "";
        String passwd = "";
        String sql = "select settest,multisettest,listtest from collection test";
        try {
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            Connection con = DriverManager.getConnection(url,user,passwd);
            Statement stmt = con.createStatement();
            CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);
            CUBRIDResultSetMetaData rsmd = (CUBRIDResultSetMetaData) rs.getMetaData();
            int numofColumn = rsmd.getColumnCount();
            while (rs.next()) {
                for (int j=1; j<=numofColumn; j++) {
                    Object[] reset = (Object[]) rs.getCollection(j); //a
                    for (int m=0 ; m < reset.length ; m++)
                        System.out.print(reset[m] +",");
                    System.out.print(" | ");
                }
                System.out.print("\n");
            }
            rs.close();
            stmt.close();
            con.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

예제 2

```

import java.sql.*;
import java.io.*;
import java.lang.*;
import cubrid.sql.*;
import cubrid.jdbc.driver.*;

// create class collection_test(
// settest set(integer),
// multisettest multiset(integer),
// listtest list(Integer)
// );
//

// insert into collection_test values({1,2,3},{1,2,3},{1,2,3});
// insert into collection_test values({2,3,4},{2,3,4},{2,3,4});
// insert into collection_test values({3,4,5},{3,4,5},{3,4,5});

class SetOP_Sample

```

```

{
    public static void main (String args [])
    {
        String url = "jdbc:cubrid:127.0.0.1:33000:demodb::";
        String user = "";
        String passwd = "";
        String sql = "select collection_test from collection_test";
        try {
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            CUBRIDConnection con =(CUBRIDConnection)
            DriverManager.getConnection(url,user,passwd);
            Statement stmt = con.createStatement();
            CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);
            while (rs.next ()) {
                CUBRIDOID oid = rs.getOID(1);
                oid.addToSet("settest",new Integer(10));
                oid.addToSet("multisettest",new Integer(20));
                oid.addToSequence("listtest",1,new Integer(30));
                oid.addToSequence("listtest",100,new Integer(100));
                oid.putIntoSequence("listtest",99,new Integer(99));
                oid.removeFromSet("settest",new Integer(1));
                oid.removeFromSet("multisettest",new Integer(2));
                oid.removeFromSequence("listtest",99);
                oid.removeFromSequence("listtest",1);
            }
            con.commit();
            rs.close();
            stmt.close();
            con.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

GLO 사용

GLO를 사용하려면 DriverManager를 통해 연결할 때 반드시 **CUBRIDConnection**으로 해야 한다.

CUBRIDConnection으로부터 얻은 연결은 **getNewGLO()** 메소드를 사용해서 **FileInputStream** 객체를 입력할 수 있다. 그 결과로 **CUBRIDOID** 클래스의 인스턴스가 반환된다.

예제 1

```

import java.sql.*;
import java.io.*;
import java.lang.*;
import cubrid.jdbc.driver.*;
import cubrid.sql.*;

//-----
// create class filetest under glo
//
// assumes that testin1, testin2 files are in the location
// of the execution of this program.
// result of the execution will create in the location
// of execution,
//-----

class GLO_Sample
{
    public static void main (String args [])
    {
        String url = "jdbc:cubrid:127.0.0.1:33000:demodb::";

```



```

String user = "";
String passwd = "";
try {
    Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
} catch (Exception e) {
    e.printStackTrace();
}
try {
    CUBRIDConnection con = (CUBRIDConnection)
        DriverManager.getConnection(url, user, passwd);
    FileInputStream inputfile = new FileInputStream("testin");
    CUBRIDOID oid = con.getNewGLO("filetest", inputfile);
    inputfile.close();
    System.out.println("getNewGLO1..");
    FileOutputStream outputfile = new FileOutputStream("testout");
    oid.loadGLO(outputfile);
    outputfile.close();
    System.out.println("loadGLO1..");
    inputfile = new FileInputStream("testin2");
    oid.saveGLO(inputfile);
    inputfile.close();
    System.out.println("saveGLO2..");
    outputfile = new FileOutputStream("testout2");
    oid.loadGLO(outputfile);
    outputfile.close();
    System.out.println("loadGLO2..");
    con.commit();
    con.close();
} catch (SQLException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

예제 2

```

import java.sql.*;
import java.lang.*;
import cubrid.jdbc.driver.*;
import cubrid.sql.*;

class SetOID Sample
{
    public static void main (String args [])
    {
        String url = "jdbc:cubrid:127.0.0.1:33000:demodb::";
        String user = "";
        String passwd = "";
        String usql = "update location set city='Pusan' where location=?";
        String sql = "select location, city from location";

        try {
            Class.forName("cubrid.jdbc.driver.CUBRIDDriver");
        } catch (Exception e) {
            e.printStackTrace();
        }
        try {
            CUBRIDConnection con = (CUBRIDConnection)
                DriverManager.getConnection(url, user, passwd);
            Statement stmt = con.createStatement();
            CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);
            CUBRIDOID oid = null;

            rs.next()
            oid = rs.getOID(1);

            CUBRIDPreparedStatement ps = (CUBRIDPreparedStatement)
                con.prepareStatement(usql);

            ps.setOID(1, oid); // OID of the last tuple

```

```

        ps.executeUpdate();
        rs.close();
        stmt.close();
        con.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}

```

자동 증가 특성의 컬럼 값 검색

자동 증가 특성

자동 증가 특성(AUTO_INCREMENT)은 자동으로 각 행의 숫자 값을 증가 생성하는 컬럼에 대한 특성으로서, 보다 자세한 사항은 테이블 생성의 [컬럼 정의](#)를 참고한다. 수치형 도메인(SMALLINT, INTEGER, DECIMAL(p , 0), NUMERIC(p , 0))에 대해서만 정의할 수 있다.

자동 증가 특성은 JDBC 프로그램에서 자동 생성된 키로 인식되고, 이 키의 검색을 사용하려면 자동 생성된 키 값을 검색할 행을 삽입할 시기를 표시해야 한다. 이를 수행하기 위하여

Connection.prepareStatement와 **Statement.execute** 메소드를 호출하여 플래그를 설정해야 한다. 이때, 실행된 명령문은 **INSERT** 문 또는 **INSERT** within **SELECT** 문이어야 하며, 다른 명령문의 경우 JDBC 드라이버가 플래그를 설정하는 매개변수를 무시한다.

수행 단계

- 다음 방법 중 하나를 사용하여 자동 생성된 키를 반환하려는지 표시한다. 자동 증가 특성 컬럼을 지원하는 데이터베이스 서버의 테이블에 대해 다음의 양식을 사용하며, 각 양식은 단일 행 **INSERT** 문에 대해서만 적용 가능하다.
 - 아래와 같이 **PreparedStatement** 오브젝트를 작성한다.


```
Connection.prepareStatement(sql statement, Statement.RETURN_GENERATED_KEYS);
```
 - **Statement.execute** 메소드를 사용하여 행을 삽입할 경우, 아래와 같이 **Statement.execute** 메소드를 사용한다.


```
Statement.execute(sql statement, Statement.RETURN_GENERATED_KEYS);
```
- **PreparedStatement.getGeneratedKeys** 메소드 또는 **Statement.getGeneratedKeys** 메소드를 호출하여 자동 생성된 키 값이 포함된 **ResultSet** 오브젝트를 검색한다. **ResultSet**에서 자동 생성된 키의 데이터 유형은 해당 도메인의 데이터 유형에 상관 없이 **DECIMAL**이다.

예제

다음 예제는 자동 증가 특성이 있는 테이블을 생성하고, 데이터를 테이블에 입력하여, 자동 증가 특성 컬럼에 자동 생성된 키 값이 입력되고 해당 키값이 **Statement.getGeneratedKeys()** 메소드를 통해 정상적으로 검색되는지를 점검하는 예제이다. 앞서 설명한 단계에 해당하는 명령문의 코멘트에 각 단계를 표시하였다.

```

import java.sql.*;
import java.math.*;
import cubrid.jdbc.driver.*;

```

```

Connection con;
Statement stmt;
ResultSet rs;
java.math.BigDecimal idColVar;
...
stmt = con.createStatement();    // Create a Statement object

stmt.executeUpdate(
"CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), "
+ "IDENTCOL INTEGER AUTO_INCREMENT)"); // Create table with identity column

stmt.execute(
"INSERT INTO EMP_PHONE (EMPNO, PHONENO) "
+ "VALUES ('000010', '5555')",          // Insert a row <Step 1>
Statement.RETURN_GENERATED_KEYS);      // Indicate you want automatically

rs = stmt.getGeneratedKeys();           // generated keys
// Retrieve the automatically <Step 2>
// generated key value in a ResultSet.
// Only one row is returned.
// Create ResultSet for query

while (rs.next()) {
    java.math.BigDecimal idColVar = rs.getBigDecimal(1);
    // Get automatically generated key
    // value
    System.out.println("automatically generated key value = " + idColVar);
}
rs.close();                             // Close ResultSet
stmt.close();                           // Close Statement

```

CUBRIDConnection

개요

표준 **Connection** 클래스가 확장된 **CUBRIDConnection** 클래스는 다음과 같은 추가된 메소드를 가지고 있다.

반환 타입	메소드 이름
CUBRIDOID	<code>getNewGLO(String className, InputStream stream)</code>
CUBRIDOID	<code>getNewGLO(String className, InputStream stream, int length)</code>

getNewGLO

설명

명시된 *className*, 저장할 *stream*으로 새로운 GLO를 만든다. 저장할 스트림의 길이를 *length*로 지정할 수 있다.

구문

```

CUBRIDOID getNewGLO(String className, InputStream stream)
CUBRIDOID getNewGLO(String className, InputStream stream, int length)

```

리턴 값

- GLO가 저장된 객체의 **CUBRIDOID**

예제

```
// 예제에 사용된 스키마:
//
// create class filetest under glo;
//

CUBRIDConnection con = (CUBRIDConnection)
    DriverManager.getConnection(url,user,passwd);

// Input file name : "testin1"
FileInputStream inputfile = new FileInputStream("testin1");
CUBRIDOID oid = con.getNewGLO("filetest", inputfile);
inputfile.close();

con.commit();
// CUBRIDOID사용시에는 자동 커밋 기능을 사용할 수 없기 때문에
// 이 트랜잭션 종료를 위해 커밋을 명시해야 한다.
```

CUBRIDOID

개요

CUBRIDOID 클래스는 OID를 다루기 위해 다음과 같은 메소드를 가지고 있다.

반환 타입	메소드 이름
void	addToSequence(String attrName, int index, Object value)
void	addToSet(String attrName, Object value)
static CUBRIDOID	getNewInstance(CUBRIDConnection con, String oidStr)
String	getOidString()
String	getTableName()
ResultSet	getValues(String[] attrNames)
Boolean	isInstance()
void	loadGLO(OutputStream stream)
void	putIntoSequence(String attrName, int index, Object value)
void	remove()
void	removeFromSequence(String attrName, int index)
void	removeFromSet(String attrName, Object value)
void	saveGLO(InputStream stream)
void	saveGLO(InputStream stream, int length)
void	setReadLock()
void	setValues(String[] attrNames, Object[] values)
void	setWriteLock()

addToSequence

설명

CUBRIDOID에 해당하는 인스턴스의 *attrName*에 명시된 이름에 해당하는 **SEQUENCE** 속성 중 *value*에 명시된 값을 **SEQUENCE**의 *index*번째 원소 앞에 삽입한다.

구문

```
void addToSequence(String attrName, int index, Object value)
```

예제

```
//create class foo(c list of int )
//insert into foo values({3})

String sql = "select foo from foo";

Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1);          // get OID
    oid.addToSequence("c",1, new Integer(22)); // c: {3}-> {22,3}
}
```

addToSet

설명

CUBRIDOID에 해당하는 인스턴스의 *attrName*에 명시된 이름에 해당하는 **SET** 또는 **MULTISET** 속성에 *value*에 명시된 값을 삽입한다.

구문

```
void addToSet(String attrName, Object value)
```

예제

```
//create class foo(a set of int, b multiset of int )
//insert into foo values({1},{2})
String sql = "select foo from foo";

Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);
while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1);          // get OID
    oid.addToSet("a",new Integer(11));    // a : {1} -> {1,11}
    oid.addToSet("b",new Integer(13));    // b : {2} -> {2, 13}
}
```

getNewInstance

설명

OID 문자열을 **CUBRIDOID** 객체로 변환시키고 **CUBRIDOID** 객체를 반환한다.

구문

```
static CUBRIDOID getNewInstance(CUBRIDConnection con, String oidStr)
```

리턴 값

- **CUBRIDOID** 객체

예제

```
String sql = "select foo from foo";

CUBRIDConnection con = (CUBRIDConnection)
    DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID realoid = rs.getOID(1); // get OID (CUBRIDOID)
    // CUBRIDOID -> OID string
    String stringoid = realoid.getOidString();
    // OID string -> CUBRIDOID
    realoid = CUBRIDOID.getInstance(con, stringoid);
}
```

getOidString

설명

CUBRIDOID 객체를 OID 문자열로 변환시키고 문자열을 반환한다.

구문

```
String getOidString()
```

리턴 값

- 문자열

예제

```
String sql = "select foo from foo";

CUBRIDConnection con = (CUBRIDConnection)
    DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID realoid = rs.getOID(1); // get OID
    // CUBRIDOID -> OID string
    String stringoid = realoid.getOidString();
}
```

```
// OID string -> CUBRIDOID
realoid = CUBRIDOID.getNewInstance(con,stringoid);
}
```

getTableName

설명

CUBRIDOID에 해당하는 인스턴스의 테이블 이름을 반환한다.

구문

```
String getTableName()
```

리턴 값

- **CUBRIDOID**에 해당하는 인스턴스의 테이블 이름

예제

```
String sql = "select foo from foo";

CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1);

    String tablename = oid.getTableName();
    System.out.println(tablename );
}
```

getValues

설명

요청된 속성의 값을 담고 있는 **ResultSet**을 반환한다.

구문

```
ResultSet getValues (String[] attrNames)
```

리턴 값

- **ResultSet**

예제

```
// create class foo ( a string, b int )
// insert into foo values('CUBRID', 2001)

String sql = "select foo from foo";
String[] attr = { "a", "b" }; // class's column name list
CUBRIDResultSet rs= (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1);
    ResultSet rsoid = oid.getValues(attr);
}
```

isInstance

설명

CUBRIDOID에 해당하는 인스턴스가 존재하면 참(true)을, 그렇지 않을 경우 거짓(false)을 반환한다.

구문

```
Boolean isInstance()
```

리턴 값

- 참(true) : **CUBRIDOID**에 해당하는 인스턴스가 존재함.
- 거짓(false) : **CUBRIDOID**에 해당하는 인스턴스가 존재하지 않음.

예제

```
String sql = "select foo from foo";

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1);
    System.out.print("isInstance : " + oid.isInstance()); // true
    oid.remove(); // remove the object in the oid
    System.out.print("After remove, isInstance : "
        + oid.isInstance()); // false
}
```

loadGLO

설명

주어진 *stream* 안에 **CUBRIDOID**에 해당하는 GLO를 로드한다.

구문

```
void loadGLO(OutputStream stream)
```

예제

```
CUBRIDConnection con = (CUBRIDConnection)
    DriverManager.getConnection(url,user,passwd);

// filename : testin1, classname : filetest
FileInputStream inputfile = new FileInputStream("testin1");
CUBRIDOID oid = con.getNewGLO("filetest", inputfile);
inputfile.close();

// filename : testout1, classname : filetest
FileOutputStream outputfile = new FileOutputStream("testout1");
System.out.println("loadGLO..");
oid.loadGLO(outputfile);
outputfile.close();

con.commit();
// 이 트랜잭션을 커밋해야 한다.
// CUBRIDOID를 사용시에는 자동 커밋 기능을 사용할 수 없다.
```


putIntoSequence

설명

CUBRIDOID에 해당하는 인스턴스의 *attrName*에 명시된 이름에 해당하는 **SEQUENCE** 속성의 *index*번째 원소값을 *value*에 명시된 값으로 변경한다.

구문

```
void putIntoSequence(String attrName, int index, Object value)
```

예제

```
//create class foo(c list of int )
//insert into foo values({1})

String sql = "select foo from foo";

Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1);           // get OID
    oid.putIntoSequence("c",1, new Integer(10)); // c:{1}->{10}
}
```

remove

설명

CUBRIDOID에 해당하는 인스턴스를 삭제한다.

구문

```
void remove()
```

예제

```
String sql = "select foo from foo" ;

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1);
    System.out.print("isInstance : " + oid.isInstance()); // true
    oid.remove(); // remove the object in the oid
    System.out.print(" After remove .isInstance : " +
        oid.isInstance()); // false
}
```

removeFromSequence

설명

CUBRIDOID에 해당하는 인스턴스의 *attrName*에 명시된 이름에 해당하는 **SEQUENCE** 속성의 *index*번째 원소를 삭제한다.

구문

```
void removeFromSequence(String attrName, int index)
```

예제

```
//create class foo(c list of int )
//insert into foo values(1,3)

String sql = "select foo from foo";

Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1);        // get OID
    oid.removeFromSequence("c",1);        // c: {1,3} -> {3}
}
```

removeFromSet

설명

CUBRIDOID에 해당하는 인스턴스의 *attrName*에 명시된 이름에 해당하는 **SET** 속성 중 *value*와 일치하는 값을 삭제한다. 일치하는 값이 하나 이상일 경우 처음에 일치하는 값을 제거한다.

구문

```
void removeFromSet(String attrName, Object value)
```

예제

```
//create class foo(a set of int, b multiset of int )
//insert into foo values({1,11},{2,13})

String sql = "select foo from foo";
Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();
CUBRIDResultSet rs= (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1);        // get OID
    oid.removeFromSet("a",new Integer(11)); // a: {1,11} ->{1}
    oid.removeFromSet("a",new Integer(13)); // b: {2,13} ->{2}
}
```

saveGLO

설명

CUBRIDOID에 해당하는 GLO에 주어진 *stream*을 저장한다. *length*를 지정할 경우, 주어진 *stream*을 *length* 바이트 만큼 저장한다.

구문

```
void saveGLO(InputStream stream)
void saveGLO(InputStream stream, int length)
```

예제

```
CUBRIDConnection con = (CUBRIDConnection) DriverManager.getConnection(url,user,passwd);

// filename : testin1, classname : filetest
FileInputStream inputfile = new FileInputStream("testin1");
CUBRIDOID oid = con.getNewGLO("filetest", inputfile);
inputfile.close();

// filename : testin2, classname : filetest
FileInputStream inputfile = new FileInputStream("testin2");
System.out.println("saveGLO..");
oid.saveGLO( inputfile );
inputfile.close();

con.commit();
// 이 트랜잭션을 커밋해야 한다.
// CUBRIDOID를 사용시에는 자동 커밋 기능을 사용할 수 없다.
CUBRIDConnection con = (CUBRIDConnection) DriverManager.getConnection(url,user,passwd);

// filename : testin1, classname : filetest
FileInputStream inputfile = new FileInputStream("testin1");
CUBRIDOID oid = con.getNewGLO("filetest", inputfile);
inputfile.close();

// filename : testin2, classname : filetest
FileInputStream inputfile = new FileInputStream("testin2");
System.out.println("saveGLO..");
oid.saveGLO( inputfile, 100 ); // save just 100 byte
inputfile.close();

con.commit();
// You must commit this transaction
// NOT AUTO COMMIT on CUBRIDOID
```

setReadLock

설명

CUBRIDOID에 해당하는 인스턴스에 읽기 잠금(read-lock)을 설정한다.

구문

```
void setReadLock ()
```

예제

```
String sql = "select foo from foo";

CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1);
    oid.setReadLock();
}
```

setValues

설명

attrNames 배열에 명시된 속성의 값을 *values* 배열에 명시된 값으로 변경한다.

구문

```
void setValues(String[] attrNames, Object[] values)
```

예제

```
// create class foo ( a string, b int )

String sql = "select foo from foo";
String[] attr = { "a", "b" }; // a list of attribute names
String[] values = {"CUBRID", new Integer(2001)};

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1);
    oid.setValues(attr, values );
}
```

setWriteLock

설명

CUBRIDOID에 해당하는 인스턴스에 쓰기 잠금(write-lock)을 설정한다.

구문

```
void setWriteLock()
```

예제

```
String sql = "select foo from foo";

CUBRIDResultSet rs = (CUBRIDResultSet) stmt.executeQuery(sql);

while (rs.next ()) {
    CUBRIDOID oid = rs.getOID(1);
    oid.setWriteLock();
}
```

CUBRIDPreparedStatement

개요

표준 **PreparedStatement** 클래스가 확장된 **CUBRIDPreparedStatement** 클래스는 다음과 같은 추가된 메소드를 가지고 있다.

반환 타입	메소드 이름
CUBRIDOID	executeInsert()
void	setCollection(int index, Object[] array)
void	setOID(int index, CUBRIDOID oid)

executeInsert

설명

PreparedStatement 객체 내의 **INSERT**문을 실행하고 삽입된 인스턴스에 해당하는 **CUBRIDOID**를 반환한다.

구문

```
CUBRIDOID executeInsert()
```

리턴 값

- 삽입된 인스턴스에 해당하는 **CUBRIDOID**

예제

```
String sql = "insert into testtable(a) values(?)";  
CUBRIDPreparedStatement pstmt = (CUBRIDPreparedStatement)  
    con.prepareStatement(sql);  
pstmt.setString(1, "CUBRID");  
CUBRIDOID oid = pstmt.executeInsert();
```

setCollection

설명

*array*에 해당하는 컬렉션을 예비문(prepared statement) *index*번째 파라미터에 설정한다.

(CUBRID에는 Set, Multiset, Sequence 세가지 타입의 컬렉션이 있다.)

구문

```
void setCollection(int index, Object[] array)
```

예제

```
String[] str = { "abc", "def"};  
psmt.setCollection(1, str);
```

setOID

설명

*oid*에 명시된 **CUBRIDOID**를 예비문 *index*번째 파라미터에 설정한다.

구문

```
void setOID(int index, CUBRIDOID oid)
```

CUBRIDResultSet

개요

표준 **ResultSet** 클래스가 확장된 **CUBRIDResultSet** 클래스는 다음과 같은 추가된 메소드를 가지고 있다.

반환 타입	메소드 이름
Object	getCollection(int attrIndex)
Object	getCollection(String attrName)
CUBRIDOID	getOid()
CUBRIDOID	getOid(int attrIndex)
CUBRIDOID	getOid(String attrName)

getCollection

설명

*attrIndex*에 명시된 인덱스나 *attrName*에 명시된 열 이름에 해당하는 열의 값을 반환한다. 반환된 객체는 `String[]`와 같은 배열로 변환할 수 있다.

구문

```
Object getCollection(int attrIndex)
Object getCollection(String attrName)
```

리턴 값

- attrIndex*에 명시된 인덱스, 또는 *attrName*에 명시된 열 이름에 해당하는 열의 값

getOID

설명

*attrIndex*에 명시된 인덱스나 *attrName*에 명시된 열 이름에 해당하는 열의 값을 **CUBRIDOID**로 변환하여 반환한다.

*attrIndex*나 *attrName*을 명시하지 않았을 경우에는 **ResultSet**의 현재 행의 **CUBRIDOID**를 반환한다. 이는 **ResultSet**이 **TYPE_SCROLL_SENSITIVE**이거나 **CONCUR_UPDATABLE**일 경우에만 유효하다.

구문

```
CUBRIDOID getOID(int attrIndex)
CUBRIDOID getOID(String attrName)
CUBRIDOID getOID()
```

리턴 값

- CUBRIDOID

CUBRIDResultSetMetaData

개요

표준 **ResultSetMetaData** 클래스가 확장된 **CUBRIDResultSetMetaData** 클래스는 다음과 같은 추가된 메소드를 가지고 있다.

반환 타입	메소드 이름
int	getElementType(int columnIndex)
String	getElementTypeName(int columnIndex)

getElementType

설명

컬렉션 원소의 타입을 **java.sql.Types**에 정의된 *int*로 반환한다. 만약 *columnIndex*번째 열의 데이터 타입이 컬렉션(**SET**, **MULTISET**, **SEQUENCE**)이 아니면 **SQLException**이 발생한다.

구문

```
int getElementType(int columnIndex)
```

리턴 값

- 컬렉션 원소의 타입 (int)

getElementTypeName

설명

컬렉션의 원소 타입 이름을 반환한다. 만약 *columnIndex*번째 열의 데이터 타입이 컬렉션(**SET**, **MULTISET**, **SEQUENCE**)이 아니면 **SQLException**이 발생한다.

구문

```
String getElementTypeName(int columnIndex)
```

리턴 값

- 컬렉션의 원소 타입 이름

예제

```
// 이 예제에서 다음의 스키마가 사용된다.  
//
```

```
// create class foo(
//   a  set(int),
//   b  multiset(int),
//   c  sequence(int)
// );

String sql = "select * from foo";
Connection con = DriverManager.getConnection(url,user,passwd);

Statement stmt = con.createStatement();

CUBRIDResultSet rs = (CUBRIDResultSet)stmt.executeQuery(sql);
CUBRIDResultSetMetaData rsmd = (CUBRIDResultSetMetaData)
    rs.getMetaData();

int numberOfColumn = rsmd.getColumnCount();
for (int i=1; i <= numberOfColumn; i++ ) {
    System.out.println(rsmd.getElementType(i) );
    System.out.println(rsmd.getElementTypeName(i) );
}
```

CUBRIDStatement

개요

표준 **Statement** 클래스가 확장된 **CUBRIDStatement** 클래스는 다음과 같은 추가된 메소드를 가지고 있다.

반환 타입	메소드 이름
CUBRIDOID	executeInsert(String insertStmt)

executeInsert

설명

*insertStmt*에 명시된 SQL문으로 추가된 새로운 행에 해당하는 **CUBRIDOID**를 반환한다.

구문

```
CUBRIDOID executeInsert(String insertStmt)
```

리턴 값

- 추가된 행의 **CUBRIDOID**

예제

```
String sql = "insert into testable(a) values (1)"

CUBRIDStatement stmt = (CUBRIDStatement) con.createStatement();
CUBRIDOID oid = stmt.executeInsert(sql);
```


ODBC API

ODBC 프로그래밍

CUBRID ODBC 드라이버

CUBRID ODBC 드라이버는 ODBC 코어 부분과 Level 1과 Level 2 일부 API를 지원한다. CUBRID ODBC 드라이버는 ODBC Spec 3.x를 기반으로 한다. 그러므로 ODBC Spec 2.x을 이용하여 작성한 프로그램에 대해서 하위 호환성을 완벽하게 보장하지는 않는다. 그리고 CUBRID는 객체관계형 DBMS이므로 관계형 DBMS에 맞게 구현된 일부 ODBC 특징은 지원되지 않는다.

OID와 컬렉션 사용

ODBC는 관계형 DBMS에 맞게 설계되었다. 그러므로 CUBRID ODBC는 CUBRID의 OID와 컬렉션과 같은 일부 객체지향 특징을 지원하지 않는다. CUBRID는 관계형 데이터 모델과 객체지향 데이터 모델을 통합한 객체관계형 DBMS이기 때문이다.

OID 사용

CUBRID ODBC 드라이버는 OID를 string (char(32))으로 간주하므로 OID를 포함하는 **INSERT**, **UPDATE**, **DELETE**를 아래와 같이 사용할 수 있다. OID string을 사용할 때는 반드시 작은 따옴표 (")처리가 필요하다. 다음 예제의 member 속성의 도메인은 객체(OID)와 같다.

```
insert into foo(member) values('@12|34|56')
delete from foo where member = '@12|34|56'
update foo set age = age + 1 where member = '@12|34|56'
```

컬렉션 사용

컬렉션 타입 : **SET**, **MULTISET**, **SEQUENCE** 세가지가 있다. CUBRID ODBC 드라이버는 컬렉션을 string(longvarchar)으로 간주한다. 컬렉션은 **SELECT** 질의에서 "{value_1, value_2, ... value_n}"와 같이 중괄호 안의 쉼표를 이용해 각 요소를 구분하여 획득한다.

CUBRID와 ODBC의 데이터 타입 Mapping

CUBRID 데이터 타입	ODBC 데이터 타입
Char	SQL_CHAR
Varchar	SQL_VARCHAR
String	SQL_LONGVARCHAR
Nchar	SQL_CHAR
Varnchar	SQL_VARCHAR
Bit	SQL_BINARY

varying bit	SQL_VARBINARY
Numeric	SQL_NUMERIC
Int	SQL_INTEGER
Short	SQL_SMALLINT
Float	SQL_FLOAT
Double	SQL_DOUBLE
Bigint	SQL_BIGINT
Date	SQL_TYPE_DATE
Time	SQL_TYPE_TIME
Timestamp	SQL_TYPE_TIMESTAMP
Datetime	SQL_TYPE_TIMESTAMP
Monetary	SQL_DOUBLE
Oid	SQL_CHAR(32)
set, multiset, sequence	SQL_VARCHAR(MAX_STRING_LENGTH)

하위 호환성

다음은 ODBC 3.x 드라이버의 하위 호환성 지원에 따라 ODBC 2.x 함수 중 ODBC 3.x에서 지원하지 않는 함수 대신에 어떤 함수가 지원되는지를 나타낸다.

ODBC 2.x 함수

ODBC 2.x 함수	ODBC 3.x 함수
SQLAllocConnect	SQLAllocHandle
SQLAllocEnv	SQLAllocHandle
SQLAllocStmt	SQLAllocHandle
SQLBindParam	SQLBindParameter
SQLColAttributes	SQLColAttribute
SQLError	SQLGetDiagRec
SQLFreeConnect	SQLFreeHandle
SQLFreeEnv	SQLFreeHandle
SQLFreeStmt with SQL_DROP	SQLFreeHandle
SQLGetConnectOption	SQLGetConnectAttr

SQLGetStmtOption	SQLGetStmtAttr
SQLParamOptions	SQLSetStmtAttr
SQLSetConnectOption	SQLSetConnectAttr
SQLSetParam	SQLBindParameter
SQLSetScrollOption	SQLSetStmtAttr
SQLSetStmtOption	SQLSetStmtAttr
SQLTransact	SQLEndTran

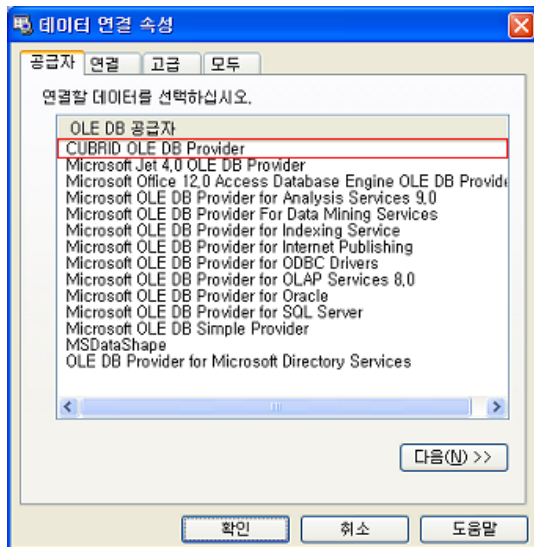
OLE DB API

OLE DB 프로그래밍

데이터 연결 속성 대화 상자 사용

[데이터 연결 속성] 대화 상자에서는 현재 사용하고 있는 Windows 운영 체제에 있는 각종 OLE DB 공급자를 확인하고 연결 속성을 설정할 수 있다.

Windows용 CUBRID OLE DB Provider를 올바르게 설치하였다면 다음 그림과 같이 [데이터 연결 속성] 대화 상자의 공급자 목록에 'CUBRID OLE DB Provider'가 나타난다.



'CUBRID OLE DB Provider'를 선택한 뒤 [다음] 버튼을 클릭하면 다음과 같이 [연결] 탭이 나타난다. [연결] 탭에서 원하는 연결 속성을 설정한다.

데이터 연결 속성

공급자 연결 고급 모두

이 데이터로 연결하려면 다음을 지정하십시오:

1. 데이터 원본 및/또는 데이터의 위치를 입력하십시오:

데이터 원본(D): itrackdb

위치(L): 192.168.1.11

2. 서버에 로그인할 때 사용할 정보를 입력하십시오:

☐ Windows NT의 통합 보안 사용(W)

☒ 특정 사용자 이름 및 암호 사용(U):

사용자 이름(N): itrack

암호(P): *****

☐ 빈 암호(B) ☐ 암호 저장 허용(S)

3. 사용할 초기 카탈로그를 입력하십시오(I):

연결 테스트(I)

확인 취소 도움말

- **데이터 원본** : CUBRID 데이터베이스의 이름을 입력한다.
- **위치** : CUBRID 브로커가 구동 중인 서버의 IP 주소 또는 호스트 이름을 입력한다.
- **사용자 이름** : 데이터베이스 서버에 로그인할 때 사용할 사용자 이름을 입력한다.
- **암호** : 데이터베이스 서버에 로그인할 때 사용할 암호를 입력한다.

연결 속성을 모두 설정한 후 [모두] 탭을 누른다.

데이터 연결 속성

공급자 연결 고급 모두

다음은 이 데이터 형식에 대한 초기화 속성입니다. 값을 편집하려면 [값 편집]을 선택하십시오.

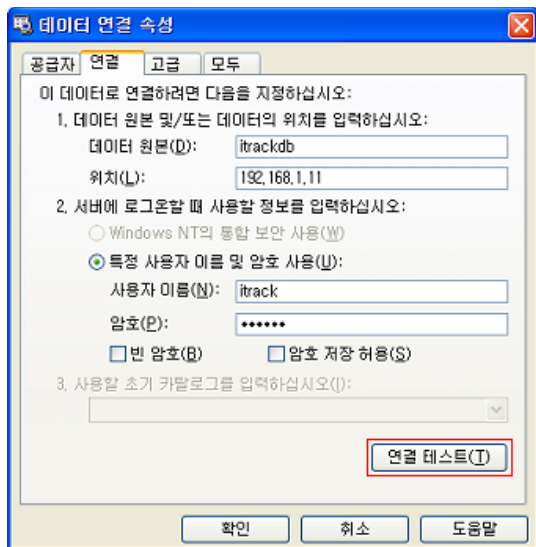
이름	값
Cache Authentication	False
Data Source	itrackdb
Encrypt Password	False
Fetch Size	100
Location	192.168.1.11
Mask Password	False
Password	*****
Persist Encrypted	False
Persist Security Info	False
Port	31000
User ID	itrackdb

값 편집(E)...

확인 취소 도움말

[모두] 탭을 클릭하면 현재 설정한 각각의 항목 값을 확인할 수 있다. 설정 값을 편집하려면 값을 편집하려는 항목을 더블 클릭한다. [속성 값 편집] 대화 상자가 나타나면 원하는 값을 입력한 뒤 [확인]을 누른다. 위 그림은 [Port] 번호는 '31000', [Fetch Size]는 '100'으로 설정한 예이다.

모든 설정을 마친 뒤, [연결] 탭에서 [연결 테스트] 버튼을 클릭하면 연결이 제대로 되는지 테스트해 볼 수 있다.



연결 문자열(connection string) 구성

ADO(ActiveX Data Object) 혹은 ADO.net을 이용하여 CUBRID OLE DB Provider 프로그래밍을 할 때 연결 문자열(connection string)은 다음과 같이 작성한다.

항목	예	설명
Provider	CUBRIDProvider	공급자 이름
Data Source	demodb	데이터베이스 이름
Location	192.168.1.11	CUBRID 브로커 서버 IP 주소 또는 호스트 이름
User ID	PUBLIC	사용자 ID
Password	xxx	비밀번호
Port	30000	브로커 Port 번호
Fetch Size	100	Fetch 크기

위의 예를 이용한 연결 문자열은 다음과 같다.

```
"Provider = CUBRIDProvider;Data Source = demodb;Location = 192.168.1.11;User ID = PUBLIC;Password =xxx;Port = 30000;Fetch Size = 100"
```

.NET 환경에서의 멀티 스레드 프로그래밍

Microsoft의 .NET 환경에서 CUBRID OLE DB Provider를 이용하여 프로그래밍할 때 추가로 고려해야 할 사항은 다음과 같다.

관리 환경에서 ADO.NET을 통한 멀티 스레드 프로그래밍을 할 때에는, CUBRID OLE DB Provider가 오직 STA(Single Threaded Apartment) 속성만을 지원하므로, Thread 객체의 ApartmentState 속성 값을 ApartmentState.STA 값으로 변경해야 한다.

만약 아무런 설정을 하지 않는다면 Thread 객체의 이 속성 디폴트 값으로 Unknown 값이 반환되기 때문에 멀티 스레드 프로그래밍 시 비정상적으로 동작할 수 있다.

주의 OLE DB의 모든 객체는 COM 객체이다. 현재 CUBRID OLE DB Provider는 COM threading model 중 apartment threading model만을 지원하고 free threading model은 지원하지 않는다. 이는 .NET 환경에만 해당하는 사항은 아니고 모든 multi-threaded 환경에 해당하는 내용이다.

PHP API

PHP 프로그래밍

일반 특징

연결/트랜잭션

- 데이터베이스 연결 : 데이터베이스 응용에서 첫 단계는 [cubrid_connect\(\)](#) 함수를 사용하는 것으로 데이터베이스 연결을 제공한다. [cubrid_connect\(\)](#) 함수가 성공적으로 수행되면, 데이터베이스를 사용할 수 있는 모든 함수를 사용할 수 있다. 응용을 완전히 끝내기 전에 [cubrid_disconnect\(\)](#) 함수를 호출하는 것은 매우 중요하다. [cubrid_disconnect\(\)](#) 함수는 현재 발생된 트랜잭션을 끝나치고 [cubrid_connect\(\)](#) 함수에 의해 생성된 연결 핸들과 모든 요청 핸들을 종료한다.
- 데이터베이스 트랜잭션 : [cubrid_commit\(\)](#) 이나 [cubrid_rollback\(\)](#) 함수를 사용하여 트랜잭션을 커밋 또는 롤백 할 수 있다. [cubrid_disconnect\(\)](#) 함수는 트랜잭션을 끝나치고 언커밋 트랜잭션을 롤백한다.

질의 처리

질의 실행

다음은 질의 실행을 위한 기본 단계이다.

- 연결 핸들 생성
- SQL 질의 요청에 대한 요청 핸들 생성
- 결과 가져오기
- 요청 핸들 종료

```
$con = cubrid_connect("192.168.1.12", 12345, "demodb");
if($con) {
    $req = cubrid_execute($con, "select * from dept");
    if($req) {
        while ($row = cubrid_fetch($req)) {
            echo $row["name"];
            echo $row["position"];
        }
        cubrid_close_request($req);
    }
    cubrid_disconnect($con);
}
```

질의 결과의 열 타입과 이름

[cubrid_column_types\(\)](#) 함수를 사용하여 열 타입이 들어있는 배열을 얻을 수 있고, [cubrid_column_names\(\)](#) 함수를 사용하여 열의 이름이 들어있는 배열을 얻을 수 있다.

```
$req = cubrid_execute($con, "select * from person");
if($req) {
    $coltypes = cubrid_column_types($req);
    $colnames = cubrid_column_names($req);

    while (list($key, $coltype) = each ($coltypes))
        echo $coltype;
}
```



```

while (list($key, $colname) = each ($colnames))
    echo $colname

    cubrid_close_request($req);
}

```

커서 조정

질의 결과의 위치를 설정할 수 있다. [cubrid_move_cursor\(\)](#) 함수를 사용하여 커서를 세 가지 포인트(질의 결과의 처음, 현재 커서 위치, 질의 결과의 끝) 중 한 포인트로부터 일정한 위치로 이동할 수 있다.

```

$req = cubrid_execute($con, "select * from person");
if($req) {
    cubrid_move_cursor ($req, 10, CUBRID_CURSOR_CURRENT);
    while ($row = cubrid_fetch($req, CUBRID_ASSOC)) {
        echo $row["id"];
        echo $row["name"];
    }
}

```

결과 배열 타입

[cubrid_fetch\(\)](#) 함수의 결과에는 세가지 종류의 배열 타입 중 하나가 사용된다. [cubrid_fetch\(\)](#) 함수가 호출될 때 배열의 타입을 결정할 수 있다. 그 중 하나인 연관배열은 문자열 색인을 사용한다. 두 번째로 수치배열은 숫자 순서 색인을 사용한다. 마지막 배열은 연관배열과 수치배열을 둘 다 포함한다.

- 수치배열

```

while (list($id, $name) = cubrid_fetch($req, CUBRID_NUM)) {
    echo $id;
    echo $name;
}

```

- 연관배열

```

while ($row = cubrid_fetch($req, CUBRID_ASSOC)) {
    echo $row["id"];
    echo $row["name"];
}

```

카탈로그 연산

클래스, 가상 클래스, 속성, 메소드, 트리거, 제약 조건 등 데이터베이스의 스키마 정보는 [cubrid_schema\(\)](#) 함수를 호출하여 얻을 수 있다. [cubrid_schema\(\)](#) 함수의 리턴 값은 2차원 배열이다.

```

$attrs=cubrid_schema($con,CUBRID_SCH_ATTRIBUTE,"person");
if ($attrs != -1) {
    while (list ($key, $attr) = each($attrs)) {
        echo $row["NAME"];
        echo $row["DOMAIN"];
    }
}

```

에러 처리

에러가 발생하면 대부분의 PHP 인터페이스 함수는 에러 메시지를 출력하고 false나 -1을 반환한다.

[cubrid_error_msg\(\)](#), [cubrid_error_code\(\)](#) 그리고 [cubrid_error_code_facility\(\)](#) 함수를 사용하면 각각 에러 메시지, 에러 코드, 에러 기능 코드를 확인할 수 있다.

[cubrid_error_code_facility\(\)](#) 함수의 결과 값은 **CUBRID_FACILITY_DBMS** (DBMS 에러), **CUBRID_FACILITY_CAS** (CAS 서버 에러), **CUBRID_FACILITY_CCI** (CCI 에러), **CUBRID_FACILITY_CLIENT** (PHP 모듈 에러) 중 하나이다.

CUBRID 특징

OID 사용

`cubrid_execute()` 함수에서 `CUBRID_INCLUDE_OID` 옵션을 업데이트할 수 있는 질의를 함께

사용하면 `cubrid_current_oid()` 함수를 통해 업데이트된 현재 행의 OID 값을 가져올 수 있다.

```
$req = cubrid_execute($con, "select * from person",
CUBRID_INCLUDE_OID);
if ($req) {
while ($row = cubrid_fetch ($req)) {
echo cubrid_current_oid ($req);
echo $row["&quot;id&quot;"];
echo $row["&quot;name&quot;"];
}
cubrid_close_request ($req);
}
```

OID를 사용하여 인스턴스의 모든 속성, 지정한 속성 또는 한 속성의 값을 얻을 수 있다.

만약 `cubrid_get()` 함수에 속성을 명시하지 않으면 모든 속성의 값을 반환한다(a). 만약 배열 데이터 타입으로 속성을 명시하면 지정한 속성 값이 들어있는 배열은 연관배열로 반환된다(b). 만약 문자열 타입으로 한 속성을 명시하면 속성의 값이 반환된다(c).

```
$attrarray = cubrid_get ($con, $oid); // (a)
$attrarray = cubrid_get ($con, $oid, array("id", "name")); // (b)
$attrarray = cubrid_get ($con, $oid, "id"); // (c)
```

OID를 사용하여 인스턴스의 속성 값을 갱신할 수도 있다. 하나의 속성의 값을 갱신하려면 속성 이름을 문자열 타입으로 명시하고 값을 명시한다(a). 다중 속성의 값을 설정하려면 속성 명과 값을 연관배열로 명시해야 한다(b).

```
$cubrid_put ($con, $oid, "id", 1); // (a)
$cubrid_put ($con, $oid, array("id">1, "name">"Tomas")); // (b)
```

컬렉션 사용

- 컬렉션 데이터 타입은 PHP 배열 데이터 타입을 통해 사용할 수 있고 배열 데이터 타입을 지원하는 PHP 함수를 사용할 수 있다. 다음은 `cubrid_fetch()` 함수를 사용하여 질의 결과를 가져오는 예제이다.

```
$row = cubrid_fetch ($req);
$col = $row["customer"];
while (list ($key, $cust) = each ($col)) {
echo $cust;
}
```

- 컬렉션 속성의 값도 얻을 수 있다. 다음은 `cubrid_col_get()` 함수를 사용하여 컬렉션 속성 값을 가져오는 예제이다.

```
$tels = cubrid_col_get ($con, $oid, "tels");
while (list ($key, $tel) = each ($tels)) {
echo $tel."<br>";
}
```

- `cubrid_set_add()` 함수와 `cubrid_set_drop()` 함수를 사용하면 컬렉션 타입의 값을 직접적으로 갱신할 수 있다.

```
$tels = cubrid_col_get ($con, $oid, "tels");
while (list ($key, $tel) = each ($tels)) {
$res = cubrid_set_drop ($con, $oid, "tel", $tel);
}
cubrid_commit ($con);
```

GLO 사용

GLO 데이터 타입은 데이터베이스에 파일을 저장하기 위해 사용된다.

`cubrid_new_glo()` 함수를 사용하면 GLO 인스턴스를 생성할 수 있고 생성된 인스턴스에 파일이 저장된다.

`cubrid_save_to_glo()` 함수를 사용하면 명시한 GLO에 파일을 저장할 수 있다. `cubrid_load_from_glo()` 함수를 사용하면 GLO 인스턴스로부터 데이터를 읽어올 수 있다. `cubrid_send_glo()` 함수를 사용하면 GLO 데이터를 PHP 표준 출력으로 전송할 수 있다.

```
$oid = $cubrid_new_glo ($con, "glo", "input.txt");
$cubrid_load_from_glo ($con, $oid, "output.txt");
Header ("Content-type: image/jpeg");
cubrid_send_glo ($con, $oid);
```

cubrid_affected_rows

설명

cubrid_affected_rows 함수는 SQL 문(**INSERT**, **DELETE**, **UPDATE**)에 의해 적용 받은 행의 개수를 얻는데 사용된다.

구문

```
int cubrid_affected_rows (int req_handle)
```

- *req_handle*: 요청 핸들

리턴 값

- 성공: SQL 문에 의해 적용 받은 행의 개수를 반환
- 실패: -1

예제

```
$req = cubrid_execute ($con, "delete from person where name like 'j%' ");
if ($req) {
$row count = cubrid_affected_rows ($req);
echo $row_count;
cubrid_close_request ($req);
}
```

관련 항목

- `cubrid_execute`

cubrid_bind

설명

cubrid_bind 함수는 파라미터가 표시된 [cubrid_prepare](#)의 변수에 값을 대입할 목적으로 사용된다. 대입 값의 타입은 다음 표와 같다.

구분	바인드 타입	대응 SQL 타입
지원	STRING	CHAR, VARCHAR
	NCHAR	NCHAR, NVARCHAR
	BIT	BIT, VARBIT
	NUMERIC 또는 NUMBER	SHORT, INT, NUMERIC
	FLOAT	FLOAT
	DOUBLE	DOUBLE
	TIME	TIME
	DATE	DATE
	TIMESTAMP	TIMESTAMP
	OBJECT	OBJECT
	NULL	NULL
미지원	SET	SET
	MULTISET	MULTISET
	SEQUENCE	SEQUENCE

구문

```
int cubrid_bind (int req_handle, int bind_index, string bind_value [, string bind_value_type])
```

- *req_handle* : [cubrid_prepare](#)의 결과로 받은 요청 핸들
- *bind_index* : 바인딩할 위치
- *bind_value* : 바인딩할 실제 값
- *bind_value_type* : 바인딩할 값의 타입. 기본적으로 생략 가능. 생략 시 자동으로 해당 타입에 맞게 변환되어 처리된다. 단, **NCHAR**, **BIT**는 반드시 타입을 인자로 넘겨주어야 한다.

리턴 값

- 성공 : true
- 실패 : false

예제

```
$con = cubrid_connect ("dbsvr.cubrid.com", 12345, "demodb");
if ($con) {
    $sql = "insert into tbl values ( ?,?,?)";
    $req = cubrid_prepare( $con, $sql, CUBRID_INCLUDE_OID );
    $i = 0;
    while ( $i < 2 ) {
        $res = cubrid_bind( $req, 1, "1", "NUMBER");
        $res = cubrid_bind( $req, 2, "2");
        $res = cubrid_bind( $req, 3, "04:22:34 PM 08/07/2007");
        $res = cubrid_execute( $req );
        $i = $i + 1;
    }
}
```

관련 항목

- [cubrid_execute](#)
- [cubrid_prepare](#)

cubrid_close_request

설명

cubrid_close_request 함수는 *req_handle* 인수에 주어진 요청 핸들을 닫고, 핸들에 관련된 메모리 영역을 해제한다.

구문

```
int cubrid_close_request (int req_handle)
```

- *req_handle* : 요청 핸들

리턴 값

- 성공 : true
- 실패 : false

예제

```
$con = cubrid_connect ("dbsvr.cubrid.com", 12345, "demodb");
if ($con) {
    echo "connected successfully";
    $req = cubrid_execute ( $con, "select * from members",
                           CUBRID_INCLUDE_OID | CUBRID_ASYNC);
    if ($req) {
        while ( list ($id, $name) = cubrid_fetch ($req) ){
            echo $id;
            echo $name;
        }
        cubrid_close_request($req);
    }
    cubrid_disconnect($con);
}
```

관련 항목

- [cubrid_execute](#)

cubrid_col_get

설명

cubrid_col_get 함수는 주어진 컬렉션 타입(set, multiset, sequence) 속성의 원소 내용을 배열형식으로 얻는데 사용된다.

구문

```
int cubrid_col_get(int conn_handle, string oid, string attr_name)
```

- *conn_handle*: 연결 핸들
- *oid*: 원하는 인스턴스의 OID
- *attr_name*: 인스턴스로부터 읽기를 원하는 속성 이름

리턴 값

- 성공: 원하는 원소들이 들어있는 배열(0: 기본 수치 배열)
- 실패: false. 속성이 빈 컬렉션이나 **NULL**인 경우와 오류를 구별하기 위해서 오류가 발생하면 경고 메시지를 출력하고, **cubrid_error_code**를 통해 오류를 확인할 수 있다.

예제

```
$elem_array = cubrid_col_get ($con, $oid, "tel");  
while (list ($key, $val) = each ($elem_array)) {  
    echo "tel: $val\n";  
}
```

cubrid_col_size

설명

cubrid_col_size 함수는 컬렉션 타입(set, multiset, sequence) 속성의 원소 개수를 얻는데 사용된다.

구문

```
int cubrid_col_size(int conn_handle, string oid, string attr_name)
```

- *conn_handle*: 연결 핸들
- *oid*: 원하는 인스턴스의 OID
- *attr_name*: 인스턴스의 원하는 속성 이름

리턴 값

- 성공: 원소 개수
- 실패: -1

예제

```
$elem_count = cubrid_col_size ($con, $oid, "tel");  
echo "$oid (tel) has $elem_count elements\n";
```

cubrid_column_names

설명

cubrid_column_names 함수는 *req_handle*을 이용해 질의 결과에 대한 열 이름들을 얻는데 사용된다.

구문

```
mixed cubrid_column_names (int req_handle)
```

- *req_handle*: 요청 핸들

리턴 값

- 성공: 열 이름이 들어있는 배열
- 실패: false

예제

```
$req = cubrid_execute ($con, "select * from person");
if ($req) {
    $coltypes = cubrid_column_types ($req);
    $colnames = cubrid_column_names ($req);
    while (list ($key, $coltype) = each ($coltypes))
        echo $coltype;
    while (list ($key, $colname) = each ($colnames))
        echo $colname;
    cubrid_close_request ($req);
}
```

관련 항목

- [cubrid_execute](#)
- [cubrid_prepare](#)
- [cubrid_column_types](#)

cubrid_column_types

설명

cubrid_column_types 함수는 *req_handle*을 이용해 질의 결과에 대한 열 타입들을 얻는데 사용된다.

구문

```
mixed cubrid_column_types (int req_handle)
```

- *req_handle*: 요청 핸들

리턴 값

- 성공: 열의 타입이 들어있는 배열
- 실패: false

예제

```
$req = cubrid_execute ($con, "select * from person");
if ($req) {
    $coltypes = cubrid_column_types ($req);
    $colnames = cubrid_column_names ($req);
    while (list ($key, $coltype) = each ($coltypes))
        echo $coltype;
    while (list ($key, $colname) = each ($colnames))
        echo $colname;
    cubrid_close_request ($req);
}
```

관련 항목

- [cubrid_execute](#)
- [cubrid_prepare](#)
- [cubrid_column_names](#)

cubrid_commit

설명

cubrid_commit 함수는 *conn_handle*이 가리키는 접속에서 현재 진행중인 트랜잭션을 커밋한다.

cubrid_commit 함수가 호출된 이후에는 서버와의 연결이 끊어지지만 연결 핸들은 유효하다.

구문

```
int cubrid_commit (int conn_handle)
```

- *conn_handle*: 연결 핸들

리턴 값

- 성공 : true
- 실패 : false

예제

```
$req = cubrid_execute ($oid, "insert into person values
(2, 'John')");
if ($req) {
    cubrid_close_request ($req);
    if ($failed) {
        cubrid_rollback ($con);
    } else {
        cubrid_commit ($con);
    }
}
```

관련 항목

- [cubrid_rollback](#)

cubrid_connect

설명

cubrid_connect 함수는 주어진 서버의 주소, 포트 번호, 데이터베이스 이름, 사용자 이름, 비밀번호를 이용해서 서버와의 연결 환경을 설정한다. 만약 사용자 이름과 비밀번호를 설정하지 않으면 기본값으로 **PUBLIC**이 설정된다.

구문

```
int cubrid_connect (string host, int port, string dbname[, string userid[, string passwd]])
```

- *host*: 브로커 서버의 IP 주소 및 호스트 이름
- *port*: 브로커 서버의 포트 번호
- *dbname*: 데이터베이스 이름
- *userid*: 데이터베이스 사용자 이름
- *passwd*: 데이터베이스 사용자 비밀번호

리턴 값

- 성공: 연결 핸들
- 실패: false

예제

```
$con = cubrid connect ("210.211.133.100", 12345, "demodb");

if ($con) {
    echo "connected successfully";
    $req =cubrid execute($con, "insert into person values
    (1,'James')");

    if ($req) {
        cubrid close request ($req);
        cubrid commit ($con);
    } else {
        cubrid rollback ($con);
    }
    cubrid disconnect ($con);
}
```

관련 항목

- **cubrid_disconnect**

cubrid_connect_with_url

설명

url 인자로 전달된 접속 정보를 이용하여 데이터베이스로 연결을 시도한다. PHP에서 HA 기능을 사용하는 경우, 이 함수의 *url* 인자에 active 서버의 연결 정보 및 장애 발생 시 failover할 standby 서버의 연결 정보를 명시해야 한다. 서버 연결에 성공하면 연결 핸들 ID를 반환하고, 실패하면 에러 코드를 반환한다.

메모 [OH2]: cci_connect_with_url의 설명과 98%동일하므로 영문을 참고하면 됨.

구문

```
int cubrid_connect_with_url (char *url [, char *db user, char *db password ])  
  
<url> ::=  
cci:cubrid:<host>:<db name>:<db user>:<db password>:[?<properties>]  
  
<properties> ::= <property> [<property>]  
<property> ::= althosts=<alternative hosts> [rctime=<time>]  
<alternative hosts> ::= <standby broker1 host>:<port> [, <standby broker2 host>:<port>]  
  
<host> := HOSTNAME | IP ADDR  
<time> := SECOND
```

- url: (IN) 서버 연결 정보 문자 스트링
- host: 마스터 데이터베이스의 호스트 이름 또는 IP 주소
- db_name: 데이터베이스 이름
- db_user: 데이터베이스 사용자 이름
- db_password: 데이터베이스 사용자 비밀번호
- **althosts**=standby broker1 host, standby broker2 host, ...: active 서버에 연결할 수 없는 경우, 그 다음으로 연결을 시도(failover)할 standby 서버의 브로커 정보를 나타낸다. failover할 브로커를 여러 개 지정할 수 있고, **althosts**에 나열한 순서대로 연결을 시도한다.
- **rctime**: 장애가 발생했던 active 브로커에 연결을 시도하는 주기이다. 장애 발생 후 **althosts**에 명시한 브로커로 접속하여(failover) 트랜잭션을 종료한 후, **rctime**만큼 시간이 경과할 때마다 마스터 데이터베이스의 active 브로커에 연결을 시도한다. 디폴트 값은 600초이다.
- db_user: (IN) 데이터베이스 사용자 이름
- db_passwd: (IN) 데이터베이스 사용자 비밀번호

리턴 값

- 성공: 연결 핸들 ID (int)
- 실패: 에러 코드

cubrid_current_oid

설명

cubrid_current_oid 함수는 질의 결과에서 현재 커서 위치의 OID를 얻어온다. **cubrid_current_oid**를 사용하기 위해서는 갱신할 수 있는 질의를 실행해야 하며, 질의 수행시 **CUBRID_INCLUDE_OID** 옵션이 들어있어야 한다.

구문

```
mixed cubrid_current_oid (int req_handle)
```

- req_handle: 요청 핸들

리턴 값

- 성공: 현재 커서 위치의 OID

- 실패 : false

예제

```
$req =cubrid_execute($con,"select * from person where id =1",
                    CUBRID INCLUDE OID);
if ($req) {
    cubrid_fetch ($req);
    $oid =cubrid_current_oid ($req);
    cubrid_close_request ($req);
    echo "OID is $oid";
}
```

관련 항목

- [cubrid_execute](#)

cubrid_data_seek

설명

cubrid_data_seek 함수는 결과 레코드를 가리키는 내부 포인터가 *row_number*에 해당하는 레코드를 가리키도록 포인터 위치를 이동시킨다. 이 함수를 호출한 후, [cubrid_fetch_assoc](#) 함수와 같은 결과 셋을 fetch하는 함수를 호출하면 해당 레코드가 반환된다.

구문

```
bool cubrid_data_seek (int $result, int $row_number)
```

- *result*: 대상 결과 핸들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.
- *row_number*: 결과 포인터가 새로 가리킬 row의 번호를 지정한다.

리턴 값

- 성공 : true
- 실패 : false

예제

```
<?php
$link = cubrid_connect('localhost', 8080, 'cubrid_database', 'cubrid_user',
'cubrid_password');
if (!$link) {
    die('Could not connect: ' . cubrid_error_msg ());
}
$query = 'SELECT last name, first name FROM friends';
$result = cubrid_execute($link, $query);
if (!$result) {
    die('Query failed: ' . cubrid_error_msg ());
}
/* fetch rows in reverse order */
for ($i = cubrid_num_rows ($result) - 1; $i >= 0; $i--) {
    if (!cubrid_data_seek($result, $i)) {
        echo "Cannot seek to row $i: " . cubrid_error_msg () . "\n";
        continue;
    }

    if (!$row = cubrid_fetch_assoc($result)) {
        continue;
    }
}
```

```

    }

    echo $row['last name'] . ' ' . $row['first name'] . "<br />\n";
}
?>

```

cubrid_disconnect

설명

cubrid_disconnect 함수는 현재 진행중인 트랜잭션을 중지하고, 서버와의 연결을 해제 및 연결 핸들을 닫는다. 만약 현재까지 아직 닫히지 않은 요청 핸들이 있으면 모두 닫히게 된다.

구문

```
int cubrid_disconnect (int conn_handle)
```

- *conn_handle*: 연결 핸들

리턴 값

- 성공: true
- 실패: false

예제

```

$con = cubrid_connect ("210.211.133.100", 12345, "demodb");
if ($con) {
    echo "connected successfully";
    $req = cubrid_execute ( $con, "insert into person values(1,'James')");
    if ($req) {
        cubrid_close_request ($req);
        cubrid_commit ($con);
    } else {
        cubrid_rollback ($con);
    }
    cubrid_disconnect ($con);
}

```

관련 항목

- **cubrid_connect**

cubrid_drop

설명

cubrid_drop 함수는 OID를 이용하여 데이터베이스의 원하는 인스턴스를 삭제한다.

구문

```
int cubrid_drop (int conn_handle, string oid)
```

- *conn_handle*: 연결 핸들
- *oid*: 삭제하기 원하는 인스턴스의 OID

리턴 값

- 성공 : true
- 실패 : false

예제

```
$deloid = cubrid_get ($con, $oid, "order");
$res = cubrid_drop ($con, $deloid);
if ($res) {
    cubrid_commit ($con);
} else {
    cubrid_rollback ($con);
}
```

관련 항목

- `cubrid_is_instance`

cubrid_error_code

설명

`cubrid_error_code` 함수는 API 실행 중 발생한 오류에 대한 오류 코드를 얻어온다. 일반적으로 API가 false를 반환했을 때 오류코드를 얻을 수 있다.

구문

```
int cubrid_error_code ()
```

리턴 값

- 에러 코드

예제

```
$req = cubrid_execute ($con, "select id, name from person");
if ($req) {
    while (list ($id, $name) = cubrid_fetch($req))
        echo $id, $name;
} else {
    echo "Error Code: ", cubrid_error_code ();
    echo "Error Facility: ", cubrid_error_code_facility ();
    echo "Error Message: ", cubrid_error_msg ();
}
```

관련 항목

- `cubrid_error_code_facility`
- `cubrid_error_msg`

cubrid_error_code_facility

설명

cubrid_error_code_facility 함수는 API 실행 중 발생한 오류의 오류 코드로부터 기능 코드(오류가 발생한 수준)를 얻는데 사용된다. 일반적으로 API가 거짓을 반환할 때 오류 코드를 얻을 수 있다.

구문

```
int cubrid_error_code_facility ()
```

리턴 값

- 발생한 에러 코드의 기능 코드 :
CUBRID_FACILITY_DBMS, CUBRID_FACILITY_CAS,
CUBRID_FACILITY_CCI, CUBRID_FACILITY_CLIENT

예제

```
$req = cubrid_execute ($con, "select id, name from person");  
if ($req) {  
    while (list ($id, $name) = cubrid_fetch($req))  
        echo $id, $name;  
} else {  
    echo "Error Code: ", cubrid_error_code ();  
    echo "Error Facility: ", cubrid_error_code_facility ();  
    echo "Error Message: ", cubrid_error_msg ();  
}
```

관련 항목

- **cubrid_error_code**
- **cubrid_error_msg**

cubrid_error_msg

설명

cubrid_error_msg 함수는 API 실행 중 발생한 오류에 대한 오류 메시지를 얻는 데 사용된다.

일반적으로 API가 거짓을 반환했을 때 오류 메시지를 얻을 수 있다.

구문

```
string cubrid_error_msg ()
```

리턴 값

- 발생한 오류 메시지

예제

```
$req = cubrid_execute ($con, "select id, name from person");  
if ($req) {  
    while (list ($id, $name) = cubrid_fetch($req))  
        echo $id, $name;  
}
```

```

} else {
    echo "Error Code: ", cubrid_error_code ();
    echo "Error Facility: ", cubrid_error_code_facility ();
    echo "Error Message: ", cubrid_error_msg ();
}

```

관련 항목

- `cubrid_error_code`
- `cubrid_error_code_facility`

cubrid_execute

설명

cubrid_execute 함수는 주어진 SQL 문을 실행하는 데 사용되며, *conn_handle*과 SQL을 이용하여 질의를 실행하고 생성된 요청 핸들을 반환한다. 파라미터 바인딩이 필요 없는 경우에 간단하게 질의를 수행할 때 적절한 방법이다.

[cubrid_prepare](#)와 [cubrid_bind](#)를 통해 prepared statement를 수행할 때도 **cubrid_execute**를 이용하며, 이때 요구되는 인자는 *request_handle*과 *option*이다.

질의 수행 후 행의 OID를 가져올 것 인지와 비동기 모드 질의를 수행할지를 결정할 때 *option* 인수를 사용할 수 있다. **CUBRID_INCLUDE_OID**와 **CUBRID_ASYNC**를 비트 OR 연산자(`|`)를 사용해서 지정할 수 있다. 지정하지 않으면 두 가지 모두 선택되지 않는다.

cubrid_prepare를 수행하기 위한 *request_handle*이 첫번째 인자일 경우, **CUBRID_ASYNC**만 옵션으로 가능하다.

구문

```
int cubrid_execute (int conn_handle, string SQL [, int option])
```

- *conn_handle* : 연결 핸들
- *SQL* : 실행할 SQL
- *option* : 질의 실행 옵션 CUBRID_INCLUDE_OID, CUBRID_ASYNC

```
int cubrid_execute (int request_handle[, int option])
```

- *request_handle* : **cubrid_prepare** 핸들
- *option* : 질의 실행 옵션 CUBRID_ASYNC

리턴 값

- 성공 : 요청 핸들
- 실패 : false

예제

```

$con = cubrid_connect ("dbsvr.cubrid.com", 33000, "demodb");
if ($con) {
    echo "connected successfully";
}

```

```

    $req = cubrid execute ( $con, "select * from members",
                          CUBRID INCLUDE OID |
                          CUBRID ASYNC);    if ($req) {
while ( list ($id, $name) = cubrid fetch ($req) ){    echo $id;
    echo $name;
}
    cubrid close request ($req); }    cubrid disconnect ($con);
} $con = cubrid_connect ("dbsvr.cubrid.com", 33000, "demodb");

if ($con) {    echo "connected successfully";

$sql = "insert into tbl values ( ?,?,?)"; $req = cubrid_prepare
    ( $con, $sql, CUBRID INCLUDE OID );

    $i = 0;
    while ( $i < 2 ) {
        $res = cubrid bind( $req, 1, "1", "NUMBER");
        $res = cubrid bind( $req, 2, "2");
        $res = cubrid bind( $req, 3, "04:22:34 PM 08/07/2007");
        $res = cubrid execute( $req );
        $i = $i + 1;
    }
}

```

관련 항목

- **cubrid_close_request**
- **cubrid_commit**
- **cubrid_rollback**
- **cubrid_prepare**
- **cubrid_bind**

cubrid_fetch

설명

cubrid_fetch 함수는 질의 결과로부터 한 행을 얻는 데 사용된다. 결과를 가져온 후 커서는 자동으로 다음 행으로 이동한다.

구문

```
mixed cubrid_fetch (int req_handle [, int type])
```

- *req_handle* : 요청 핸들
- *type* : 가져올 결과 배열 타입
CUBRID_NUM, CUBRID_ASSOC,
CUBRID_BOTH, CUBRID_OBJECT

리턴 값

- 성공 : 결과 배열이나 객체.

type 인수에서 결정한다. *type* 인수를 생략했을 때는 CUBRID_BOTH로 간주한다. 질의의 결과를 객체 데이터 타입으로 받을 경우는 결과 열 이름이 PHP에서 허용하는 식별자의 규칙에 맞아야 사용할 수 있다. 예를 들어 "count(*)"와 같은 열 이름은 객체 타입으로 받아서 사용할 수 없다.

*type*에 따른 결과의 형태는 다음과 같다.

- CUBRID_NUM : 수치 배열 (0-기본)
- CUBRID_ASSOC : 연관 배열
- CUBRID_BOTH : 수치 그리고 연관 배열 (기본값)
- CUBRID_OBJECT : 질의 결과의 열 이름과 같은 이름의 속성을 가지는 객체
- 실패 : false

예제

```
$req = cubrid_execute ( $con, "select * from members",
                        CUBRID_INCLUDE_OID | CUBRID_ASYNC);
if ($req) {
    while ( list ($id, $name) = cubrid_fetch ($req) ) {
        echo $id;
        echo $name;
    }
    cubrid_close_request ($req);
}
$req = cubrid_execute ($con, "select * from teams");
if ($req) {
    while ($row = cubrid_fetch ($req, CUBRID_OBJECT)) {
        echo $row->id;
        echo $row->name;
    }
}
```

관련 항목

- [cubrid_execute](#)

cubrid_fetch_assoc

설명

cubrid_fetch_assoc 함수는 fetch한 레코드에 해당하는 연관 배열을 반환하고, 내부 데이터 포인터를 앞으로 이동시킨다. 다음 레코드가 존재하지 않는 경우에는 **FALSE**를 반환한다.

구문

```
array cubrid_fetch_assoc ( int $result )
```

- result : 대상 결과 핸들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.

예제

```
<?php
$query = 'SELECT id,email FROM people WHERE id = '42'';
$result = cubrid_execute($link, $query);
if (!$result) {
    die('Query failed: ' . cubrid_error_msg ());
}
$row= cubrid_fetch_assoc($result);
$lengths = cubrid_fetch_lengths($result);

print_r($row);
print_r($lengths);
?>
```

```

Output:
Array
(
    [id] => 42
    [email] => user@example.com
)
Array
(
    [0] => 2
    [1] => 16
)

```

cubrid_fetch_field

설명

cubrid_fetch_field 함수는 필드(또는 컬럼) 정보를 포함하는 객체를 반환한다. 이 함수는 질의 결과 내의 컬럼들에 관한 정보를 얻기 위해 사용할 수 있으며, 반환하는 객체들의 특성은 아래를 참고한다.

- **name** : 해당 컬럼 이름
- **table** : 해당 컬럼이 속한 테이블의 이름
- **def** : 해당 컬럼에 정의된 디폴트 값
- **max_length** : 해당 컬럼의 최대 길이
- **not_null** : 해당 컬럼에 **NOT NULL** 속성이 정의된 경우, 1
- **unique_key** : 해당 컬럼에 **UNIQUE KEY**가 정의된 경우, 1
- **multiple_key** : 해당 컬럼이 non-unique key 컬럼인 경우, 1
- **numeric** : 해당 컬럼 타입이 numeric인 경우, 1
- **type** : 해당 컬럼의 타입

구문

```
object cubrid_fetch_field ( int $result [, int $field_offset= 0 ] )
```

- **result** : 대상 결과 행들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.
- **field_offset** : 필드 오프셋을 숫자로 지정하며, 0부터 시작한다. 만약, 필드 오프셋이 지정되지 않으면, 이 함수에 의해 아직 조회되지 않은 다음 필드가 조회된다.

예제

```

<?php
$link = cubrid connect('localhost', 8080, 'cubrid database', 'cubrid user',
'cubrid_password');
if (!$link) {
    die('Could not connect: ' . cubrid_error_msg ());
}
$query = 'SELECT last_name, first_name FROM friends';
$result = cubrid execute($link, $query);
if (!$result) {
    die('Query failed: ' . cubrid_error_msg ());
}
/* fetch rows in reverse order */
$i = 0;
while ($i < cubrid num fields($result)) {
    echo "Information for column $i:<br />\n";
    $meta = cubrid fetch field($result, $i);
    if (!$meta) {
        echo "No information available<br />\n";
    }
}

```

```

    }
    echo "<pre>
        max length:      $meta->max length
        multiple key:    $meta->multiple key
        name:            $meta->name
        not null:        $meta->not null
        numeric:         $meta->numeric
        table:           $meta->table
        type:            $meta->type
        default:         $meta->def
        unique_key:      $meta->unique_key
    </pre>";
    $i++;
}
?>

```

cubrid_fetch_lengths

설명

cubrid_fetch_lengths 함수는 fetch한 마지막 레코드의 각 필드 값 길이를 배열로 반환한다. 실패하면 **FALSE**를 반환한다.

구문

```
array cubrid_fetch_lengths ( int $result )
```

- result*: 대상 결과 핸들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.

예제

```

<?php
$query = 'SELECT id,email FROM people WHERE id = '42'';
$result = cubrid_execute($link, $query);
if (!$result) {
    die('Query failed: ' . cubrid_error_msg ());
}
$row= cubrid_fetch_assoc($result);
$lengths = cubrid_fetch_lengths($result);

print_r($row);
print_r($lengths);
?>

Output:
Array
(
    [id] => 42
    [email] => user@example.com
)
Array
(
    [0] => 2
    [1] => 16
)

```

cubrid_fetch_object

설명

cubrid_fetch_object 함수는 지정된 result 식별자와 연관된 결과로부터 하나의 레코드를 인출하여 속성을 가진 객체로 반환하고, 내부 데이터 포인터를 앞으로 이동시킨다. 더 이상의 행이 존재하지 않으면 **FALSE**를

반환한다. 이 함수의 실행을 통해 반환된 객체는 해당 레코드가 가진 필드의 이름을 그 속성으로 가지며, 해당 속성을 통해 각 필드 값을 참조할 수 있다.

구문

```
object cubrid_fetch_object ( int $result )
```

- *result*: 대상 결과 핸들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.

예제

```
<?php
$link = cubrid_connect('localhost', 8080, 'cubrid_database', 'cubrid_user',
'cubrid_password');
if (!$link) {
    die('Could not connect: ' . cubrid_error_msg ());
}
$result = cubrid_execute($link, "select * from mytable");
while ($row = cubrid_fetch_object($result)) {
    echo $row->user id;
    echo $row->fullname;
}
?>
```

cubrid_fetch_row

설명

cubrid_fetch_row 함수는 지정된 *result* 식별자와 연관된 결과로부터 하나의 레코드를 인출하여 배열로 반환하고, 내부 데이터 포인터를 앞으로 이동시킨다. 더 이상의 행이 존재하지 않으면 **FALSE**를 반환한다. 각 필드 값은 배열로 저장된다.

구문

```
array cubrid_fetch_row ( int $result )
```

- *result*: 대상 결과 핸들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.

예제

```
<?php
$link = cubrid_connect('localhost', 8080, 'cubrid_database', 'cubrid_user',
'cubrid_password');
if (!$link) {
    die('Could not connect: ' . cubrid_error_msg ());
}
$result = cubrid_execute($link, "SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
    echo 'Could not run query: ' . cubrid_error_msg ();
    exit;
}
$row = cubrid_fetch_row($result);

echo $row[0]; // 42
echo $row[1]; // the email value
?>
```

cubrid_field_flags

설명

cubrid_field_flags 함수는 지정된 필드의 플래그를 반환한다. 플래그는 공백 문자로 구분된 하나의 플래그당 하나의 단어로 구성되며, **explode()**를 사용하여 반환되는 값을 분리할 수 있다.

구문

```
string cubrid_field_flags ( int $result , int $field_offset )
```

- *result*: 대상 결과 핸들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.
- *field_offset*: 0으로 시작하는 필드 오프셋을 지정한다.

예제

```
<?php
$result = cubrid_execute($link, "SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
    echo 'Could not run query: ' . cubrid_error_msg();
    exit;
}
$flags = cubrid_field_flags($result, 0);

echo $flags;
print_r(explode(' ', $flags));
?>

Array
(
    [0] => not_null
    [1] => primary_key
    [2] => auto_increment
)
```

cubrid_field_len

설명

cubrid_fetch_len 함수는 지정된 필드의 길이를 반환하고, 실패하면 **FALSE**를 반환한다.

구문

```
string cubrid_field_len ( int $result , int $field_offset )
```

- *result*: 대상 결과 핸들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.
- *field_offset*: 0으로 시작하는 필드 오프셋을 지정한다. 오프셋 값이 존재하지 않으면 에러가 출력된다.

예제

```
<?php
$result = cubrid_execute($link, "SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
    echo 'Could not run query: ' . cubrid_error_msg();
    exit;
}
// Will get the length of the id field as specified in the database
// schema.
$length = cubrid_field_len($result, 0);
echo $length;
?>
```

cubrid_field_name

설명

cubrid_field_name 함수는 지정된 필드 이름을 반환하고, 실패하면 **FALSE**를 반환한다.

구문

```
string cubrid_field_name ( int $result , int $field_offset )
```

- *result*: 대상 결과 행들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.
- *field_offset*: 0으로 시작하는 필드 오프셋을 지정한다. 오프셋 값이 존재하지 않으면 에러가 출력된다.

예제

```
<?php
$result = cubrid_execute($link, "select * from users");
if (!$result) {
    echo 'Could not run query: ' . cubrid_error_msg();
    exit;
}

echo mysql_field_name($result, 0) . "\n";
echo mysql_field_name($result, 2);
?>

Output:
user_id
password
```

cubrid_field_seek

설명

cubrid_field_seek 함수는 [cubrid_fetch_field](#) 함수에서 사용할 필드의 오프셋 값을 설정한다. 오프셋을 포함하지 않는 [cubrid_fetch_field](#) 함수가 호출되면, 이 함수에 지정된 필드 오프셋 값이 반환된다.

구문

```
string cubrid_field_seek ( int $result , int $field_offset )
```

- *result*: 대상 결과 행들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.
- *field_offset*: 0으로 시작하는 필드 오프셋을 지정한다. 오프셋 값이 존재하지 않으면 에러가 출력된다.

리턴 값

- 성공 : true
- 실패 : false

예제

```
<?php
$result = cubrid_execute($link, "select * from users");
if (!$result) {
    echo 'Could not run query: ' . cubrid_error_msg();
    exit;
}
```

```
$row = cubrid_fetch_array($result);
echo $row['ID'] . ' ' . $row['Name'];
cubrid_field_seek($result,2);
echo $row['ID'] . ' ' . $row['Name'];
?>
```

cubrid_field_table

설명

cubrid_field_table 함수는 지정된 필드가 속한 테이블 이름을 반환한다.

구문

```
string cubrid_field_table ( int $result , int $field_offset )
```

- *result*: 대상 결과 핸들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.
- *field_offset*: 0으로 시작하는 필드 오프셋을 지정한다. 오프셋 값이 존재하지 않으면 에러가 출력된다.

예제

```
<?php
$query = "SELECT account.*, country.* FROM account, country WHERE country.name =
'Portugal' AND account.country id = country.id";

$result = cubrid_execute($link, $query);
if (!$result) {
echo 'Could not run query: ' . cubrid_error_msg();
exit;
}

// Lists the table name and then the field name
for ($i = 0; $i < cubrid_num_fields($result); ++$i) {
$table = cubrid_field_table($result, $i);
$field = cubrid_field_name($result, $i);

echo "$table: $field\n";
}

?>
```

cubrid_field_type

설명

cubrid_field_type 함수는 지정한 필드의 타입을 반환하며, 리턴되는 필드 타입은 CUBRID가 지원하는 데이터 타입 중 하나이다(예: "int", "float" 또는 "string").

구문

```
string cubrid_field_type ( int $result , int $field_offset )
```

- *result*: 대상 결과 핸들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.
- *field_offset*: 0으로 시작하는 필드 오프셋을 지정한다. 오프셋 값이 존재하지 않으면 에러가 출력된다.

예제

```
<?php
$query = "SELECT * FROM func";
```

```

$result = cubrid_execute($link, $query);
if (!$result) {
    echo 'Could not run query: ' . cubrid_error_msg();
    exit;
}

$fields = cubrid_num_fields($result);
$rows = cubrid_num_rows($result);
$table = cubrid_field_table($result, 0);
echo "Your '" . $table . "' table has " . $fields . " fields and " . $rows . "
record(s)\n";
echo "The table has the following fields:\n";
for ($i=0; $i < $fields; $i++) {
    $type = cubrid_field_type($result, $i);
    $name = cubrid_field_name($result, $i);
    $len = cubrid_field_len($result, $i);
    $flags = cubrid_field_flags($result, $i);
    echo $type . " " . $name . " " . $len . " " . $flags . "\n";
}
?>

Output:
Your 'func' table has 4 fields and 1 record(s)
The table has the following fields:
string name 64 not_null primary_key binary
int ret 1 not null
string dl 128 not_null
string type 9 not_null enum

```

cubrid_free_result

설명

cubrid_free_result 함수는 질의 결과를 메모리로부터 해제시킨다.

구문

```
bool cubrid_free_result ( int $result )
```

- result*: 대상 결과 행들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.

예제

```

<?php
$result = cubrid_execute("SELECT id,email FROM people WHERE id = '42'");
/* Use the result, assuming we're done with it afterwards */
$row = cubrid_fetch_assoc($result);

/* Now we free up the result and continue on with our script */
cubrid_free_result($result);

echo $row['id'];
echo $row['email'];
?>

```

cubrid_get

설명

cubrid_get 함수는 OID를 이용해 인스턴스의 원하는 속성을 얻어온다. *attr* 인수에 문자열 타입을 사용하여 하나의 속성을 얻을 수도 있고, 배열 타입을 사용하여 여러 개의 속성을 한꺼번에 얻을 수도 있다.

구문

```
mixed cubrid_get (int conn_handle, string oid[, mixed attr])
```

- *conn_handle*: 연결 핸들
- *oid*: 값을 얻기 원하는 인스턴스의 OID
- *attr*: 값을 얻기 원하는 속성의 이름

리턴 값

attr 인수에 문자열 타입으로 설정했을 경우에 결과는 문자열로 반환되며, *attr* 인수에 배열 타입(0-기본 수치배열)으로 설정했을 경우에 결과는 연관배열로 반환된다. 만약 *attr* 인수가 생략되는 경우에는 인스턴스의 모든 속성을 연관 배열 형태로 받아온다.

- 성공: 요청한 속성의 내용
- 실패: false. 빈 문자열이나 **NULL**과 오류를 구별하기 위해서 오류가 발생하면, 경고 메시지를 출력하고, [cubrid_error_code](#)를 통해 오류를 확인할 수 있다.

예제

```
$attrarray = cubrid_get ($con, $oid);  
echo $attrarray["id"];  
echo $attrarray["name"];
```

관련 항목

- [cubrid_put](#)

cubrid_get_charset

설명

cubrid_get_charset 함수는 현재 CUBRID 연결에 설정된 문자 세트(charset)를 문자열로 반환한다.

구문

```
string cubrid_get_charset ( void )
```

예제

```
<?php  
printf("current charset: %s\n", cubrid_get_charset($req));  
?>
```

cubrid_get_class_name

설명

cubrid_get_class_name 함수는 OID로부터 클래스 이름을 얻는 데 사용된다.

구문

```
mixed cubrid_is_instance (int conn_handle, string oid)
```

- *conn_handle*: 연결 핸들
- *oid*: 존재 여부를 알기 원하는 인스턴스의 OID

리턴 값

- 성공: 클래스 이름
- 실패: false

예제

```
$target oid = cubrid get ($con, $oid, "customer");
$class name = cubrid get class name ($con, $target oid);
if ($class name) {
    echo "class name of $oid is $class name\n";
} else {
    echo "error\n";
}
```

관련 항목

- **cubrid_is_instance**
- **cubrid_drop**

cubrid_get_client_info

설명

cubrid_get_client_info 함수는 현재 cci 라이브러리의 버전을 문자열로 반환한다.

구문

```
string cubrid_get_client_info ( void )
```

예제

```
<?php
printf("CUBRID client info: %s\n", cubrid get client info());
?>
```

cubrid_get_db_parameter

설명

cubrid_get_db_parameter 함수는 데이터베이스에 설정된 파라미터를 배열로 반환한다.

구문

```
array cubrid_get_db_parameter ( $req )
```

예제

```
<?php
printf("CUBRID DB parameters: ");
printf_r(cubrid_get_db_parameter($req));
?>
```

cubrid_get_server_info

설명

cubrid_get_server_info 함수는 현재 접속한 CUBRID 서버의 버전을 문자열로 반환한다.

구문

```
string cubrid_get_server_info ( void )
```

예제

```
<?php
printf("CUBRID server info: %s\n", cubrid_get_server_info());
?>
```

cubrid_insert_id

설명

cubrid_insert_id 함수는 이전의 **INSERT** 질의에 의해 갱신된 **AUTO_INCREMENT** 컬럼을 위해 생성된 ID를 검색하며, 모든 **AUTO_INCREMENT** 컬럼들과 그 값들을 배열로 반환한다. 이전 질의로부터 **AUTO_INCREMENT** 값이 생성되지 않는 경우에는 0을 반환하고, CUBRID 접속에 실패한 경우 **FALSE**를 반환한다.

구문

```
array cubrid_insert_id ( string $class_name [, int $connection_handle] )
```

- *class_name*: **AUTO_INCREMENT** 값을 생성한 마지막 **INSERT** 문이 실행된 대상 테이블 이름을 지정한다.
- *connection_handle*: **cubrid_connect**를 호출하여 획득한 마지막 연결 핸들을 지정한다.

예제

```
<?php
$result = cubrid_execute($link, "INSERT INTO mytable (product) values ('kossu')");
if (!$result) {
    echo 'Could not run query: ' . cubrid_error_msg();
    exit;
}
print_r(cubrid_insert_id("mytable"));
?>
```

cubrid_is_instance

설명

cubrid_is_instance 함수는 OID가 가리키는 인스턴스가 데이터베이스에 존재하는지 검사하는 데 사용한다.

구문

```
int cubrid_is_instance (int conn_handle, string oid)
```

- *conn_handle*: 연결 핸들

- *oid*: 존재하는지 알기 원하는 인스턴스의 OID

리턴 값

- 1: 인스턴스 있음
- 0: 인스턴스 없음
- -1: 오류 발생

예제

```
$target oid = cubrid get ($con, $oid, "customer");
$res = cubrid is instance ($con, $target oid);
if ($res == 1) {
    echo "$oid is presents.\n";
} else if ($res == 0) {
    echo "$oid is not presents.\n";
} else {
    echo "error\n";
}
```

관련 항목

- **cubrid_drop**
- **cubrid_get_class_name**

cubrid_list_dbs

설명

cubrid_list_dbs 함수는 서버에 존재하는 모든 데이터베이스의 이름을 배열로 반환한다.

구문

```
array cubrid_list_dbs ( )
```

예제

```
<?php
printf("CUBRID Databases: ");
printf r(cubrid_list_dbs());
?>
```

cubrid_load_from_glo

설명

cubrid_load_from_glo 함수는 glo 인스턴스로부터 데이터를 읽어 원하는 파일에 저장한다.

구문

```
int cubrid_load_from_glo (int conn_handle, string oid, string file_name)
```

- *conn_handle*: 연결 핸들
- *oid*: 데이터를 읽어올 glo 인스턴스의 OID

- *file_name*: 데이터를 저장할 파일의 이름

리턴 값

- 성공 : true
- 실패 : false

예제

```
$req = cubrid execute ($con, "select image from person where id=1");
if ($req) {
    list ($oid) = cubrid fetch($req);
    cubrid_close_request($req);
    $res = cubrid load from glo ($con, $oid, "output.jpg");
    if ($res) {
        echo "image changed successfully";
    }
}
```

관련 항목

- **cubrid_new_glo**
- **cubrid_save_to_glo**
- **cubrid_send_glo**

cubrid_lock_read

설명

cubrid_lock_read 함수는 OID를 이용해 원하는 인스턴스에 읽기 잠금(read lock)을 설정한다.

구문

```
int cubrid_lock_read (int conn_handle, string oid)
```

- *conn_handle*: 연결 핸들
- *oid*: 잠금을 설정하기 원하는 인스턴스의 OID

리턴 값

- 성공 : true
- 실패 : false

예제

```
$lock_oid = cubrid get ($con, $oid, "next id");
$res = cubrid_lock_read ($con, $lock_oid);
```

관련 항목

- **cubrid_lock_write**

cubrid_lock_write

설명

cubrid_lock_write 함수는 OID를 이용해 원하는 인스턴스에 쓰기 잠금(write lock)을 설정한다.

구문

```
int cubrid_lock_write (int conn_handle, string oid)
```

- *conn_handle*: 연결 핸들
- *oid*: 잠금을 설정하기 원하는 인스턴스의 OID

리턴 값

- 성공 : true
- 실패 : false

예제

```
$lock_oid = cubrid_get ($con, $oid, "next id");  
$res = cubrid_lock_write ($con, $lock_oid);
```

관련 항목

- **cubrid_lock_read**

cubrid_move_cursor

설명

cubrid_move_cursor 함수는 req_handle에 대한 현재의 커서 위치를 origin에서 정한 방향으로 offset 인수에 지정한 값만큼 이동한다. origin은 결과의 처음(CUBRID_CURSOR_FIRST), 결과의 현재 위치(CUBRID_CURSOR_CURRENT), 결과의 끝(CUBRID_CURSOR_LAST)를 사용할 수 있고, 만약 origin을 지정하지 않을 경우 기본값으로 CUBRID_CURSOR_CURRENT를 사용한다.

만약 커서의 이동 값이 결과의 범위를 벗어날 경우, 커서는 결과의 범위가 끝난 다음의 위치로 이동하게 된다. 예를 들어 크기가 10인 결과에서 커서를 20의 위치로 움직이게 되면 11번째 위치로 이동한 후, CUBRID_NO_MORE_DATA를 반환한다.

구문

```
int cubrid_move_cursor (int req_handle, int offset[, int origin])
```

- *req_handle*: 요청 핸들
- *offset*: 커서를 움직일 개수
- *origin*: 커서를 움직일 기준
CUBRID_CURSOR_FIRST,

```
CUBRID_CURSOR_CURRENT,  
CUBRID_CURSOR_LAST
```

리턴 값

- CUBRID_CURSOR_SUCCESS
- 실패 : CUBRID_NO_MORE_DATA

예제

```
cubrid move cursor ($req handle, 1, CUBRID_CURSOR_FIRST);  
// 처음의 위치로 커서를 이동한다.  
$row = cubrid_fetch ($req_handle);  
echo $row["id"], $row["name"];  
  
cubrid move cursor ($req handle, 1, CUBRID_CURSOR_LAST);  
// 커서의 위치를 끝으로 이동한다.  
$row = cubrid_fetch ($req_handle);  
echo $row["id"], $row["name"];
```

관련 항목

- [cubrid_execute](#)

cubrid_new_glo

설명

cubrid_new_glo 함수는 요청된 클래스(glo 클래스)에 glo 인스턴스를 생성한다. 생성하는 glo는 LO 타입이며, *file_name* 파일에 저장한다.

구문

```
mixed cubrid_new_glo (int conn_handle, string class_name, string file_name)
```

- *conn_handle* : 연결 핸들
- *class_name* : glo를 생성하려고 하는 클래스의 이름
- *file_name* : glo를 생성한 후 glo에 저장하고 싶은 파일의 이름

리턴 값

- 성공 : 생성된 인스턴스의 OID
- 실패 : false

예제

```
$oid = cubrid_new_glo ($con, "glo", "input.jpg");  
if ($oid){  
    $req = cubrid_execute ($con,  
                           "insert into person(image) values ($oid)");  
    if ($req) {  
        echo "image inserted successfully";  
        cubrid_close_request ($req);  
        cubrid_commit($con);  
    }  
}
```

```
}  
}
```

관련 항목

- [cubrid_save_to_glo](#)
- [cubrid_load_from_glo](#)
- [cubrid_send_glo](#)

cubrid_num_cols

설명

cubrid_num_cols 함수는 수행한 질의 결과의 열 수를 얻는다. 수행한 질의가 **SELECT** 문일 때만 가능하다.

구문

```
int cubrid_num_cols (int req_handle)
```

- *req_handle*: 요청 핸들

리턴 값

- 성공: 열의 수
- 오류 발생: -1

예제

```
$req = cubrid_execute ($con, "select * from member");  
if ($req) {  
    $rows_count = cubrid_num_rows ($req);  
    $cols_count = cubrid_num_cols ($req);  
    echo "result set rows count : $rows\n";  
    echo "result set columns count : $cols\n";  
    cubrid_close_request ($req);  
}
```

관련 항목

- [cubrid_execute](#)
- [cubrid_num_rows](#)

cubrid_num_fields

설명

cubrid_num_fields 함수는 결과 셋으로부터 필드의 개수를 반환하고, 실패하면 **FALSE**를 반환한다.

구문

```
int cubrid_num_fields ( int $result )
```

- *result*: 대상 결과 핸들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.

예제

```
<?php
$result = cubrid_execute($link, "SELECT id,email FROM people WHERE id = '42'");
if (!$result) {
    echo 'Could not run query: ' . cubrid_error_msg();
    exit;
}

/* returns 2 because id,email === two fields */
echo cubrid_num_fields($result);
?>
```

cubrid_num_rows

설명

cubrid_num_rows 함수는 실행한 질의 결과의 행 수를 얻는다. 실행한 질의가 **SELECT** 문일 때만 사용가능하며, **INSERT**, **UPDATE**, **DELETE** 질의의 결과에 대해서 알고 싶을 경우는 **cubrid_affected_rows**를 사용해야 한다. **cubrid_num_rows**는 동기 질의일 경우에만 사용가능하며, 비동기 질의일 때는 0을 반환한다.

구문

```
int cubrid_num_rows (int req_handle)
```

- *req_handle* : 요청 핸들

리턴 값

- 성공 : 행의 개수
- 0 : 비동기 질의
- -1 : 오류 발생

예제

```
$req = cubrid_execute ($con, "select * from member");
if ($req) {
    $rows_count = cubrid_num_rows ($req);
    $cols_count = cubrid_num_cols ($req);
    echo "result set rows count : $rows\n";
    echo "result set columns count : $cols\n";
    cubrid_close_request ($req);
}
```

관련 항목

- [cubrid_execute](#)
- **cubrid_num_cols**
- **cubrid_affected_rows**

cubrid_prepare

설명

cubrid_prepare 함수는 주어진 커넥션 핸들에 프리 컴파일된 SQL 문을 나타내는 API 이다. SQL 문은, 프리 컴파일되어 **cubrid_prepare**에 포함된다. 이것은 이 문장을 여러 차례 효율적으로 실행하는 목적으로 사용할 수 있고, Long Data를 처리하는데 효율적으로 사용할 수 있다. SQL 문은 단 하나만 올 수 있으며, 파라미터는 SQL 문의 적합한 위치에 MARK(?) 를 삽입할 수 있다. 파라미터는 **INSERT** 문장의 **VALUES** 절이나 SQL 문의 **WHERE** 절에서 값을 대입하고자 할 위치에 추가한다. MARK(?) 에 값을 대입하는 것은 [cubrid_bind](#)에 의해서만 할 수 있다.

구문

```
int cubrid_prepare (int conn_handle, string prepare_stmt [, int option])
```

- *conn_handle* : 연결 핸들
- *prepare_stmt* : prepare 질의문
- *option* : OID 반환 옵션 CUBRID_INCLUDE_OID

리턴 값

- 성공 : 요청 핸들(request handle)
- 실패 : false

예제

```
if ($con) {
$sql = "insert into tbl values ( ?,?,?)";
$req = cubrid_prepare( $con, $sql, CUBRID_INCLUDE_OID );
$i = 0;
while ( $i < 2 ) {
$res = cubrid_bind( $req, 1, "1", "NUMBER");
$res = cubrid_bind( $req, 2, "2");
$res = cubrid_bind( $req, 3, "04:22:34 PM 08/07/2007");
$res = cubrid_execute( $req );
$i = $i + 1;
}
}
```

관련 항목

- [cubrid_execute](#)
- [cubrid_bind](#)

cubrid_put

설명

cubrid_put 함수는 주어진 oid를 이용해 인스턴스의 원하는 속성들의 값을 바꾼다. 한 개의 속성을 바꾸기 위해서는 attr은 문자열 타입으로 하나의 속성을 입력하고, value에 수정할 내용을 정수나 부동소수점이나 문자열 타입으로 입력한다. 여러 개의 속성들을 동시에 바꾸기 위해서는 attr은 지정하지 않고 value만

연관배열 형태로 넘겨준다. 그러나, 컬렉션 타입의 속성에 대해서는 사용할 수 없다. 컬렉션 타입의 속성에 대해서 사용하려면 컬렉션 관련 API(cubrid_set_add, cubrid_set_drop, ...)를 사용해야 한다.

구문

```
int cubrid_put (int conn_handle, string oid[, string attr], mixed value)
```

- *conn_handle*: 연결 핸들
- *oid*: 값을 바꾸기 원하는 인스턴스의 OID
- *attr*: 값을 바꾸기 원하는 속성의 이름
- *value*: 바꾸기 원하는 속성의 값

리턴 값

- 성공 : true
- 실패 : false

예제

```
$attrarray = cubrid_get ($con, $oid);  
$attrarray["name"] = "New Name";  
cubrid_put ($con, $oid, $attrarray);  
cubrid_put ($con, $oid, "name", "New Name2");  
cubrid_put ($con, $oid, "hobbies", array("aa", "bb"));
```

관련 항목

- **cubrid_get**
- **cubrid_set_add**
- **cubrid_set_drop**
- **cubrid_seq_insert**
- **cubrid_seq_drop**
- **cubrid_seq_put**

cubrid_real_escape_string

설명

cubrid_real_escape_string 함수는 질의를 서버로 전송하기 전에 특수 문자열을 이스케이프(escape)하여 안전하게 질의를 수행할 수 있도록 한다. 특히, 이진 데이터를 입력하는 경우에는 반드시 이 함수를 사용하여야 한다.

cubrid_real_escape_string 함수는 다음의 문자에 백슬래시(\)를 붙인다.

- \x00, \n, \r, \, ' , "

구문

```
string cubrid_real_escape_string (string $unescaped_string [, resource $link_identifier ] )
```

- *unescaped_string* : 이스케이프할 문자열을 지정한다.
- *link_identifier* : connection을 지정한다. 이를 지정하지 않으면, **cubrid_connect** 함수에 의해 최근 성립된 연결을 사용한다.

리턴 값

- 성공: 이스케이프된 문자열
- 에러: FALSE

예제

```
<?php
$query = sprintf("SELECT * FROM users WHERE user='%s' AND password='%s'",
cubrid_real_escape_string($user),
cubrid_real_escape_string($password));
?>
```

cubrid_result

설명

cubrid_result 함수는 결과 셋으로부터 하나의 셀에 있는 데이터를 반환하고, 실패하면 **FALSE**를 반환한다.

만약 거대한 결과셋에 작업하는 경우라면, **cubrid_result** 함수 대신 레코드 전체를 인출하는 함수 중 하나를 사용하여야 한다. 이러한 함수들은 한 번의 함수 호출로 다중 셀의 데이터를 반환하므로

cubrid_result 함수보다 더욱 빠르다. 또한, 필드 인수로 숫자 오프셋을 지정하는 것이 필드명 또는 테이블명.필드명 인수를 지정하는 것보다 더 빠르다.

구문

```
string cubrid_result ( int $result , int $row [, mixed $field= 0 ] )
```

- *result* : 대상 결과 핸들을 지정한다. 이는 [cubrid_execute](#) 함수를 호출하여 얻을 수 있다.
- *row* : 검색할 행의 번호를 지정한다. 행 번호는 0부터 시작한다.
- *field* : 검색할 필드 오프셋을 지정하거나 필드 이름을 지정한다. 숫자 오프셋, 필드명, 또는 테이블명.필드명으로 지정할 수 있으며, 컬럼 별칭(alias)이 정의된 경우라면, 필드명 대신 별칭을 사용할 수 있다. 이 인수가 지정되지 않으면, 첫 번째 필드를 검색한다.

예제

```
<?php
$result = cubrid_execute($link, "SELECT name FROM work.employee");
if (!$result) {
    echo 'Could not run query: ' . cubrid_error_msg();
    exit;
}
if (!$result) {
    die('Could not query: ' . mysql_error());
}
echo mysql_result($result, 2); // outputs third employee's name
?>
```

cubrid_rollback

설명

cubrid_rollback 함수는 *conn_handle*이 가리키는 연결에서 진행 중인 트랜잭션을 롤백한다.

cubrid_rollback 함수가 호출된 이후에는 서버와의 연결이 끊어지지만, 연결 핸들은 유효하다.

구문

```
int cubrid_rollback (int conn_handle)
```

- *conn_handle*: 연결 핸들

리턴 값

- 성공 : true
- 실패 : false

예제

```
$req = cubrid_execute ($oid, "insert into person values (2,'John')");
if ($req) {
    cubrid_close_request ($req);
    if ($failed) {
        cubrid_rollback ($con);
    } else {
        cubrid_commit ($con);
    }
}
```

관련 항목

- **cubrid_commit**
- **cubrid_disconnect**

cubrid_save_to_glo

설명

cubrid_save_to_glo 함수는 원하는 파일을 glo 인스턴스에 저장한다.

구문

```
int cubrid_save_to_glo (int conn_handle, string oid, string file_name)
```

- *conn_handle*: 연결 핸들
- *oid*: 파일을 저장할 glo 인스턴스의 OID
- *file_name*: 저장하고 싶은 파일의 이름

리턴 값

- 성공 : true
- 실패 : false

예제

```
$req = cubrid execute ($con, "select image from person where id=1");
if ($req) {
    list ($oid) = cubrid fetch($req);
    cubrid_close_request($req);
    $res = cubrid save to glo ($con, $oid, "input.jpg");
    if ($res) {
        echo "image changed successfully";
    }
}
```

관련 항목

- `cubrid_new_glo`
- `cubrid_load_from_glo`
- `cubrid_send_glo`

cubrid_schema

설명

cubrid_schema 함수는 데이터베이스의 원하는 스키마 정보를 얻어온다. 만약 특정 클래스와 관련된 정보를 얻기 위해서는 *class_name*, 특정 속성과 관련된 정보(현재 CUBRID_SCH_ATTR_PRIVILEGE에서만 사용)를 얻기 위해서는 *attr_name*을 지정해 주어야 한다.

구문

```
mixed cubrid_schema (int conn_handle, int schema_type[, string class_name[, string attr_name]])
```

- *conn_handle* : 연결 핸들
- *schema_type* : 얻고 싶은 스키마 종류
- *class_name* : 스키마를 얻어오려고 하는 클래스
- *attr_name* : 스키마를 얻어오려고 하는 속성

리턴 값

- 성공 : 스키마 정보가 들어있는 배열
- 실패 : -1

cubrid_schema의 결과는 2차원 배열형(열(연관배열) * 행(수치배열))으로 반환되며, 스키마의 종류와 각 스키마의 종류에 따라 반환되는 결과 배열의 열 구성은 다음과 같다.

스키마	컬럼 번호	컬럼 이름	값
CUBRID_SCH_CLASS	1	NAME	0 : System class
			1 : vclass
			2 : class
	2	TYPE	

CUBRID_SCH_VCLASS	1	NAME	1 : vclass
	2	TYPE	
CUBRID_SCH_QUERY_SPEC	1	QUERY_SPEC	
CUBRID_SCH_ATTRIBUTE	1	ATTR_NAME	
	2	DOMAIN	
	3	SCALE	
	4	PRECISION	
	5	INDEXED	1 : indexed
	6	NON NULL	1 : non null
	7	SHARED	1 : shared
	8	UNIQUE	1 : unique
	9	DEFAULT	
	10	ATTR_ORDER	base : 1
	11	CLASS_NAME	
	12	SOURCE_CLASS	
CUBRID_SCH_CLASS_ATTRIBUTE	1	ATTR_NAME	
	2	DOMAIN	
	3	SCALE	
	4	PRECISION	
	5	INDEXED	1 : indexed
	6	NON NULL	1 : non null
	7	SHARED	1 : shared
	8	UNIQUE	1 : unique
	9	DEFAULT	
	10	ATTR_ORDER	base : 1
	11	CLASS_NAME	
	12	SOURCE_CLASS	
CUBRID_SCH_METHOD	1	NAME	
	2	RET_DOMAIN	
	3	ARG_DOMAIN	

CUBRID_SCH_METHOD_FILE	1	METHOD_FILE	
CUBRID_SCH_SUPERCLASS	1	CLASS_NAME	
	2	TYPE	
CUBRID_SCH_SUBCLASS	1	CLASS_NAME	
	2	TYPE	
CUBRID_SCH_CONSTRAINT	1	TYPE	0 : unique 1 : index
	2	NAME	
	3	ATTR_NAME	
CUBRID_SCH_TRIGGER	1	NAME	
	2	STATUS	
	3	EVENT	
	4	TARGET_CLASS	
	5	TARGET_ATTR	
	6	ACTION_TIME	
	7	ACTION	
	8	PRIORITY	
	9	CONDITION_TIME	
	10	CONDITION	
CUBRID_SCH_CLASS_PRIVILEGE	1	CLASS_NAME	
	2	PREVILEGE	
	3	GRANTABLE	
CUBRID_SCH_ATTR_PRIVILEGE	1	ATT_NAME	
	2	PREVILEGE	
	3	GRANTABLE	

예제

```
$attrs = cubrid_schema ($con, CUBRID_SCH_ATTRIBUTE, "person");
while (list($key, $value) = each($attrs)) {
    echo $value["NAME"];
    echo $value["DOMAIN"];
}
```


관련 항목

- `cubrid_new_glo`
- `cubrid_load_from_glo`
- `cubrid_send_glo`

cubrid_send_glo

설명

`cubrid_send_glo` 함수는 glo 인스턴스로부터 데이터를 꺼내 PHP 표준출력으로 보내준다.

구문

```
int cubrid_send_glo (int conn_handle, string oid)
```

- `conn_handle`: 연결 핸들
- `oid`: 데이터를 얻어올 glo 인스턴스의 OID

리턴 값

- 성공 : true
- 실패 : false

예제

```
$req = cubrid_execute ($con, "select image from person where id =1");
if ($req) {
    list ($oid) = cubrid_fetch($req);
    cubrid_close_request($req);
    Header ("Content-type: image/jpeg");
    cubrid_send_glo ($con, $oid);
}
```

관련 항목

- `cubrid_new_glo`
- `cubrid_save_to_glo`
- `cubrid_load_from_glo`

cubrid_seq_drop

설명

`cubrid_seq_drop` 함수는 주어진 **SEQUENCE** 타입의 속성에 원하는 원소를 데이터베이스에서 삭제한다.

구문

```
int cubrid_seq_drop(int conn_handle, string oid, string attr_name, int index)
```

- `conn_handle`: 연결 핸들
- `oid`: 원하는 인스턴스의 OID

- *attr_name*: 인스턴스의 원하는 속성 이름
- *index*: 삭제하고 싶은 원소의 인덱스. 디폴트 값은 1.

리턴 값

- 성공 : true
- 실패 : false

예제

```
$elems = cubrid_col_get ($con, $oid, "style");
$i = 1;
while (list ($key, $val) = each($elems)) {
  if ($val == "1") {
    echo $val;
    cubrid_seq_drop ($con, $oid, "style", $i);
  }
  $i++;
}
```

관련 항목

- **cubrid_seq_insert**
- **cubrid_seq_put**

cubrid_seq_insert

설명

cubrid_seq_insert 함수는 SEQUENCE 타입의 속성의 원하는 위치에 원하는 원소를 삽입한다.

구문

```
int cubrid_seq_insert (int conn_handle, string oid, string attr_name, int index, string seq_element)
```

- *conn_handle*: 연결 핸들
- *oid*: 원하는 인스턴스의 OID
- *attr_name*: 인스턴스의 원하는 속성 이름
- *index*: 새로운 원소를 삽입하기 원하는 위치(1-기본)
- *seq_string*: 삽입하고자 하는 원소의 내용

리턴 값

- 성공 : true
- 실패 : false

예제

```
cubrid_seq_insert ($con, $oid, "tel", 1, "02-3430-1200");
cubrid_seq_insert ($con, $oid, "tel", 1, "02-3430-1300");
```

관련 항목

- `cubrid_seq_drop`
- `cubrid_seq_put`

cubrid_seq_put

설명

`cubrid_seq_put` 함수는 주어진 SEQUENCE 타입의 속성의 원하는 원소의 내용을 변경한다.

구문

```
int cubrid_seq_put (int conn handle, string oid, string attr name, int index, string seq_element)
```

- `conn_handle`: 연결 핸들
- `oid`: 원하는 인스턴스의 OID
- `attr_name`: 인스턴스의 원하는 속성 이름
- `index`: 변경하기 원하는 원소의 인덱스(1-기본)
- `seq_element`: 변경하고자 하는 원소의 내용

리턴 값

- 성공 : true
- 실패 : false

예제

```
cubrid_seq_put ($con, $oid, "tel", 1, "02-3430-1200");  
cubrid_seq_put ($con, $oid, "tel", 2, "02-3430-1300");
```

관련 항목

- `cubrid_seq_insert`
- `cubrid_seq_drop`

cubrid_set_add

설명

`cubrid_set_add` 함수는 주어진 SET 타입(set, multiset) 속성에 원하는 원소를 하나 삽입한다.

구문

```
int cubrid_set_add (int conn handle, string oid, string attr_name, string set_element)
```

- `conn_handle`: 연결 핸들
- `oid`: 원하는 인스턴스의 OID
- `attr_name`: 인스턴스의 원하는 속성 이름

- *set_string*: 삽입하고자 하는 원소의 내용

리턴 값

- 성공 : true
- 실패 : false

예제

```
cubrid_set_add ($con, $oid, "friend", "James");
cubrid_set_add ($con, $oid, "friend", "Michael");
```

관련 항목

- **cubrid_set_drop**

cubrid_set_drop

설명

cubrid_set_drop 함수는 주어진 SET 타입(set, multiset) 속성에서 원하는 원소를 데이터베이스에서 삭제한다.

구문

```
int cubrid_set_drop (int conn_handle, string oid, string attr_name, string set_element)
```

- *conn_handle*: 연결 핸들
- *oid*: 원하는 인스턴스의 OID
- *attr_name*: 인스턴스의 원하는 속성 이름
- *set_element*: 삭제하고 싶은 원소의 내용

리턴 값

- 성공 : true
- 실패 : false

예제

```
cubrid_set_drop ($con, $oid, "friend", "James");
cubrid_set_drop ($con, $oid, "friend", "Michael");
```

관련 항목

- **cubrid_set_add**

cubrid_unbuffered_query

설명

cubrid_unbuffered_query 함수는 결과 행을 fetch하거나 버퍼링하지 않고, [cubrid_execute](#) 함수와 마찬가지로 지정된 질의를 서버로 전송한다. 이는 거대한 결과 셋을 생성하는 SQL 문에 의해 점유되는 메모리를 절약할 수 있으며, 해당 SQL 문의 실행이 완료될 때까지 기다리지 않고 첫 번째 레코드가 조회된 직후부터 결과셋에 대한 작업을 수행할 수 있다.

다중 DB 접속 환경에서는 옵션 인수인 *link_identifier*를 지정하여야 한다.

구문

```
resource cubrid_unbuffered_query ( string $query [, int $link_identifier ] )
```

- *query*: 대상 SQL문을 지정한다.
- *link_identifier*: connection을 지정한다. 이를 지정하지 않으면, [cubrid_connect](#) 함수에 의해 최근 성립된 연결을 사용한다.

리턴 값

- **SELECT** 문에서 성공하면 resource를 반환하고, 실패하면 *FALSE*를 반환한다.
- **UPDATE, DELETE, DROP** 등 다른 SQL 문에서 성공하면 **TRUE**를 반환하고, 에러가 발생하면 **FALSE**를 반환한다.

예제

```
<?php
$result = cubrid_unbuffered_query("INSERT INTO mytable (product) values ('kossu')", $link);
if (!$result) {
    echo 'Could not run query: ' . cubrid_error_msg();
    exit;
}
printf("Last inserted record has id %d\n", cubrid_insert_id());
?>
```

cubrid_version

설명

cubrid_version 함수를 사용하면 CUBRID PHP 모듈의 버전 정보를 확인할 수 있다.

구문

```
string cubrid_version ()
```

리턴 값

- n 버전 정보 (예 : "1.2.0")

예제

```
echo cubrid_version();
```

관련 항목

- `cubrid_error_code`
- `cubrid_error_code_facility`

CCI API

CCI 개요

개요

CCI (C Client Interface)는 CUBRID 브로커와 응용 클라이언트 사이에 위치하여, C 기반의 응용 클라이언트가 브로커를 통해 CUBRID 데이터베이스 서버로 접근할 수 있는 인터페이스로서, CAS 응용 서버를 이용하는 도구(예: PHP, ODBC)를 만들기 위한 하부 구조로도 사용된다. 여기서, CUBRID 브로커는 응용 클라이언트로부터 받은 질의를 데이터베이스에 전달하고, 실행 결과를 응용 클라이언트로 전송하는 역할을 수행한다.

CCI를 사용하기 위해서는 헤더 파일과 라이브러리 파일이 필요하다.

- 헤더 파일 : **cas_cci.h**
- 라이브러리 파일
- **\$CUBRID/lib/libcascci.so** (Windows : **cascci.dll**)
- **\$CUBRID/lib/libcascci.a** (Windows : **cascci.lib**)

CCI 를 이용한 응용 프로그램 작성

기본적인 작성 순서는 다음과 같으며, prepared statement 사용을 위해서는 변수에 데이터를 바인딩하는 작업이 추가된다. 이를 예제 1 및 예제 2에 구현하였다. CCI에서는 **SELECT_AUTO_COMMIT** 파라미터가 **ON**으로 설정된 상태에서 **SELECT** 문에 대해서만 자동 커밋 모드를 지원하므로, 그렇지 않은 경우에는 [cci_end_tran\(\)](#) 함수를 이용하여 명시적으로 트랜잭션을 커밋 또는 롤백해야 한다. CCI에서의 자동 커밋 모드 설정 및 사용에 관한 상세한 설명은 [브로커별 파라미터](#)의 **SELECT_AUTO_COMMIT** 및 [cci_end_tran\(\)](#)를 참고한다.

- 데이터베이스 연결 핸들 열기(관련 함수: [cci_connect\(\)](#), [cci_connect_with_url\(\)](#))
- prepared statement를 위한 요청 핸들 얻기 (관련 함수: [cci_prepare\(\)](#))
- prepared statement에 데이터 바인딩하기(관련 함수: [cci_bind_param\(\)](#))
- prepared statement 실행하기(관련 함수: [cci_execute\(\)](#))
- 실행 결과 처리하기(관련 함수: [cci_cursor\(\)](#), [cci_fetch\(\)](#), [cci_get_data\(\)](#), [cci_get_result_info\(\)](#))
- 요청 핸들 닫기(관련 함수: [cci_close_req_handle\(\)](#))
- 데이터베이스 연결 핸들 닫기(관련 함수: [cci_disconnect\(\)](#))

예제 1

```
//Example to execute a simple query
#include <stdio.h>
#include "cas_cci.h"
#define BUFSIZE (1024)
```

```

int
main (void)
{
    int con = 0, req = 0, col count = 0, i, ind;
    int error;
    char *data;
    T_CCI_ERROR cci_error;
    T_CCI_COL_INFO *col_info;
    T_CCI_SQLX_CMD cmd_type;
    char *query = "select * from code";

    //getting a connection handle for a connection with a server
    con = cci_connect ("localhost", 44000, "demodb", "dba", "");
    if (con < 0)
    {
        printf ("cannot connect to database\n");
        return 1;
    }

    //preparing the SQL statement
    req = cci_prepare (con, query, 0, &cci_error);
    if (req < 0)
    {
        printf ("prepare error: %d, %s\n", cci_error.err_code,
            cci_error.err_msg);
        goto handle_error;
    }

    //getting column information when the prepared statement is the SELECT query
    col_info = cci_get_result_info (req, &cmd_type, &col_count);
    if (col_info == NULL)
    {
        printf ("get result info error: %d, %s\n", cci_error.err_code,
            cci_error.err_msg);
        goto handle_error;
    }

    //Executing the prepared SQL statement
    error = cci_execute (req, 0, 0, &cci_error);
    if (error < 0)
    {
        printf ("execute error: %d, %s\n", cci_error.err_code,
            cci_error.err_msg);
        goto handle_error;
    }
    while (1)
    {
        //Moving the cursor to access a specific tuple of results
        error = cci_cursor (req, 1, CCI_CURSOR_CURRENT, &cci_error);
        if (error == CCI_ER_NO_MORE_DATA)
        {
            break;
        }
        if (error < 0)
        {
            printf ("cursor error: %d, %s\n", cci_error.err_code,
                cci_error.err_msg);
            goto handle_error;
        }

        //Fetching the query result into a client buffer
        error = cci_fetch (req, &cci_error);
        if (error < 0)
        {
            printf ("fetch error: %d, %s\n", cci_error.err_code,
                cci_error.err_msg);
            goto handle_error;
        }
        for (i = 1; i <= col count; i++)
        {
            //Getting data from the fetched result

```



```

        error = cci_get_data (req, i, CCI_A_TYPE_STR, &data, &ind);
        if (error < 0)
        {
            printf ("get data error: %d, %d\n", error, i);
            goto handle_error;
        }
        printf ("%s\t", data);
    }
    printf ("\n");
}

//Closing the request handle
error = cci_close_req_handle (req);
if (error < 0)
{
    printf ("close req handle error: %d, %s\n", cci_error.err_code,
            cci_error.err_msg);
    goto handle_error;
}

//Disconnecting with the server
error = cci_disconnect (con, &cci_error);
if (error < 0)
{
    printf ("error: %d, %s\n", cci_error.err_code, cci_error.err_msg);
    goto handle_error;
}

return 0;

handle_error:
if (req > 0)
    cci_close_req_handle (req);
if (con > 0)
    cci_disconnect (con, &cci_error);

return 1;
}

```

예제 2

```

//Example to execute a query with a bind variable

char *query = "select * from nation where name = ?";
char namebuf[128];

//getting a connection handle for a connection with a server
con = cci_connect ("localhost", 44000, "demodb", "dba", "");
if (con < 0)
{
    printf ("cannot connect to database ");
    return 1;
}

//preparing the SQL statement
req = cci_prepare (con, query, 0, &cci_error);
if (req < 0)
{
    printf ("prepare error: %d, %s ", cci_error.err_code,
            cci_error.err_msg);
    goto handle_error;
}

//Binding data into a value
strcpy (namebuf, "Korea");
error =
    cci_bind_param (req, 1, CCI_A_TYPE_STR, &namebuf, CCI_U_TYPE_STRING,
                    CCI_BIND_PTR);
if (error < 0)
{
    printf ("bind param error: %d ", error);
    goto handle_error;
}

```

```
}
```

CCI 에러 코드와 에러 메시지

아래는 CCI에서 출력되는 에러 코드 및 해당 에러 메시지를 정리한 표이다.

에러 코드	에러 메시지	비고
CCI_ER_ALLOC_CON_HANDLE	"Cannot allocate connection handle"	
CCI_ER_ATYPE	"Invalid T_CCI_A_TYPE value"	
CCI_ER_BIND_ARRAY_SIZE	"Array binding size is not specified"	
CCI_ER_BIND_INDEX	"Parameter index is out of range"	바인드할 데이터의 index 가 유효하지 않음.
CCI_ER_COLUMN_INDEX	"Column index is out of range"	
CCI_ER_COMMUNICATION	"Cannot communicate with server"	
CCI_ER_CON_HANDLE	"Invalid connection handle"	
CCI_ER_CONNECT	"Cannot connect to CUBRID CAS"	서버와 연결 시도 시 CAS 접속에 실패함.
CCI_ER_DELETED_TUPLE	"Current row was deleted"	
CCI_ER_FILE	"Cannot open file"	파일을 열거나 읽기/쓰기 실패함.
CCI_ER_HOSTNAME	"Unknown host name"	
CCI_ER_INVALID_CURSOR_POS	"Invalid cursor position"	
CCI_ER_INVALID_URL	"Invalid url string"	
CCI_ER_ISOLATION_LEVEL	"Unknown transaction isolation level"	
CCI_ER_NO_MORE_DATA	"Invalid cursor position"	
CCI_ER_NO_MORE_MEMORY	"Memory allocation error"	사용 가능한 메모리가 부족함.
CCI_ER_OBJECT	"Invalid oid string"	

CCI_ER_OID_CMD	"Invalid T_CCI_OID_CMD value"	
CCI_ER_TRAN_TYPE	"Unknown transaction type"	
CCI_ER_PARAM_NAME	"Invalid T_CCI_DB_PARAM value"	
CCI_ER_REQ_HANDLE	"Cannot allocate request handle"	
CCI_ER_SAVEPOINT_CMD	"Invalid T_CCI_SAVEPOINT_CMD value"	cci_savepoint() 함수의 인자로 유효하지 않은 T_CCI_SAVEPOINT_CMD 값이 사용됨.
CCI_ER_SET_INDEX	"Invalid set index"	T_SET 에 포함된 set 원소를 가져올 때 잘못된 인덱스가 지정 됨.
CCI_ER_STRING_PARAM	"Invalid string argument"	string parameter 가 NULL 이거나 empty string 임.
CCI_ER_THREAD_RUNNING	"Thread is running"	"CCI_EXEC_THREAD 플래그를 주고 cci_execute()를 실행하고, cci_get_thread_result()를 통해 스레드의 실행 결과를 확인하는 경우, 아직 해당 스레드가 실행 중임을 나타냄."
CCI_ER_TRAN_TYPE	"Unknown transaction type"	
CCI_ER_TYPE_CONVERSION	"Type conversion error"	주어진 타입의 값을 실제 데이터의 타입으로 변경할 수 없음.
CCI_ER_DBMS CAS_ER_DBMS	"CUBRID DBMS Error"	서버와 연결 시도 시, CAS 접속은 성공했으나 데이터베이스 접속에 실패함.
CAS_ER_COLLECTION_DOMAIN	"Heterogeneous set is not supported"	지원되지 않는 set 타입임.
CAS_ER_COMMUNICATION	"Cannot receive data from client"	
CAS_ER_DB_VALUE	"Cannot make DB_VALUE"	

CAS_ER_DBSERVER_DISCONNECTED	"Cannot communicate with DB Server"	
CAS_ER_FREE_SERVER	"Cannot process the request. Try again later"	CAS 를 할당할 수 없음.
CAS_ER_INVALID_CALL_STMT	"Illegal CALL statement"	
CAS_ER_NO_MORE_DATA	"Invalid cursor position"	
CAS_ER_NO_MORE_MEMORY	"Memory allocation error"	
CAS_ER_NO_MORE_RESULT_SET	"No More Result"	
CAS_ER_NOT_AUTHORIZED_CLIENT	"Authorization error"	접근이 허용되지 않은 클라이언트임.
CAS_ER_NOT_COLLECTION	"The attribute domain must be the set type"	컬럼 타입이 set 타입이 아님.
CAS_ER_NUM_BIND	"Invalid parameter binding value argument"	바인딩될 개수와 전송된 데이터 개수가 일치하지 않음.
CAS_ER_OBJECT	"Invalid oid"	
CAS_ER_OPEN_FILE	"Cannot open file"	
CAS_ER_PARAM_NAME	"Invalid T_CCI_DB_PARAM value"	"get_db_parameter, set_db_parameter : 유효하지 않은 파라미터 이름임."
CAS_ER_QUERY_CANCEL	"Cannot cancel the query"	
CAS_ER_UNKNOWN_U_TYPE	"Invalid T_CCI_U_TYPE value"	
CAS_ER_TYPE_CONVERSION	"Type conversion error"	
CAS_ER_SCHEMA_TYPE	"Invalid T_CCI_SCH_TYPE value"	
CAS_ER_STMT_POOLING	"Invalid plan"	
CAS_ER_TRAN_TYPE	"Invalid transaction type argument"	
CAS_ER_TYPE_CONVERSION	"Type conversion error"	
CAS_ER_UNKNOWN_U_TYPE	"Invalid T_CCI_U_TYPE value"	

CAS_ER_VERSION	"Version mismatch"	클라이언트와 서버의 버전이 유효하지 않음.
----------------	--------------------	----------------------------

C Type Definition

이름	타입	멤버	설명
T_CCI_ERROR	struct	char err_msg[1024]	데이터베이스
		int err_code	에러 정보 표현
T_CCI_BIT	struct	int size	bit 타입 표현
		char *buf	
T_CCI_DATE	struct	short yr	timestamp,
		short mon	date, time 타
		short day	입 표현
		short hh	
		short mm	
		short ss	
		short ms	
T_CCI_SET	void*		set 타입 표현
T_CCI_COL_INFO	struct	T_CCI_U_TYPE type	SELECT 문에
		char is_non_null	대한 컬럼 정
		short scale	보 표현
		int precision	
		char *col_name	
		char *real_attr	
		char *class_name	
T_CCI_QUERY_RESULT	struct	int result_count	batch 실행에
		int stmt_type	대한 결과
		char *err_msg	
		char oid[32]	
T_CCI_PARAM_INFO	struct	T_CCI_PARAM_MODE mode	input 파라미
		T_CCI_U_TYPE type	터에 대한 정

		short scale	보 표현
		int precision	
T_CCI_U_TYPE	enum	CCI_U_TYPE_UNKNOWN	데이터베이스
		CCI_U_TYPE_NULL	타입 정보
		CCI_U_TYPE_CHAR	
		CCI_U_TYPE_STRING	
		CCI_U_TYPE_NCHAR	
		CCI_U_TYPE_VARNCHAR	
		CCI_U_TYPE_BIT	
		CCI_U_TYPE_VARBIT	
		CCI_U_TYPE_NUMERIC	
		CCI_U_TYPE_INT	
		CCI_U_TYPE_SHORT	
		CCI_U_TYPE_MONETARY	
		CCI_U_TYPE_FLOAT	
		CCI_U_TYPE_DOUBLE	
		CCI_U_TYPE_DATE	
		CCI_U_TYPE_TIME	
		CCI_U_TYPE_TIMESTAMP	
		CCI_U_TYPE_SET	
		CCI_U_TYPE_MULTISSET	
		CCI_U_TYPE_SEQUENCE	
		CCI_U_TYPE_OBJECT	
		CCI_U_TYPE_BIGINT	
		CCI_U_TYPE_DATETIME	
T_CCI_A_TYPE	enum	CCI_A_TYPE_STR	API에서 사
		CCI_A_TYPE_INT	용되는 타
		CCI_A_TYPE_FLOAT	입 정보 표
		CCI_A_TYPE_DOUBLE	현
		CCI_A_TYPE_BIT	

		CCI_A_TYPE_DATE	
		CCI_A_TYPE_SET	
		CCI_A_TYPE_BIGINT	
T_CCI_DB_PARAM	enum	CCI_PARAM_ISOLATION_LEVEL	데이터베이스 파라미터 이 름
		CCI_PARAM_LOCK_TIMEOUT	
		CCI_PARAM_MAX_STRING_LENGTH	
		CCI_PARAM_AUTO_COMMIT	
T_CCI_SCH_TYPE	enum	CCI_SCH_CLASS	
		CCI_SCH_VCLASS	
		CCI_SCH_QUERY_SPEC	
		CCI_SCH_ATTRIBUTE	
		CCI_SCH_CLASS_ATTRIBUTE	
		CCI_SCH_METHOD	
		CCI_SCH_CLASS_METHOD	
		CCI_SCH_METHOD_FILE	
		CCI_SCH_SUPERCLASS	
		CCI_SCH_SUBCLASS	
		CCI_SCH_CONSTRAINT	
		CCI_SCH_TRIGGER	
		CCI_SCH_CLASS_PRIVILEGE	
		CCI_SCH_ATTR_PRIVILEGE	
		CCI_SCH_DIRECT_SUPER_CLASS	
		CCI_SCH_PRIMARY_KEY	
T_CCI_CUBRID_STMT	enum	CUBRID_STMT_ALTER_CLASS	
		CUBRID_STMT_ALTER_SERIAL	
		CUBRID_STMT_COMMIT_WORK	
		CUBRID_STMT_REGISTER_DATABASE	
		CUBRID_STMT_CREATE_CLASS	
		CUBRID_STMT_CREATE_INDEX	
		CUBRID_STMT_CREATE_TRIGGER	

CUBRID_STMT_CREATE_SERIAL
CUBRID_STMT_DROP_DATABASE
CUBRID_STMT_DROP_CLASS
CUBRID_STMT_DROP_INDEX
CUBRID_STMT_DROP_LABEL
CUBRID_STMT_DROP_TRIGGER
CUBRID_STMT_DROP_SERIAL
CUBRID_STMT_EVALUATE
CUBRID_STMT_RENAME_CLASS
CUBRID_STMT_ROLLBACK_WORK
CUBRID_STMT_GRANT
CUBRID_STMT_REVOKE
CUBRID_STMT_STATISTICS
CUBRID_STMT_INSERT
CUBRID_STMT_SELECT
CUBRID_STMT_UPDATE
CUBRID_STMT_DELETE
CUBRID_STMT_CALL
CUBRID_STMT_GET_ISO_LVL
CUBRID_STMT_GET_TIMEOUT
CUBRID_STMT_GET_OPT_LVL
CUBRID_STMT_SET_OPT_LVL
CUBRID_STMT_SCOPE
CUBRID_STMT_GET_TRIGGER
CUBRID_STMT_SET_TRIGGER
CUBRID_STMT_SAVEPOINT
CUBRID_STMT_PREPARE
CUBRID_STMT_ATTACH
CUBRID_STMT_USE
CUBRID_STMT_REMOVE_TRIGGER

		CUBRID_STMT_RENAME_TRIGGER
		CUBRID_STMT_ON_LDB
		CUBRID_STMT_GET_LDB
		CUBRID_STMT_SET_LDB
		CUBRID_STMT_GET_STATS
		CUBRID_STMT_CREATE_USER
		CUBRID_STMT_DROP_USER
		CUBRID_STMT_ALTER_USER
T_CCI_CURSOR_POS	enum	CCI_CURSOR_FIRST
		CCI_CURSOR_CURRENT
		CCI_CURSOR_LAST
T_CCI_TRAN_ISOLATION	enum	TRAN_COMMIT_CLASS_UNCOMMIT_INSTANCE
		TRAN_COMMIT_CLASS_COMMIT_INSTANCE
		TRAN_REP_CLASS_UNCOMMIT_INSTANCE
		TRAN_REP_CLASS_COMMIT_INSTANCE
		TRAN_REP_CLASS_REP_INSTANCE
		TRAN_SERIALIZABLE
T_CCI_PARAM_MODE	enum	CCI_PARAM_MODE_UNKNOWN
		CCI_PARAM_MODE_IN
		CCI_PARAM_MODE_OUT
		CCI_PARAM_MODE_INOUT

cci_bind_param

설명

prepared statement에서 *bind* 변수에 데이터를 바인딩하기 위하여 사용되는 함수이다. 이때, 주어진 *a_type*의 *value*의 값을 실제 바인딩되어야 하는 타입으로 변환하여 저장한다. 이후, [cci_execute\(\)](#)가 호출될 때 저장된 데이터가 서버로 전송된다. 같은 *index*에 대해서 여러 번 **cci_bind_param()**을 호출할 경우 마지막으로 설정한 값이 유효하다.

데이터베이스에 **NULL**을 바인딩할 경우 다음의 두 가지 형태를 가질 수 있다.

- *value* 값이 **NULL** 포인터인 경우.

- `u_type`이 `CCI_U_TYPE_NULL`인 경우.

`flag`에 `CCI_BIND_PTR`이 설정되어 있을 경우 `value` 변수의 포인터만 복사하고(shallow copy) 값은 복사하지 않는다. `flag`가 설정되지 않는 경우 메모리를 할당하여 `value` 변수의 값을 복사(deep copy)한다. 만약 같은 메모리 버퍼를 이용하여 여러 개의 컬럼을 바인딩할 경우라면, `CCI_BIND_PTR flag`를 설정하지 않아야 한다.

`T_CCI_A_TYPE`은 CCI 응용 프로그램 내에서 사용되는 C 언어의 타입을 의미하며, int, float 등의 primitive 타입과 `T_CCI_BIT`, `T_CCI_DATE` 등의 CCI 가 정의한 user-defined 타입으로 구성된다. 각 타입에 대한 식별자는 아래의 표와 같이 정의되어 있다.

a_type	value 타입
<code>CCI_A_TYPE_STR</code>	<code>char**</code>
<code>CCI_A_TYPE_INT</code>	<code>int*</code>
<code>CCI_A_TYPE_FLOAT</code>	<code>float*</code>
<code>CCI_A_TYPE_DOUBLE</code>	<code>double*</code>
<code>CCI_A_TYPE_BIT</code>	<code>T_CCI_BIT*</code>
<code>CCI_A_TYPE_SET</code>	<code>T_CCI_SET*</code>
<code>CCI_A_TYPE_DATE</code>	<code>T_CCI_DATE*</code>
<code>CCI_A_TYPE_BIGINT</code>	<code>int64_t*</code> (Windows 는 <code>__int64*</code>)

또한, `T_CCI_U_TYPE`은 CUBRID 데이터베이스에서 지원하는 타입을 의미하며, 각 타입에 대한 식별자 정의는 아래 표를 참고한다. `cci_bind_param()` 함수에서 이 두 타입이 사용되는 이유는 C 언어가 이해하는 A 타입의 데이터를 데이터베이스가 이해할 수 있는 U 타입의 데이터로 변환하기 위한 정보를 전달하기 위해서이다. `T_CCI_A_TYPE` 및 `T_CCI_U_TYPE` enum은 모두 `cas_cci.h` 파일에 정의되어 있다.

u_type	value 타입
<code>CCI_U_TYPE_CHAR</code>	<code>char**</code>
<code>CCI_U_TYPE_STRING</code>	<code>char**</code>
<code>CCI_U_TYPE_NCHAR</code>	<code>char**</code>
<code>CCI_U_TYPE_VARCHAR</code>	<code>char**</code>
<code>CCI_U_TYPE_BIT</code>	<code>T_CCI_BIT*</code>
<code>CCI_U_TYPE_VARBIT</code>	<code>T_CCI_BIT*</code>
<code>CCI_U_TYPE_NUMERIC</code>	<code>char**</code>
<code>CCI_U_TYPE_INT</code>	<code>int*</code>
<code>CCI_U_TYPE_SHORT</code>	<code>int*</code>

CCI_U_TYPE_MONETARY	Double*
CCI_U_TYPE_FLOAT	float*
CCI_U_TYPE_DOUBLE	Double*
CCI_U_TYPE_DATE	T_CCI_DATE*
CCI_U_TYPE_TIME	T_CCI_DATE*
CCI_U_TYPE_TIMESTAMP	T_CCI_DATE*
CCI_U_TYPE_OBJECT	char**
CCI_U_TYPE_BIGINT	int64_t* (Windows 는 __int64*)
CCI_U_TYPE_DATETIME	T_CCI_DATE*

구문

```
int cci_bind_param(int req_handle, int index, T_CCI_A_TYPE a_type, void *value,
T_CCI_U_TYPE u_type, char flag)
```

- *req_handle* : (IN) prepared statement의 요청 핸들.
- *index* : (IN) 바인딩될 위치이며, 1부터 시작.
- *a_type* : (IN) *value*의 타입.
- *value* : (IN) 바인딩될 데이터 값.
- *u_type* : (IN) 데이터베이스에 반영될 데이터 타입.
- *flag* : (IN) bind_flag (CCI_BIND_PTR).

리턴 값

- 에러 코드(0: 성공)

에러 코드

- CCI_ER_REQ_HANDLE
- CCI_ER_TYPE_CONVERSION
- CCI_ER_BIND_INDEX
- CCI_ER_ATYPE
- CCI_ER_NO_MORE_MEMORY

cci_bind_param_array

설명

prepare된 *req_handle*에 대해서 파라미터 배열을 바인딩한다. 이 후, [cci_execute_array\(\)](#)가 발생할 때 저장된 *value* 포인터에 의해 데이터가 서버로 전송된다. 같은 *index*에 대해서 여러 번 [cci_bind_param_array\(\)](#)이

호출될 경우 마지막 설정된 값이 유효하다. 데이터에 **NULL**을 바인딩할 경우 *null_ind*에 0이 아닌 값을 설정한다.

value 값이 **NULL** 포인터인 경우, 또는 *u_type*이 **CCI_U_TYPE_NULL**인 경우 모든 데이터가 **NULL**로 바인딩되며 *value*에 의해 사용되는 데이터 버퍼는 재사용될 수 없다.

*a_type*에 대한 *value*의 데이터 타입은 [cci_bind_param\(\)](#)의 설명을 참조한다.

구문

```
int cci_bind_param_array(int req_handle, int index, T_CCI_A_TYPE a_type, void *value, int *null_ind, T_CCI_U_TYPE u_type)
```

- *req_handle* : (IN) prepared statement의 요청 핸들
- *index* : (IN) 바인딩될 위치
- *a_type* : (IN) *value*의 타입
- *value* : (IN) 바인딩될 데이터 값
- *null_ind* : (IN) **NULL** indicator array (0 : not **NULL**, 1 : **NULL**)
- *u_type* : (IN) 데이터베이스에 반영될 데이터 타입

리턴 값

- 에러 코드(0 : 성공)

에러 코드

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_TYPE_CONVERSION**
- **CCI_ER_BIND_INDEX**
- **CCI_ER_ATYPE**
- **CCI_ER_BIND_ARRAY_SIZE**

cci_bind_param_array_size

설명

[cci_bind_param_array\(\)](#)에서 사용될 array의 크기를 결정한다. [cci_bind_param_array\(\)](#)가 사용되기 전에 반드시 [cci_bind_param_array_size\(\)](#)가 먼저 호출 되어야 한다.

구문

```
int cci_bind_param_array_size(int req_handle, int array_size)
```

- *req_handle* : (IN) prepared statement의 요청 핸들
- *array_size* : (IN) binding array size

리턴 값

- 에러 코드(0 : 성공)

에러 코드

- CCI_ER_REQ_HANDLE

cci_close_req_handle

설명

[cci_prepare\(\)](#)로 획득한 요청 핸들을 종료(close)한다.

구문

```
int cci_close_req_handle(int req_handle)
```

- *req_handle* : (IN) 요청 핸들

리턴 값

- 에러 코드(0 : 성공)

에러 코드

- CCI_ER_REQ_HANDLE
- CCI_ER_COMMUNICATION

cci_col_get

설명

collection type의 속성 값을 가져온다. 클래스 이름이 C이고 set_attr의 domain이 set(multiset, sequence)인 경우 다음의 query와 같다.

```
SELECT a FROM C, TABLE(set_attr) AS t(a) WHERE C = oid;
```

즉, 멤버 개수가 레코드 개수가 된다.

구문

```
int cci_col_get (int conn_handle, char *oid_str, char *col_attr, int *col_size, int *col_type, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) oid
- *col_attr* : (IN) collection 속성 이름
- *col_size* : (OUT) collection 크기 (-1 : null)
- *col_type* : (OUT) collection 타입 (set, multiset, sequence : u_type)
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 요청 핸들

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_CONNECT
- CCI_ER_OBJECT
- CCI_ER_DBMS

cci_col_seq_drop

설명

sequence 속성 값에 index(base:1) 번째의 멤버를 drop시킨다. 다음은 seq 속성 값에서 첫 번째 값을 삭제하는 예이다.

```
cci_col_seq_drop(con_id, oid_str, seq_attr, 1, err_buf);
```

구문

```
int cci_col_seq_drop (int conn_handle, char *oid_str, char *col_attr, int index,  
T_CCI_ERROR *err_buf)
```

- *conn_handle*: (IN) 연결 핸들
- *oid_str*: (IN) oid
- *col_attr*: (IN) collection 속성 이름
- *index*: (IN) 인덱스
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_CONNECT
- CCI_ER_OBJECT
- CCI_ER_DBMS

cci_col_seq_insert

설명

sequence 속성 값에서 index(base:1) 번째에 멤버를 추가시킨다. 다음은 seq 속성 값에서 1번에 값 'a'를 추가하는 예이다.

```
cci_col_seq_insert(con_id, oid_str, seq_attr, 1, "a", err_buf);
```

구문

```
int cci_col_seq_insert (int conn_handle, char *oid_str, char *col_attr, int index, char *value, T_CCI_ERROR *err_buf)
```

- *conn_handle*: (IN) 연결 핸들
- *oid_str*: (IN) oid
- *col_attr*: (IN) collection 속성 이름
- *index*: (IN) 인덱스
- *value*: (IN) 순차적 엘리먼트(스트링)
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_CONNECT
- CCI_ER_OBJECT
- CCI_ER_DBMS

cci_col_seq_put

설명

sequence 속성 값에 index(base:1) 번째의 멤버를 새로운 값으로 대체한다.. 다음은 seq 속성 값에서 1번 값을 'a'로 대체하는 예이다.

```
cci_col_seq_put(con_id, oid_str, seq_attr, 1, "a", err_buf);
```

구문

```
int cci_col_seq_put (int conn_handle, char *oid_str, char *col_attr, int index, char *value, T_CCI_ERROR *err_buf)
```

- *conn_handle*: (IN) 연결 핸들
- *oid_str*: (IN) oid
- *col_attr*: (IN) collection 속성 이름
- *index*: (IN) 인덱스
- *value*: (IN) 순차적인 값
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_CONNECT
- CCI_ER_OBJECT
- CCI_ER_DBMS

cci_col_set_add

설명

set 속성 값에 member 하나를 추가한다. 다음은 set 속성 값에 'a'를 추가하는 예이다.

```
cci_col_set_add(con_id, oid_str, set_attr, "a", err_buf);
```

구문

```
int cci_col_set_add (int conn handle, char *oid_str, char *col_attr, char *value,  
T_CCI_ERROR *err_buf)
```

- *conn_handle*: (IN) 연결 핸들
- *oid_str*: (IN) oid
- *col_attr*: (IN) collection 속성 이름
- *value*: (IN) set 엘리먼트
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_CONNECT
- CCI_ER_OBJECT
- CCI_ER_DBMS

cci_col_set_drop

설명

set 속성 값에서 멤버 하나를 drop시킨다. 다음은 set 속성 값에서 'a'를 삭제하는 예이다.

```
cci_col_set_drop(con_id, oid_str, set_attr, "a", err_buf);
```

구문

```
int cci_col_set_drop (int conn handle, char *oid_str, char *col_attr, char *value,  
T_CCI_ERROR *err_buf)
```

- *conn_handle*: (IN) 연결 핸들

- *oid_str*: (IN) oid
- *col_attr*: (IN) collection 속성 이름
- *value*: (IN) set 엘리먼트(스트링)
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드

cci_col_size

설명

set(seq) 속성의 개수를 가져온다.

구문

```
int cci_col_size (int conn_handle, char *oid_str, char *col_attr, int *col_size,
T_CCI_ERROR *err_buf)
```

- *conn_handle*: (IN) 연결 핸들
- *oid_str*: (IN) oid
- *col_attr*: (IN) collection 속성 이름
- *col_size*: (OUT) collection 크기 (-1 : NULL)
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0 : 성공)

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_CONNECT
- CCI_ER_OBJECT
- CCI_ER_DBMS

cci_connect

설명

데이터베이스 서버에 대한 연결 핸들을 할당받고 해당 서버와 연결을 시도한다. 서버 연결에 성공하면 연결 핸들 ID를 반환하고, 실패하면 에러 코드를 반환한다.

구문

```
int cci_connect(char *ip, int port, char *db_name, char *db_user, char *db_password)
```

- *ip*: (IN) 서버 IP 문자 스트링 (호스트 이름)
- *port*: (IN) 브로커 포트(**\$CUBRID/conf/cubrid_broker.conf** 파일에 설정된 포트를 사용)
- *db_name*: (IN) 데이터베이스 이름
- *db_user*: (IN) 데이터베이스 사용자 이름
- *db_passwd*: (IN) 데이터베이스 사용자 비밀번호

리턴 값

- 성공: 연결 핸들 ID (int)
- 실패: 에러 코드

에러 코드

- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_HOSTNAME**
- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_CONNECT**

cci_connect_with_url

설명

url 인자로 전달된 접속 정보를 이용하여 데이터베이스로 연결을 시도한다. CCI에서 HA 기능을 사용하는 경우, 이 함수의 *url* 인자에 active 서버의 연결 정보 및 장애 발생 시 failover할 standby 서버의 연결 정보를 명시해야 한다. 서버 연결에 성공하면 연결 핸들 ID를 반환하고, 실패하면 에러 코드를 반환한다.

구문

```
int cci_connect_with_url (char *url [, char *db user, char *db password ])

<url> ::=
cci:CUBRID:<host>:<db_name>:<db_user>:<db_password>:[?<properties>]

<properties> ::= <property> [&<property>]
<property> ::= althosts=<alternative_hosts> [&rcetime=<time>]
<alternative_hosts> ::= <standby broker1 host>:<port> [,<standby broker2 host>:<port>]

<host> := HOSTNAME | IP_ADDR
<time> := SECOND
```

- *url*: (IN) 서버 연결 정보 문자 스트링
- *host*: 마스터 데이터베이스의 호스트 이름 또는 IP 주소
- *db_name*: 데이터베이스 이름
- *db_user*: 데이터베이스 사용자 이름
- *db_password*: 데이터베이스 사용자 비밀번호

- **alhosts**=*standby_broker1_host, standby_broker2_host, ...* : active 서버에 연결할 수 없는 경우, 그 다음으로 연결을 시도(failover)할 standby 서버의 브로커 정보를 나타낸다. failover할 브로커를 여러 개 지정할 수 있고, **alhosts**에 나열한 순서대로 연결을 시도한다.
- **rctime** : 장애가 발생했던 active 브로커에 연결을 시도하는 주기이다. 장애 발생 후 **alhosts**에 명시한 브로커로 접속하여(failover) 트랜잭션을 종료한 후, **rctime**만큼 시간이 경과할 때마다 마스터 데이터베이스의 active 브로커에 연결을 시도한다. 디폴트 값은 600초이다.
- **db_user**: (IN) 데이터베이스 사용자 이름
- **db_passwd**: (IN) 데이터베이스 사용자 비밀번호

리턴 값

- 성공 : 연결 핸들 ID (int)
- 실패 : 에러 코드

에러 코드

- **CCI_ER_NO_MORE_MEMORY**
- **CCI_ER_HOSTNAME**
- **CCI_ER_INVALID_URL**
- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

예제

```
--connection URL string when a property(alhosts) specified for HA
URL=cci:CUBRID:127.0.0.1:31000:db1:::alhosts=127.0.0.2:31000,127.0.0.3:31000

--connection URL string when properties(alhosts,rctime) specified for HA
URL=cci:CUBRID:127.0.0.1:31000:db1:::alhosts=127.0.0.2:31000,127.0.0.3:31000&rctime=600
```

cci_cursor

설명

[cci_execute\(\)](#)로 실행한 질의 결과 내의 특정 레코드에 접근하기 위하여 요청 핸들에 설정된 커서를 이동시킨다. 인자로 지정되는 *origin* 변수 값과 *offset* 값을 통해 커서의 위치가 이동되며, 이동할 커서의 위치가 유효하지 않을 경우 **CCI_ER_NO_MORE_DATA**를 반환한다.

구문

```
int cci_cursor(int req_handle, int offset, T_CCI_CURSOR_POS origin, T_CCI_ERROR *err_buf)
```

- *req_handle*: (IN) 요청 핸들
- *offset*: (IN) 이동할 오프셋

- *origin*: (IN) 커서 위치를 나타내는 변수로서, 타입은 **T_CCI_CURSOR_POS**이다. **T_CCI_CURSOR_POS** enum은 **CCI_CURSOR_FIRST**, **CCI_CURSOR_CURRENT**, **CCI_CURSOR_LAST**의 세 가지 값으로 구성된다.
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드 (0: 성공)

에러 코드

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_NO_MORE_DATA**
- **CCI_ER_COMMUNICATION**

예제

```
//the cursor moves to the first record
cci_cursor(req, 1, CCI_CURSOR_FIRST, &err_buf);

//the cursor moves to the next record
cci_cursor(req, 1, CCI_CURSOR_CURRENT, &err_buf);

//the cursor moves to the last record
cci_cursor(req, 1, CCI_CURSOR_LAST, &err_buf);

//the cursor moves to the previous record
cci_cursor(req, -1, CCI_CURSOR_CURRENT, &err_buf);
```

cci_cursor_update

설명

*cursor_pos*의 커서 위치에 대해서 *index* 번째의 컬럼 값을 *value* 값으로 update한다. 데이터베이스에 **NULL**로 update할 경우 *value*를 **NULL**로 한다. update할 수 있는 조건은 [cci_prepare](#)를 참조한다. *a_type*에 대한 *value*의 데이터 타입은 다음과 같다.

a_type	value 타입
CCI_A_TYPE_STR	char**
CCI_A_TYPE_INT	int*
CCI_A_TYPE_FLOAT	float*
CCI_A_TYPE_DOUBLE	double*
CCI_A_TYPE_BIT	T_CCI_BIT*
CCI_A_TYPE_SET	T_CCI_SET
CCI_A_TYPE_DATE	T_CCI_DATE*
CCI_A_TYPE_BIGINT	int64_t* (Windows 는 __int64*)

구문

```
int cci_cursor_update(int req_handle, int cursor_pos, int index, T_CCI_A_TYPE a_type, void *value, T_CCI_ERROR *err_buf)
```

- *req_handle*: (IN) 요청 핸들
- *cursor_pos*: (IN) 커서 위치
- *index*: (IN) 컬럼 인덱스
- *a_type*: (IN) *value* 타입
- *value*: (IN) 새로운 값
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0 : 성공)

에러 코드

- CCI_ER_REQ_HANDLE
- CCI_ER_TYPE_CONVERSION
- CCI_ER_ATYPE

cci_disconnect

설명

*conn_handle*에 대해 생성된 모든 요청 핸들을 삭제한다. 트랜잭션이 진행 중일 경우 [cci_end_tran](#)을 실행한 다음 삭제된다.

구문

```
int cci_disconnect(int conn_handle, T_CCI_ERROR * err_buf)
```

- *conn_handle*: (IN) 연결 핸들
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0 : 성공)

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_DBMS
- CCI_ER_COMMUNICATION

cci_end_tran

설명

현재 진행 중인 트랜잭션에 대해서 커밋(commit)이나 롤백(rollback)을 수행한다. 이때, 열려 있는 요청 핸들은 모두 종료되고, 데이터베이스 서버와 연결이 해제된다. 단, 서버와 연결이 끊어진 후에도 해당 연결 핸들은 유효하며, 이는 [cci_connect](#) 함수로 연결 핸들을 하나 할당 받은 경우와 동일한 상태다. *type*이 **CCI_TRAN_COMMIT**으로 지정되면 트랜잭션을 커밋하고, **CCI_TRAN_ROLLBACK**으로 지정되면 트랜잭션을 롤백한다.

구문

```
int cci_end_tran(int conn_handle, char type, T_CCI_ERROR *err_buf)
```

- *conn_handle*: (IN) 연결 핸들
- *type*: (IN) **CCI_TRAN_COMMIT** 또는 **CCI_TRAN_ROLLBACK**
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0: 성공)

에러 코드

- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_TRAN_TYPE**

참고 사항

SELECT 문에 대해서 자동 커밋 모드를 지원하며, 이를 적용하기 위해서는 **cubrid_broker.conf** 파일에 **SELECT_AUTO_COMMIT=ON**을 추가해야 한다. 단, 자동 커밋은 prepared statement가 *n*개인 경우, 서버로부터 *n*개 질의문 전부에 대해 결과 셋을 가져오기(fetch)한 시점에만 자동 커밋이 수행되므로 주의한다. 아래는 이에 관한 예제이다.

예제 1

```
$sql1 = "select * from db_user";
$sql2 = "select * from db_class where owner name = ?";

$result = cubrid_execute($con, $sql1); //select 핸들 1개, fetch 완료-> 자동 커밋
if ($result) {
    while ($row = cubrid_fetch ($result))
    {
        echo ($row[0]);

        $req = cubrid_prepare ($con, $sql2);
        cubrid_bind ($req, 1, $row[0]);
        $res = cubrid_execute ($req); //select 핸들 1개, fetch 완료 -> 자동 커밋
```

```
}
}
```

예제 2

```
$sql1 = "select * from db_user";
$sql2 = "select * from db_class where owner_name = ?";

$req = cubrid_prepare ($con, $sql2);
$result = cubrid_execute($con, $sql1); //select 핸들 2개, 1개만 fetch 완료 -> 자동 커밋하지
않음
if ($result) {
    while ($row = cubrid_fetch ($result))
    {
        echo ($row[0]);

        cubrid_bind ($req, 1, $row[0]);
        $res = cubrid_execute ($req); //모든 select의 fetch 완료 -> 자동 커밋
    }
}
```

예제 3

```
$sql1 = "select * from db user";
$sql2 = "insert into a values (?)";

$result = cubrid_execute($con, $sql1); //select 핸들 1개, fetch 완료 -> 자동 커밋
if ($result) {
    while ($row = cubrid_fetch ($result))
    {
        echo ($row[0]);

        $req = cubrid_prepare ($con, $sql2);
        cubrid_bind ($req, 1, $row[0]);
        $res = cubrid_execute ($req); //insert는 자동 커밋하지 않음
    }
}
```

예제 4

```
$sql1 = "select * from db_user";
$sql2 = "insert into a values (?)";

$req = cubrid_prepare ($con, $sql2);
$result = cubrid_execute($con, $sql1); //insert는 fetch가 없으므로 자동 커밋하지 않음
if ($result) {
    while ($row = cubrid_fetch ($result))
    {
        echo ($row[0]);

        cubrid_bind ($req, 1, $row[0]);
        $res = cubrid_execute ($req); //insert는 자동 커밋하지 않음
    }
}
```

cci_execute

설명

[cci_prepare\(\)](#)를 수행한 SQL 문(prepared statement)을 실행한다. 이 함수의 인자로 요청 핸들, *flag*, fetch되는 컬럼의 최대 크기, 오류 정보를 담을 **T_CCI_ERROR** 구조체 변수의 주소가 지정된다.

*flag*를 통해 서버로부터 질의 결과를 가져오는 방식을 동기식 또는 비동기식으로 설정할 수 있다. *flag*를 **CCI_EXEC_QUERY_ALL**로 설정하면 prepared statement를 실행한 후 질의 결과를 한번에 가져오는 동기 방식 (sync_mode)으로 설정되며, **CCI_EXEC_ASYNC**로 설정하면 질의 결과가 생성될 때마다 즉시 결과를 가져오는 비동기 방식(async_mode)으로 설정된다. 디폴트로 설정된 *flag*는 **CCI_EXEC_QUERY_ALL**이며, 이 경우 다음의 규칙이 적용된다.

- 리턴 값은 첫 번째 질의에 대한 결과이다.
- 어느 하나의 질의에서 에러가 발생할 경우 execute는 실패한 것으로 처리된다.
- q1; q2; q3와 같이 구성된 질의에 대해서 q1을 성공하고 q2에서 에러가 난 경우 q1의 수행 결과는 유효하다. 즉, 에러가 발생했을 때 이전에 성공한 질의 수행에 대해서 롤백하지 않는다.
- 질의가 성공적으로 수행된 경우 두 번째 질의에 대한 결과는 [cci_next_result](#)를 통해서 얻을 수 있다.

*max_col_size*는 prepared statement의 컬럼이 **CHAR**, **VARCHAR**, **NCHAR**, **VARNCHAR**, **BIT**, **VARBIT**일 경우 클라이언트로 전송되는 컬럼의 크기를 결정하기 위한 값이며, 0으로 설정된 경우에는 전체 데이터를 받는다.

구문

```
int cci_execute(int req_handle, char flag, int max_col_size, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) prepared statement의 요청 핸들
- *flag* : (IN) exec flag (**CCI_EXEC_ASYNC** 또는 **CCI_EXEC_QUERY_ALL**)
- *max_col_size* : (IN) fetch되는 컬럼의 크기
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 성공
 - **SELECT** : sync mode인 경우 결과 행의 개수를 반환. async mode인 경우 0을 반환.
 - **INSERT**, **UPDATE** : 반영된 행의 개수
 - 기타 질의 : 0
- 실패 : 에러 코드

에러 코드

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_BIND**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

cci_execute_array

설명

prepared statement에 하나 이상의 값이 바인딩되는 경우, 바인딩되는 변수의 값을 배열(array)로 전달받아 각각의 값을 변수에 바인딩하여 질의를 실행한다.

데이터를 바인딩하기 위해서는 [cci_bind_param_array_size\(\)](#)함수를 호출하여 배열의 크기를 지정한 후, [cci_bind_param_array\(\)](#) 함수를 이용하여 각각의 값을 변수에 바인딩하고, [cci_execute_array\(\)](#)함수를 호출하여 질의를 실행한다.

[cci_execute\(\)](#)함수를 호출하면 질의 수행 결과 셋을 가져올 수 있으나, [cci_execute_array\(\)](#)함수는 *query_result* 변수로 수행된 질의 개수를 반환한다. 실행 결과에 대한 정보를 얻기 위해서는 아래와 같은 매크로를 이용할 수 있다. 매크로에서 입력받는 각 인자에 대한 유효성 검사가 이루어지지 않으므로 주의한다. *query_result* 변수의 사용이 끝나면 [cci_query_result_free](#) 함수를 이용하여 질의 결과를 삭제해야 한다.

매크로	리턴 값 타입	의미
CCI_QUERY_RESULT_RESULT	int	결과 개수
CCI_QUERY_RESULT_ERR_MSG	char*	질의에 대한 에러 메시지
CCI_QUERY_RESULT_STMT_TYPE	int (T_CCI_CUBRID_STMT enum)	질의문의 타입

구문

```
int cci_execute_array(int req_handle, T_CCI_QUERY_RESULT **query_result, T_CCI_ERROR *err_buf)
```

- *req_handle*: (IN) prepared statement의 요청 핸들
- *query_result*: (OUT) 질의 결과(수행된 질의 개수)
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 성공: 수행된 질의의 개수
- 실패: 에러 코드

에러 코드

- CCI_ER_REQ_HANDLE
- CCI_ER_BIND
- CCI_ER_DBMS
- CCI_ER_COMMUNICATION

예제

```
char *query =
    "update participant set gold = ? where host_year = ? and nation_code = 'KOR'";
int gold[2];
char *host_year[2];
int null_ind[2];
T_CCI_QUERY_RESULT *result;
int n_executed;
...

req = cci_prepare (con, query, 0, &cci_error);
if (req < 0)
{
    printf ("prepare error: %d, %s\n", cci_error.err_code,
            cci_error.err_msg);
    goto handle_error;
}

gold[0] = 20;
host_year[0] = "2004";

gold[1] = 15;
host_year[1] = "2008";

null_ind[0] = null_ind[1] = 0;
error = cci_bind_param_array_size (req, 2);
if (error < 0)
{
    printf ("bind param array size error: %d\n", error);
    goto handle_error;
}

error =
    cci_bind_param_array (req, 1, CCI_A_TYPE_INT, &gold, null_ind,
        CCI_U_TYPE_INT);
if (error < 0)
{
    printf ("bind_param_array error: %d\n", error);
    goto handle_error;
}

error =
    cci_bind_param_array (req, 2, CCI_A_TYPE_STR, &host_year, null_ind,
        CCI_U_TYPE_INT);
if (error < 0)
{
    printf ("bind param array error: %d\n", error);
    goto handle_error;
}

n_executed = cci_execute_array (req, &result, &cci_error);
if (n_executed <
0)
{
    printf ("execute error: %d, %s\n", cci_error.err_code,
            cci_error.err_msg);
    goto handle_error;
}

for (i = 1; i <= n_executed;
i++)
{
    printf ("query %d\n", i);
    printf ("result count = %d\n", CCI_QUERY_RESULT_RESULT (result, i));
    printf ("error message = %s\n", CCI_QUERY_RESULT_ERR_MSG (result, i));
    printf ("statement type = %d\n",
            CCI_QUERY_RESULT_STMT_TYPE (result, i));
}
error = cci_query_result_free (result, n_executed);
```

```

    if (error <
0)
    {
        printf ("query_result free: %d\n", error);
        goto
handle_error;
    }

```

cci_execute_batch

설명

CCI에서 **INSERT/UPDATE/DELETE**와 같은 DML 질의를 사용하는 경우에는 여러 작업을 한번에 처리할 수 있는데, 이러한 배치 작업을 위해서 [cci_execute_array\(\)](#) 함수와 **cci_execute_batch()** 함수가 이용될 수 있다. 단, **cci_execute_batch()** 함수에서는 prepared statement를 사용할 수 없다.

인자로 지정된 *num_sql_stmt* 개의 *sql_stmt*를 수행하며, *query_result* 변수로 수행된 질의 개수를 반환한다. 실행 결과에 대한 정보를 얻기 위해서는 [cci_execute_array\(\)](#) 함수에서 사용할 수 있는 매크로([CCI_QUERY_RESULT_RESULT](#), [CCI_QUERY_RESULT_ERR_MSG](#), [CCI_QUERY_RESULT_STMT_TYPE](#))를 이용할 수 있다. 매크로에서 입력받는 각 인자에 대한 유효성 검사가 이루어지지 않으므로 주의한다. *query_result* 변수의 사용이 끝나면 [cci_query_result_free](#) 함수를 이용하여 질의 결과를 삭제해야 한다.

구문

```

int cci_execute_batch(int conn_handle, int num_sql_stmt, char **sql_stmt,
T_CCI_QUERY_RESULT **query_result, T_CCI_ERROR *err_buf)

```

- *conn_handle*: (IN) 연결 핸들
- *num_sql_stmt*: (IN) *sql_stmt*의 개수
- *sql_stmt*: (IN) SQL 문 array
- *query_result*: (OUT) *sql_stmt*의 결과
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 성공: 수행된 질의의 개수
- 실패: 에러 코드

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_DBMS
- CCI_ER_COMMUNICATION
- CCI_ER_NO_MORE_MEMORY
- CCI_ER_CONNECT

예제

```
char **queries;
T_CCI_QUERY_RESULT *result;
int n_queries, n_executed;
...

count = 3;
queries = (char **) malloc (count * sizeof (char *));
queries[0] =
    "insert into athlete(name, gender, nation code, event) values('Ji-sung Park', 'M',
'KOR', 'Soccer')";
queries[1] =
    "insert into athlete(name, gender, nation code, event) values('Joo-young Park', 'M',
'KOR', 'Soccer')";
queries[2] =
    "select * from athlete order by code desc for orderby num() < 3";
//calling cci execute batch()
n_executed = cci_execute_batch (con, count, queries, &result, &cci_error);
if (n_executed < 0)
{
    printf ("execute batch: %d, %s\n", cci_error.err code,
        cci_error.err_msg);
    goto handle_error;
}
printf ("%d statements were executed.\n", n_executed);

for (i = 1; i <= n_executed; i++)
{
    printf ("query %d\n", i);
    printf ("result count = %d\n", CCI_QUERY_RESULT_RESULT (result, i));
    printf ("error message = %s\n", CCI_QUERY_RESULT_ERR_MSG (result, i));
    printf ("statement type = %d\n",
        CCI_QUERY_RESULT_STMT_TYPE (result, i));
}

error = cci_query_result_free (result, n_executed);
if (error <
0)
{
    printf ("query result free: %d\n", error);
    goto handle_error;
}
```

cci_execute_result

설명

[cci_execute\(\)](#)에 의해 수행된 질의의 수행 결과(statement type, result count)를 가져온다. 각각의 질의에 대한 결과는 [CCI_QUERY_RESULT_STMT_TYPE](#), [CCI_QUERY_RESULT_RESULT](#)를 통해서 가져온다. 사용된 질의 결과를 [cci_query_result_free](#)를 통해 삭제해야 한다.

구문

```
int cci_execute_result(int req_handle, T_CCI_QUERY_RESULT **query_result, T_CCI_ERROR
*err_buf)
```

- *req_handle*: (IN) prepared statement의 요청 핸들
- *query_result*: (OUT) 쿼리 결과
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 성공: 수행된 질의의 개수
- 실패: 에러 코드

에러 코드

- CCI_ER_REQ_HANDLE
- CCI_ER_COMMUNICATION

예제

```
T_CCI_QUERY_RESULT *qr;
...

cci_execute( ... );
res = cci_execute_result(req_h, &qr, &err_buf);
if (res < 0) {
    /* error */
}
else {
    for (i=1 ; i <= res ; i++) {
        result_count = CCI_QUERY_RESULT_RESULT(qr, i);
        stmt_type = CCI_QUERY_RESULT_STMT_TYPE(qr, i);
    }
    cci_query_result_free(qr, res);
}
```

cci_fetch

설명

[cci_execute\(\)](#)로 실행한 질의 결과를 서버 측 CAS로부터 fetch하여 클라이언트 버퍼에 저장한다. fetch된 질의 결과에서 특정 컬럼의 데이터는 [cci_get_data\(\)](#)함수를 이용해서 확인할 수 있다.

구문

```
int cci_fetch(int req_handle, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) 요청 핸들
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0: 성공)

cci_fetch_buffer_clear

설명

클라이언트 버퍼에 임시 저장된 레코드를 삭제한다.

구문

```
int cci_fetch_buffer_clear(int req_handle)
```

- *req_handle* : (IN) 요청 핸들

리턴 값

- 에러 코드(0: 성공)

에러 코드

- CCI_ER_REQ_HANDLE

cci_fetch_sensitive

설명

서버에서 클라이언트로 결과가 전송될 때 sensitive column에 대해서 변경된 값으로 전송되도록 한다.

*req_handle*에 의한 결과가 sensitive result가 아닐 경우 [cci_fetch](#)와 동일하다. 리턴 값이

CCI_ER_DELETED_TUPLE일 경우 해당 레코드는 삭제된 경우이다.

구문

```
int cci_fetch_sensitive(int req_handle, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) 요청 핸들
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0: 성공)

에러 코드

- CCI_ER_REQ_HANDLE
- CCI_ER_NO_MORE_DATA
- CCI_ER_COMMUNICATION
- CCI_ER_DBMS
- CCI_ER_DELETED_TUPLE

cci_fetch_size

설명

[cci_fetch](#)를 통해 서버에서 클라이언트로 전송되는 레코드의 개수를 정한다.

구문

```
int cci_fetch_size(int req_handle, int fetch_size)
```

- *req_handle* : (IN) 요청 핸들
- *fetch_size* : (IN) fetch size

리턴 값

- 에러 코드(0: 성공)

에러 코드

- CCI_ER_REQ_HANDLE

cci_get_bind_num

설명

입력 바인딩(input binding) 개수를 가져온다. prepare 시 사용된 SQL 문이 여러 개의 질의로 구성되어 있을 경우, 전체 질의에서 사용된 입력 바인딩 개수를 나타낸다.

구문

```
int cci_get_bind_num(int req_handle)
```

- *req_handle* : (IN) prepared statement에 대한 요청 핸들

리턴 값

- 입력 바인딩 개수

에러 코드

- CCI_ER_REQ_HANDLE

cci_get_class_num_objs

설명

class_name 클래스의 객체 개수와 사용하고 있는 페이지 수를 가져온다. flag가 1일 경우 대략의 값을 가져오고, 0일 경우 정확한 값을 가져온다.

구문

```
int cci_get_class_num_objs(int conn_handle, char *class_name, int flag, int *num_objs, int *num_pages, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *class_name* : (IN) 클래스 이름
- *flag* : (IN) 0 또는 1
- *num_objs* : (OUT) 객체 수
- *num_pages* : (OUT) 페이지 수
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0: 성공)

에러 코드

- CCI_ER_REQ_HANDLE
- CCI_ER_COMMUNICATION
- CCI_ER_CONNECT

CCI_GET_COLLECTION_DOMAIN

설명

*u_type*이 set, multiset, sequence type인 경우 set, multiset, sequence의 domain을 가져온다. *u_type*이 set type이 아닐 경우 리턴 값은 *u_type*과 같다.

구문

```
#define CCI_GET_COLLECTION_DOMAIN(u_type)
```

리턴 값

- Type (CCI_U_TYPE)

cci_get_cur_oid

설명

Execute에서 **CCI_INCLUDE_OID**가 설정된 경우 현재 fetch된 레코드의 OID를 가져온다. OID는 page, slot, volume에 의한 스트링으로 표현된다.

구문

```
int cci_get_cur_oid(int req_handle, char *oid_str_buf)
```

- *conn_handle* : (IN) 요청 핸들
- *oid_str_buf* : (OUT) OID 스트링

리턴 값

- 에러 코드(0: 성공)

에러 코드

- CCI_ER_REQ_HANDLE

cci_get_data

설명

현재 fetch된 결과에 대해서 *col_no* 번째의 값을 가져온다. 주어진 *type* 인자에 따라 *value* 변수의 타입이 결정되고, 이에 따라 *value* 변수로 값 또는 포인터가 복사된다.

값을 복사하는 경우 *value* 변수로 전달되는 주소에 대한 메모리가 할당되어 있어야 한다. 포인터 복사의 경우 응용 클라이언트 라이브러리 내의 포인터를 반환하는 것이므로, 다음 **cci_get_data()** 함수 호출 시 해당 값이 유효하지 않게 되므로 주의한다.

포인터 복사에 의해 반환된 포인터는 해제(free)하면 안 된다. 단, 타입이 **CCI_A_TYPE_SET**인 경우 **T_CCI_SET** 타입의 set 포인터를 메모리에 할당한 후 이를 반환하므로, set 포인터를 사용한 후에는 [cci_set_free\(\)](#) 함수를 이용하여 할당된 메모리를 해제해야 한다. 아래는 *type* 인자와 그에 대응하는 *value*의 데이터 타입을 정리한 표이다.

type	value 타입	의미
CCI_A_TYPE_STR	char**	pointer copy
CCI_A_TYPE_INT	int*	value copy
CCI_A_TYPE_FLOAT	float*	value copy
CCI_A_TYPE_DOUBLE	double*	value copy
CCI_A_TYPE_BIT	T_CCI_BIT*	value copy (pointer copy for each member)
CCI_A_TYPE_SET	T_CCI_SET*	memory alloc and value copy
CCI_A_TYPE_DATE	T_CCI_DATE*	value copy
CCI_A_TYPE_BIGINT	int64_t* (Windows 는 _int64*)	value copy

구문

```
int cci_get_data(int req_handle, int col_no, int type, void *value, int *indicator)
```

- *req_handle* : (IN) 요청 핸들
- *col_no* : (IN) 컬럼 인덱스. 1부터 시작.
- *type* : (IN) *value* 변수의 데이터 타입(**T_CCI_A_TYPE**에 정의된 타입을 사용)
- *value* : (OUT) 데이터를 저장할 변수의 주소
- *indicator* : (OUT) **NULL** indicator. (-1 : **NULL**)
 - *type*이 **CCI_A_TYPE_STR**인 경우 : **NULL**이면 -1을 반환하고, **NULL**이 아니면 *value*에 저장된 문자열의 길이를 반환
 - *type*이 **CCI_A_TYPE_STR**이 아닌 경우 : **NULL**이면 -1을 반환하고, **NULL**이 아니면 0을 반환

리턴 값

- 에러 코드(0 : 성공)

에러 코드

- CCI_ER_REQ_HANDLE
- CCI_ER_TYPE_CONVERSION
- CCI_ER_COLUMN_INDEX
- CCI_ER_ATYPE

cci_get_db_parameter

설명

데이터베이스에 설정된 파라미터를 가져온다. *param_name*에 대한 *value*의 데이터 타입은 다음과 같다.

param_name	value 타입	note
CCI_PARAM_ISOLATION_LEVEL	int*	get/set
CCI_PARAM_LOCK_TIMEOUT	int*	get/set
CCI_PARAM_MAX_STRING_LENGTH	int*	get only

구문

```
int cci_get_db_parameter(int conn_handle, T_CCI_DB_PARAM param_name, void *value, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *param_name* : (IN) 데이터베이스 파라미터 이름
- *value* : (OUT) 파라미터 값
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0: 성공)

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_PARAM_NAME
- CCI_ER_DBMS
- CCI_ER_COMMUNICATION
- CCI_ER_CONNECT

cci_get_db_version

설명

DBMS (Database Management System) 버전을 가져온다.

구문

```
int cci_get_db_version(int conn_handle, char *out_buf, int out_buf_size)
```

- *conn_handle*: (IN) 연결 핸들
- *out_buf*: (OUT) 결과 버퍼
- *out_buf_size*: (IN) *out_buf* 크기

리턴 값

- 에러 코드(0: 성공)

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_COMMUNICATION
- CCI_ER_CONNECT

cci_get_result_info

설명

prepared statement가 **SELECT**일 경우, 이 함수를 이용하여 실행 결과에 대한 컬럼 정보가 저장되어 있는 **T_CCI_COL_INFO** 구조체를 가져올 수 있다. **SELECT** 질의가 아닌 경우, **NULL**을 반환하고 *num* 값은 0이 된다.

T_CCI_COL_INFO 구조체에서 컬럼 정보를 가져오기 위해서 구조체에 직접 접근해도 되지만, 다음과 같이 정의된 매크로를 이용하여 정보를 가져올 수 있다. 각 매크로의 인자로 **T_CCI_COL_INFO** 구조체의 주소와 컬럼 인덱스가 지정되며, 매크로는 **SELECT** 질의에 대해서만 호출할 수 있다. 매크로에서 입력받는 각 인자에 대한 유효성 검사가 이루어지지 않으므로 주의한다. 매크로 리턴 값의 타입이 *char**인 경우 메모리 포인터를 해제(free)하지 않아야 한다.

매크로	리턴 값 타입	의미
CCI_GET_RESULT_INFO_TYPE	T_CCI_U_TYPE	컬럼의 type
CCI_GET_RESULT_INFO_SCALE	short	컬럼의 scale
CCI_GET_RESULT_INFO_PRECISION	int	컬럼의 precision
CCI_GET_RESULT_INFO_NAME	char*	컬럼의 이름
CCI_GET_RESULT_INFO_ATTR_NAME	char*	컬럼의 속성 이름

CCI_GET_RESULT_INFO_CLASS_NAME	char*	컬럼의 클래스 이름
CCI_GET_RESULT_INFO_IN_NON_NULL	char (0 or 1)	컬럼이 NULL 인지 여부

구문

```
T_CCI_COL_INFO* cci_get_result_info(int req_handle, T_CCI_SQLX_CMD *cmd_type, int *num)
```

- *req_handle*: (IN) prepared statement에 대한 요청 핸들
- *cmd_type*: (OUT) command 타입
- *num*: (OUT) **SELECT** 문의 컬럼 개수(*cmd_type*이 **SQLX_CMD_SELECT**일 경우)

리턴 값

- 성공 : result info 포인터
- 실패 : **NULL**

예제

```
col_info = cci_get_result_info (req, &cmd_type, &col_count);
if (col_info == NULL)
{
    printf ("get result info error: %d, %s\n", cci_error.err_code,
            cci_error.err_msg);
    goto handle_error;
}
for (i = 1; i <= col_count; i++)
{
    printf ("%12s = %d\n", "type", CCI_GET_RESULT_INFO_TYPE (col_info, i));
    printf ("%12s = %d\n", "scale",
            CCI_GET_RESULT_INFO_SCALE (col_info, i));
    printf ("%12s = %d\n", "precision",
            CCI_GET_RESULT_INFO_PRECISION (col_info, i));
    printf ("%12s = %s\n", "name", CCI_GET_RESULT_INFO_NAME (col_info, i));
    printf ("%12s = %s\n", "attr name",
            CCI_GET_RESULT_INFO_ATTR_NAME (col_info, i));
    printf ("%12s = %s\n", "class name",
            CCI_GET_RESULT_INFO_CLASS_NAME (col_info, i));
    printf ("%12s = %s\n", "is_non_null",
            CCI_GET_RESULT_INFO_IS_NON_NULL (col_info, i) ? "true" : "false");
}
```

CCI_GET_RESULT_INFO_ATTR_NAME

설명

prepare된 **SELECT** 문의 *index* 번째 컬럼의 실제 속성 이름을 가져오는 매크로이다. 속성 이름이 없는 경우(상수값, 함수 등)는 ""(empty string)을 반환한다. 지정된 인자 *res_info*가 **NULL**인지, *index*가 유효한지에 대한 검사는 하지 않는다. 반환된 메모리 포인터는 사용자가 **free()**를 통해 제거할 수 없다.

구문

```
#define CCI_GET_RESULT_INFO_ATTR_NAME(T_CCI_COL_INFO* res_info, int index)
```

- *res_info*: (IN) [cci_get_result_info](#)에 의한 컬럼 정보 포인터
- *index*: (IN) 컬럼 인덱스

리턴 값

- 속성 이름(char*)

CCI_GET_RESULT_INFO_CLASS_NAME

설명

prepare된 **SELECT** 문의 *index* 번째의 클래스 이름을 가져오는 매크로이다. 지정된 인자 *res_info*가 **NULL**인지, *index*가 유효한지에 대한 검사는 하지 않는다. 반환된 메모리 포인터는 사용자가 **free()**를 통해 제거할 수 없다. 반환된 값은 **NULL**을 가질 수 있다.

구문

```
#define CCI_GET_RESULT_INFO_CLASS_NAME(T_CCI_COL_INFO* res_info, int index)
```

- *res_info*: (IN) [cci_get_result_info](#)에 의한 컬럼 정보 포인터
- *index*: (IN) 컬럼 인덱스

리턴 값

- 클래스 이름(char*)

CCI_GET_RESULT_INFO_IS_NON_NULL

설명

prepare된 **SELECT** 문의 *index* 번째 컬럼이 nullable인지에 대한 값을 가져오는 매크로이다. 지정된 인자 *res_info*가 **NULL**인지, *index*가 유효한지에 대한 검사는 하지 않는다.

구문

```
#define CCI_GET_RESULT_INFO_IS_NON_NULL(T_CCI_COL_INFO* res_info, int index)
```

- *res_info*: (IN) [cci_get_result_info](#)에 의한 컬럼 정보 포인터
- *index*: (IN) 컬럼 인덱스

리턴 값

- 0 : nullable
- 1 : non **NULL**

CCI_GET_RESULT_INFO_NAME

설명

prepare된 **SELECT** 문의 *index* 번째의 컬럼 이름을 가져오는 매크로이다. 지정된 인자 *res_info*가 **NULL**인지, *index*가 유효한지에 대한 검사는 하지 않는다. 반환된 메모리 포인터는 사용자가 **free()**를 통해 제거할 수 없다.

구문

```
#define CCI_GET_RESULT_INFO_NAME(T_CCI_COL_INFO* res_info, int index)
```

- *res_info*: (IN) [cci_get_result_info](#)에 의한 컬럼 정보 포인터
- *index*: (IN) 컬럼 인덱스

리턴 값

- 컬럼 이름(char*)

CCI_GET_RESULT_INFO_PRECISION

설명

prepare된 **SELECT** 문의 *index* 번째의 precision을 가져오는 매크로이다. 지정된 인자 *res_info*가 **NULL**인지, *index*가 유효한지에 대한 검사는 하지 않는다.

구문

```
#define CCI_GET_RESULT_INFO_PRECISION(T_CCI_COL_INFO* res_info, int index)
```

- *res_info*: (IN) [cci_get_result_info](#)에 의한 컬럼 정보 포인터
- *index*: (IN) 컬럼 인덱스

리턴 값

- precision (int)

CCI_GET_RESULT_INFO_SCALE

설명

prepare된 **SELECT** 문의 *index* 번째 컬럼의 scale을 가져오는 매크로이다. 지정된 인자 *res_info*가 **NULL**인지, *index*가 유효한지에 대한 검사는 하지 않는다.

구문

```
#define CCI_GET_RESULT_INFO_SCALE(T_CCI_COL_INFO* res_info, int index)
```

- *res_info*: (IN) [cci_get_result_info](#)에 의한 컬럼 정보 포인터
- *index*: (IN) 컬럼 인덱스

리턴 값

- scale (int)

CCI_GET_RESULT_INFO_TYPE

설명

prepare된 **SELECT** 문의 *index* 번째 컬럼의 타입을 가져오는 매크로이다. 지정된 인자 *res_info*가 **NULL**인지, *index*가 유효한지에 대한 검사는 하지 않는다.

구문

```
#define CCI_GET_RESULT_INFO_TYPE(T_CCI_COL_INFO* res_info, int index)
```

- *res_info* : (IN) [cci_get_result_info](#)에 의한 컬럼 정보 포인터
- *index* : (IN) 컬럼 인덱스

리턴 값

- 컬럼 타입 (T_CCI_U_TYPE)

cci_glo_append_data

설명

call `append_data(length, buf)` on <glo object>

구문

```
int cci_glo_append_data(int conn_handle, char *oid_str, int length, char *buf, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) glo 객체(OID 스트링)
- *length* : (IN) 덧붙일 바이트 수
- *buf* : (IN) 데이터 버퍼
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 덧붙인 바이트 수

cci_glo_compress_data

설명

call `compress_data()` on <glo object>

구문

```
int cci_glo_compress_data(int conn_handle, char *oid_str, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) glo 객체(OID 스트링)

- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드

cci_glo_data_size

설명

call data_size() on <glo object>

구문

```
int cci_glo_data_size(int conn_handle, char *oid_str, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) glo 객체(OID 스트링)
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 데이터 크기

cci_glo_delete_data

설명

call data_seek(start_pos) on <glo object>

call delete_data(length) on <glo object>

구문

```
int cci_glo_delete_data(int conn_handle, char *oid_str, int start_pos, int length, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) glo 객체(OID 스트링)
- *start_pos* : (IN) 시작 오프셋
- *length* : (IN) 삭제할 바이트 수
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 삭제한 바이트 수

cci_glo_destroy_data

설명

call destroy_data() on <glo object>

구문

```
int cci_glo_destroy_data(int conn_handle, char *oid_str, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) glo 객체(OID 스트링)
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드

cci_glo_insert_data

설명

call data_seek(start_pos) on <glo object>

call insert_data(length, buffer) on <glo object>

구문

```
int cci_glo_insert_data(int conn_handle, char *oid_str, int start_pos, int length, char *buf, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) glo 객체(OID 스트링)
- *start_pos* : (IN) 시작 오프셋
- *length* : (IN) 삽입할 바이트 수
- *buf* : (IN) 데이터 버퍼
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 삽입한 바이트 수

cci_glo_load

설명

주어진 glo 객체의 데이터를 *out_fd*에 저장한다. *oid_str*에는 [cci_glo_new\(\)](#) 함수에서 OUT으로 받은 glo 객체 정보를 입력하고, *out_fd*에는 glo 컬럼의 값을 저장할 파일 이름을 입력한다.

구문

```
int cci_glo_load(int conn_handle, char *oid_str, int out_fd, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) glo 객체(OID 스트링)
- *out_fd* : (IN) 결과 파일 기술자(descriptor)
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0: 성공)

cci_glo_new

설명

*class_name*의 glo 클래스에 *filename*에 의한 파일로 glo 객체를 만든다. glo 컬럼을 파일로 로드하거나 glo 컬럼에 데이터를 저장(save)할 파일 이름을 *filename*에 지정한다. *filename*이 **NULL**인 경우 glo 객체만을 생성한다(cci 2.3 이상).

구문

```
int cci_glo_new(int conn_handle, char *class_name, char *filename, char *oid_str, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *class_name* : (IN) glo 클래스 이름
- *filename* : (IN) 소스 파일
- *oid_str* : (OUT) glo 객체(OID 스트링)
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0: 성공)

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_DBMS
- CCI_ER_COMMUNICATION
- CCI_ER_FILE
- CCI_ER_CONNECT

예제 1

CCI에서 glo 데이터 타입을 사용하기 위해서는 기본적으로 `cci_glo_new()`, `cci_glo_save()`, `cci_glo_load()` 함수가 사용된다. 다음은 string 타입의 name 컬럼과 glo 타입의 photo 컬럼이 있는 person 테이블에 대하여, a.out이라는 파일을 glo 객체로 photo 컬럼에 저장하는 예제이다.

```
#include <stdio.h>
#include <stdlib.h>
#include <error.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "cas cci.h"

extern int errno;

int main() {
    char oid_str[24] = {0x00};
    char *query = "insert into person(name, photo) values(?,?);";
    char *i_filename = "a.out";

    int con, ret, req;
    int fd = -1;
    T CCI_ERROR err_buf;

    con = ret = req = 0;

    con = cci_connect("localhost", 30000, "testdb", "dba", "");
    if (con < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        return -1;
    }

    // Getting OID values from "glo" table
    ret = cci_glo_new(con, "glo", i_filename, oid_str, &err_buf);
    if (ret < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        cci_disconnect(con, &err_buf);
        return -1;
    }

    req = cci_prepare(con, query, CCI_PREPARE_INCLUDE_OID, &err_buf);
    if (req < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        cci_disconnect(con, &err_buf);
        return -1;
    }

    ret = cci_bind_param(req, 1, CCI_A_TYPE_STR, i_filename, CCI_U_TYPE_CHAR, (char)NULL);
    if (ret < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);

        cci_disconnect(con, &err_buf);
        return -1;
    }

    // Binding OID values
    ret = cci_bind_param(req, 2, CCI_A_TYPE_STR, oid_str, CCI_U_TYPE_OBJECT, (char)NULL);
    if (ret < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        cci_disconnect(con, &err_buf);
        return -1;
    }

    ret = cci_execute(req, 0, 0, &err_buf);
    if (ret < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        cci_disconnect(con, &err_buf);
        return -1;
    }
}
```

```

    ret = cci_end_tran(con, CCI_TRAN_COMMIT, &err_buf);
    if (ret < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        cci_disconnect(con, &err_buf);
        return -1;
    }

    cci_disconnect(con, &err_buf);

    return 1;
}

```

예제 2

다음은 CCI를 통해 데이터베이스에 저장된 데이터를 파일 시스템의 b.out으로 카피하는 예제이다.

```

#include <stdio.h>
#include <stdlib.h>
#include <error.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "cas_cci.h"

extern int errno;

int main() {
    char *oid_str;
    char *query = "select photo from person where name = 'a.out'";
    char *o_filename = "b.out";
    int con, ret, req, ind;
    int fd = -1;
    T_CCI_ERROR err_buf;

    con = ret = req = 0;

    con = cci_connect("localhost", 30000, "testdb", "dba", "");
    if (con < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        return -1;
    }

    req = cci_prepare(con, query, CCI_PREPARE_INCLUDE_OID, &err_buf);
    if (req < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        cci_disconnect(con, &err_buf);
        return -1;
    }

    ret = cci_execute(req, 0, 0, &err_buf);
    if (ret < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        cci_disconnect(con, &err_buf);
        return -1;
    }

    ret = cci_cursor(req, 1, CCI_CURSOR_CURRENT, &err_buf);
    if (ret < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        cci_disconnect(con, &err_buf);
        return -1;
    }

    ret = cci_fetch(req, &err_buf);
    if (ret < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        cci_disconnect(con, &err_buf);
        return -1;
    }

    ret = cci_get_data(req, 1, CCI_A_TYPE_STR, &oid_str, &ind);
}

```

```

    if (ret < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        cci_disconnect(con, &err_buf);
        return -1;
    }

    // opening b.out
    fd = open(o_filename, O_CREAT | O_RDWR | O_TRUNC);
    if (fd < 0) {
        perror(o_filename);
        return -1;
    }

    // copying glo data from database to the fd
    ret = cci_glo_load(con, oid_str, fd, &err_buf);
    if (ret < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        printf("TDBG : %s:%d \n", FILE__, LINE__);
        cci_disconnect(con, &err_buf);
        close(fd);
        return -1;
    }
    close(fd);

    ret = cci_end_tran(con, CCI_TRAN_COMMIT, &err_buf);
    if (ret < 0) {
        printf("%d:%s\n", err_buf.err_code, err_buf.err_msg);
        cci_disconnect(con, &err_buf);
        return -1;
    }
    cci_close_req_handle(req);
    cci_disconnect(con, &err_buf);

    return 1;
}

```

cci_glo_read_data

설명

call `data_seek(start_pos)` on <glo object>

call `read_data(length, buffer)` on <glo object>

구문

```
int cci_glo_read_data(int conn_handle, char *oid_str, int start_pos, int length, char *buf,
T_CCI_ERROR *err_buf)
```

- *conn_handle*: (IN) 연결 핸들
- *oid_str*: (IN) glo 객체(OID 스트링)
- *start_pos*: (IN) 시작 오프셋
- *length*: (IN) 읽을 바이트 수
- *buf*: (OUT) 데이터 버퍼
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 읽은 바이트 수

cci_glo_save

설명

파일 시스템에 있는 *filename* 파일의 내용을 glo 객체에 저장한다. *oid_str*에는 [cci_glo_new\(\)](#)에서 OUT으로 받은 glo 객체 정보를 입력한다.

구문

```
int cci_glo_save(int conn_handle, char *oid_str, char *filename, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) glo 객체(OID 스트링)
- *filename* : (IN) 소스 파일
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0: 성공)

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_DBMS
- CCI_ER_COMMUNICATION
- CCI_ER_FILE
- CCI_ER_CONNECT

cci_glo_truncate_data

설명

call data_seek(start_pos) on <glo object>

call truncate_data() on <glo object>

구문

```
int cci_glo_truncate_data(int conn_handle, char *oid_str, int start_pos, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) glo 객체(OID 스트링)
- *start_pos* : (IN) 시작 오프셋
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 삭제한 바이트 수

cci_glo_write_data

설명

call data_seek(start_pos) on <glo object>

call write_data(length, buffer) on <glo object>

구문

```
int cci_glo_write_data(int conn_handle, char *oid_str, int start_pos, int length, char *buf, T_CCI_ERROR *err_buf)
```

- *conn_handle*: (IN) 연결 핸들
- *oid_str*: (IN) glo 객체 (OID 스트링)
- *start_pos*: (IN) 시작 오프셋
- *length*: (IN) 기록할 바이트 수
- *buf*: (IN) 데이터 버퍼
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 기록한 바이트 수

CCI_IS_SET_TYPE, CCI_IS_MULTISSET_TYPE, CCI_IS_SEQUENCE_TYPE, CCI_IS_COLLECTION_TYPE

설명

u_type 이 set, multiset, sequence type인지를 검사한다.

구문

```
#define CCI_IS_SET_TYPE(u_type)  
#define CCI_IS_MULTISSET_TYPE(u_type)  
#define CCI_IS_SEQUENCE_TYPE(u_type)  
#define CCI_IS_COLLECTION_TYPE(u_type)
```

리턴 값

- **CCI_IS_SET_TYPE**
 - 1 : set
 - 0 : not set
- **CCI_IS_MULTISSET_TYPE**
 - 1 : multiset
 - 0 : not multiset
- **CCI_IS_SEQUENCE_TYPE**
 - 1 : sequence

- 0 : not sequence
- **CCI_IS_SET_TYPE**
 - 1 : collection (set, multiset, sequence)
 - 0 : not collection

cci_is_updatable

설명

[cci_prepare\(\)](#)를 수행한 SQL 문이 업데이트 가능한 질의인지 확인하는 함수이다. 업데이트 가능한 질의이면 1을 반환한다.

구문

```
int cci_is_updatable(int req_handle)
```

- *req_handle* : (IN) prepared statement에 대한 요청 핸들

리턴 값

- 1 : updatable
- 0 : not updatable

에러 코드

- CCI_ER_REQ_HANDLE

cci_next_result

설명

[cci_execute](#) 시 **CCI_EXEC_QUERY_ALL** *flag*가 설정된 경우 실행된 다음 질의의 결과를 가져온다.

next_result에 의해 얻은 질의에 대한 정보는 [cci_get_result_info](#)를 통해서 얻을 수 있다. next_result가 성공적으로 수행될 경우 현재의 질의에 대한 정보로 수정된다.

에러 코드가 **CAS_ER_NO_MORE_RESULT_SET**일 경우 더 이상의 result set이 존재하지 않는다.

구문

```
int cci_next_result(int req_handle, T_CCI_ERROR *err_buf)
```

- *req_handle* : (IN) prepared statement의 요청 핸들
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 성공 :
 - **SELECT** (sync mode) : 결과 개수, (async mode) : 0

- **INSERT, UPDATE** : 반영된 레코드 개수
- 기타 : 0
- 실패 : 에러 코드

에러 코드

- **CCI_ER_REQ_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**

cci_oid

설명

CCI_OID_DROP : 해당 oid를 삭제한다.

CCI_OID_IS_INSTANCE : 해당 oid가 instance oid인지를 검사한다.

CCI_OID_LOCK_READ : 해당 oid에 대해 read lock 을 설정한다.

CCI_OID_LOCK_WRITE : 해당 oid에 대해 write lock을 설정한다.

구문

```
int cci_oid(int conn_handle, T_CCI_OID_CMD cmd, char *oid_str, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *cmd* : (IN) CCI_OID_DROP, CCI_OID_IS_INSTANCE, CCI_OID_LOCK_READ, CCI_OID_LOCK_WRITE
- *oid_str* : (IN) oid
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- CCI_OID_IS_INSTANCE
 - 0 : 인스턴스 아님
 - 1 : 인스턴스
 - < 0 : 에러
- CCI_OID_DROP, CCI_OID_LOCK_READ, CCI_OID_LOCK_WRITE
 - 에러 코드(0: 성공)

에러 코드

- **CCI_ER_CON_HANDLE**
- **CCI_ER_CONNECT**
- **CCI_ER_OID_CMD**
- **CCI_ER_OBJECT**

- CCI_ER_DBMS

cci_oid_get

설명

해당 oid의 속성 값을 가져온다. *attr_name*은 속성의 array로서 마지막은 반드시 NULL로 끝나야 한다.

*attr_name*이 NULL인 경우 모든 속성에 대한 정보를 가져온다. Request handle의 형태는 "SELECT attr_name FROM oid_class WHERE oid_class = oid"의 SQL문을 실행했을 때와 동일한 형태이다.

구문

```
int cci_oid_get(int conn_handle, char *oid_str, char **attr_name, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) oid
- *attr_name* : (IN) 속성 목록
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 성공 : 요청 핸들
- 실패 : 에러 코드

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_NO_MORE_MEMORY
- CCI_ER_CONNECT

cci_oid_get_class_name

설명

해당 oid의 클래스 이름을 가져온다.

구문

```
int cci_oid_get_class_name(int conn_handle, char *oid_str, char *out_buf, int out_buf_len, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) oid
- *out_buf* : (OUT) out 버퍼
- *out_buf_len* : (IN) *out_buf* 길이
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_CONNECT
- CCI_ER_OBJECT
- CCI_ER_DBMS

cci_oid_put

설명

해당 oid의 *attr_name* 속성 값을 *new_val_str*으로 설정한다. *attr_name*의 마지막은 반드시 NULL이어야 한다. 모든 타입의 값은 string으로 표현해야 하고, string으로 표현된 값은 서버에서 속성의 타입에 따라 변환되어 데이터베이스에 반영된다. NULL 값을 넣기 위해서는 *new_val_str[i]*의 값을 NULL로 한다.

구문

```
int cci_oid_put(int conn_handle, char *oid_str, char **attr_name, char **new_val_str, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *oid_str* : (IN) oid
- *attr_name* : (IN) 속성 이름 목록
- *new_val_str* : (IN) 새 값의 목록
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0: 성공)

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_CONNECT

cci_oid_put2

설명

해당 oid의 *attr_name* 속성 값을 *new_val*로 설정한다. *attr_name*의 마지막은 반드시 NULL이어야 한다. NULL 값을 넣기 위해서는 *new_val[i]*의 값을 NULL로 지정한다.

*a_type*에 대한 *new_val[i]*의 타입은 다음 표와 같다.

a_type에 대한 new_val[i]의 타입

Type	value type
CCI_A_TYPE_STR	char**
CCI_A_TYPE_INT	int*
CCI_A_TYPE_FLOAT	float*
CCI_A_TYPE_DOUBLE	double*
CCI_A_TYPE_BIT	T_CCI_BIT*
CCI_A_TYPE_SET	T_CCI_SET*
CCI_A_TYPE_DATE	T_CCI_DATE*
CCI_A_TYPE_BIGINT	int64_t* (Windows 는 __int64*)

구문

```
int cci_oid_put2(int conn_handle, char *oidstr, char **attr_name, void **new_val, int *a_type, T_CCI_ERROR *err_buf)
```

- *conn_handle*: (IN) 연결 핸들
- *oid_str*: (IN) oid
- *attr_name*: (IN) 속성 이름 목록
- *new_val*: (IN) 새 값 배열
- *a_type*: (IN) *new_val* 타입 배열
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0: 성공. 음수: 실패)

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_CONNECT

예제

```
char *attr_name[array_size]
void *attr_val[array_size]
int a_type[array_size]
int int_val

...

attr_name[0] = "attr_name0"
attr_val[0] = &int_val
a_type[0] = CCI_A_TYPE_INT
attr_name[1] = "attr_name1"
attr_val[1] = "attr_val1"
a_type[1] = CCI_A_TYPE_STR
```

```
...
attr name[num attr] = NULL

res = cci_put2(con h, oid str, attr name, attr val, a type, &error)
...
```

cci_prepare

설명

SQL 문에 관한 요청 핸들을 획득하여 SQL 실행을 준비한다. 단, SQL 문이 여러 개의 질의로 구성된 경우, 첫 번째 질의에 대해서만 실행을 준비한다. 이 함수의 인자로 연결 핸들, SQL문, *flag*, 오류 정보를 저장할 **T_CCI_ERROR** 구조체 변수의 주소가 지정된다.

*flag*에 **CCI_PREPARE_UPDATABLE** 또는 **CCI_PREPARE_INCLUDE_OID**가 설정될 수 있다. *flag*에 **CCI_PREPARE_UPDATABLE**이 설정되면 갱신 가능한 결과 세트(updatable result set)를 만들 수 있으며, 이 경우 **CCI_PREPARE_INCLUDE_OID**는 자동 설정된다. **CCI_PREPARE_UPDATABLE**이 설정되더라도 모든 질의에 대해 갱신 가능한 결과를 만들 수 있는 것은 아니므로 SQL 문을 prepare한 후 [cci_is_updatable](#) 함수를 이용하여 갱신 가능한 질의인지 확인해야 한다.

갱신 가능한 질의의 조건은 다음과 같다.

- **SELECT** 질의여야 한다.
- 질의 결과에 OID가 포함될 수 있는 질의여야 한다.
- 갱신하고자 하는 컬럼이 **FROM** 절에 명시한 테이블에 속한 컬럼이어야 한다.

구문

```
int cci_prepare(int conn_handle, char *sql_stmt, char flag, T_CCI_ERROR *err_buf)
```

- *conn_handle*: (IN) 연결 핸들
- *sql_stmt*: (IN) SQL 문
- *flag*: (IN) prepare flag (**CCI_PREPARE_UPDATABLE** 또는 **CCI_PREPARE_INCLUDE_OID**)
- *err_buf*: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 성공: 요청 핸들 ID (int)
- 실패: 에러 코드

에러 코드

- **CCI_ER_CON_HANDLE**
- **CCI_ER_DBMS**
- **CCI_ER_COMMUNICATION**
- **CCI_ER_STR_PARAM**

- CCI_ER_NO_MORE_MEMORY
- CCI_ER_CONNECT

CCI_QUERY_RESULT_ERR_MSG

설명

[cci_execute_batch](#) 질의에 대한 에러 메시지를 가져오며, 에러 메시지가 없을 경우 ""(empty string)을 반환하는 매크로이다. 지정된 인자 *query_result*가 **NULL**인지, *index*가 유효한지에 대한 검사는 하지 않는다.

구문

```
#define CCI_QUERY_RESULT_ERR_MSG(T_CCI_QUERY_RESULT* query_result, int index)
```

- *query_result*: (IN) [cci_execute_batch](#)에 의한 쿼리 결과
- *index*: (IN) 컬럼 인덱스(base : 1)

리턴 값

- 에러 메시지

cci_query_result_free

설명

사용된 질의 결과를 삭제한다.

구문

```
int cci_query_result_free(T_CCI_QUERY_RESULT* query_result, int num_query)
```

- *query_result*: (IN) [cci_execute_batch](#)에 의한 쿼리 결과
- *num_query*: (IN) *query_result*의 array 개수

리턴 값

- 에러 코드(0: 성공)

예제

```
T_CCI_QUERY_RESULT *qr;
char **sql_stmt;

...

res = cci_execute_array(conn, &qr, &err_buf);

...

cci_query_result_free(qr, res);
```

CCI_QUERY_RESULT_RESULT

설명

[cci_execute_batch](#) 질의에 대한 result count를 가져오는 매크로이다. 지정된 인자 *query_result*가 **NULL**인지, *index*가 유효한지에 대한 검사는 하지 않는다.

구문

```
#define CCI_QUERY_RESULT_RESULT(T_CCI_QUERY_RESULT* query_result, int index)
```

- *query_result*: (IN) [cci_execute_batch](#)에 의한 쿼리 결과
- *index*: (IN) 컬럼 인덱스(base : 1)

리턴 값

- result count

CCI_QUERY_RESULT_STMT_TYPE

설명

[cci_execute_batch](#) 질의에 의한 statement type을 가져오는 매크로이다. 지정된 인자 *query_result*가 **NULL**인지, *index*가 유효한지에 대한 검사는 하지 않는다.

구문

```
#define CCI_QUERY_RESULT_STMT_TYPE(T_CCI_QUERY_RESULT* query_result, int index)
```

- *query_result*: (IN) [cci_execute_batch](#)에 의한 쿼리 결과
- *index*: (IN) 컬럼 인덱스(base : 1)

리턴 값

- statement type (T_CCI_SQLX_CMD)

cci_savepoint

설명

세이브포인트를 설정하거나 설정된 세이브포인트로 트랜잭션 롤백을 수행한다. *cmd*가 **CCI_SP_SET**로 지정되면 세이브포인트를 설정하고, **CCI_SP_ROLLBACK**인 경우 설정된 세이브포인트까지 트랜잭션을 롤백한다.

구문

```
int cci_savepoint(int conn_handle, T_CCI_SAVEPOINT_CMD cmd, char* savepoint_name,  
T_CCI_ERROR *err_buf)
```

- *conn_handle*: (IN) 연결 핸들
- *cmd*: (IN) **CCI_SP_SET** 또는 **CCI_SP_ROLLBACK**

- `savepoint_name` : (IN) 세이브포인트 이름
- `err_buf` : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드

예제

```
con = cci_connect( ... );
... /* query execute */

/* "savepoint1"이란 이름의 세이브포인트 설정
cci_savepoint(con, CCI_SP_SET, "savepoint1", err_buf);

... /* query execute */

/* 설정된 세이브포인트 "savepoint1"로 롤백 */
cci_savepoint(con, CCI_SP_ROLLBACK, "savepoint1", err_buf);
```

cci_schema_info

설명

스키마 정보를 읽어온다. 성공적으로 수행되었을 경우 결과는 request handle에 의해 관리되고, fetch, getdata를 통해서 결과를 가져올 수 있다. `class_name`, `attr_name`을 pattern match에 의해서 검색해야 할 경우 `flag`를 설정한다.

CCI_CLASS_NAME_PATTERN_MATCH, **CCI_ATTR_NAME_PATTERN_MATCH** 두 개의 `flag`가 사용된다. OR 연산자(|)로 둘을 설정할 수 있다. Pattern match를 사용할 경우 속도가 매우 떨어질 수 있다.

각 `type`에 대한 레코드의 구성은 아래 표와 같다.

타입에 대한 레코드 구성

타입	컬럼 순서	컬럼 이름	컬럼 타입
CCI_SCH_CLASS	1	NAME	char*
	2	TYPE	short 0 : system class 1 : vclass 2 : class 3 : proxy
CCI_SCH_VCLASS	1	NAME	char*
	2	TYPE	short 1 : vclass 3 : proxy
CCI_SCH_ATTRIBUTE	1	NAME	char*

	2	DOMAIN	int
	3	SCALE	int
	4	PRECISION	int
	5	INDEXED	int 1 : indexed
	6	NON_NULL	int 1 : non null
	7	SHARED	int 1 : shared
	8	UNIQUE	int 1 : unique
	9	DEFAULT	void*
	10	ATTR_ORDER	int base : 1
	11	CLASS_NAME	char*
	12	SOURCE_CLASS	char*
	13	IS_KEY	short 1 : key
CCI_SCH_CLASS_METHOD	1	NAME	char*
	2	RET_DOMAIN	int
	3	ARG_DOMAIN	char*
CCI_SCH_METHOD_FILE	1	METHOD_FILE	char*
CCI_SCH_SUPERCLASS	1	CLASS_NAME	char*
	2	TYPE	short
CCI_SCH_SUBCLASS	1	CLASS_NAME	char*
	2	TYPE	short
CCI_SCH_CONSTRAINT	1	TYPE	int 0 : unique 1 : index 2 : reverse unique 3 : reverse index
	2	NAME	char*
	3	ATTR_NAME	char*

	4	NUM_PAGES	int
	5	NUM_KEYS	int
	6	PRIMARY_KEY 1 : primary key	short
	7	KEY_ORDER	short base : 1
CCI_SCH_TRIGGER	1	NAME	char*
	2	STATUS	char*
	3	EVENT	char*
	4	TARGET_CLASS	char*
	5	TARGET_ATTR	char*
	6	ACTION_TIME	char*
	7	ACTION	char*
	8	PRIORITY	float
	9	CONDITION_TIME	char*
	10	CONDITION	char*
CCI_SCH_CLASS_PRIVILEGE	1	CLASS_NAME	char*
	2	PRIVELEGE	char*
	3	GRANTABLE	char*
CCI_SCH_ATTR_PRIVILEGE	1	ATTR_NAME	char*
	2	PRIVILEGE	char*
	3	GRANTABLE	char*
CCI_SCH_PRIMARY_KEY	1	CLASS_NAME	char*
	2	ATTR_NAME	char*
	3	KEY_SEQ	short base : 1
	4	KEY_NAME	char*

Pattern match

CCI_SCH_TYPE	Class name	ATTR_name
CCI_SCH_CLASS (VCLASS)	O	none
CCI_SCH_ATTRIBUTE (CLASS ATTRIBUTE)	O	O

CCI_SCH_CLASS_PRIVILEGE	O	none
CCI_SCH_ATTR_PRIVILEGE	X	O
CCI_SCH_PRIMARY_KEY	O	none

패턴 *flag*가 설정되지 않을 경우 주어진 클래스 이름 또는 속성 이름은 exact string match를 사용하며, 따라서 **NULL**이 주어질 경우 결과는 없다. 패턴 *flag*가 설정되어 있을 경우 클래스 이름 또는 속성 이름이 **NULL**일 경우 "%"와 동일한 결과를 얻는다.

참고 CCI_SCH_CLASS, CCI_SCH_VCLASS의 TYPE 컬럼 : proxy 타입이 추가됨. OLEDB, ODBC, PHP에서 사용할 경우 proxy와 vclass를 구분하지 않고 vclass로 보임.

구문

```
int cci_schema_info(int conn_handle, T_CCI_SCHEMA_TYPE type, char *class_name, char *attr_name, char flag, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *type* : (IN) 스키마 타입
- *class_name* : (IN) 클래스 이름 또는 NULL
- *attr_name* : (IN) 속성 이름 또는 NULL
- *flag* : (IN) 패턴 매칭 flag(CCI_CLASS_NAME_PATTERN_MATCH 또는 CCI_ATTR_NAME_PATTERN_MATCH)
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 성공 : 요청 핸들
- 실패 : 에러 코드

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_DBMS
- CCI_ER_COMMUNICATION
- CCI_ER_SCHEMA_TYPE
- CCI_ER_NO_MORE_MEMORY
- CCI_ER_CONNECT

cci_set_db_parameter

설명

데이터베이스 파라미터를 설정한다. *param_name*에 대한 *value*의 타입은 [cci_get_db_parameter\(\)](#) 를 참조한다.

구문

```
int cci_set_db_parameter(int conn_handle, T_CCI_DB_PARAM param_name, void* value, T_CCI_ERROR *err_buf)
```

- *conn_handle* : (IN) 연결 핸들
- *param_name* : (IN) 데이터베이스 파라미터 이름
- *value* : (IN) 파라미터 값
- *err_buf* : (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드(0: 성공)

에러 코드

- CCI_ER_CON_HANDLE
- CCI_ER_PARAM_NAME
- CCI_ER_DBMS
- CCI_ER_COMMUNICATION
- CCI_ER_CONNECT

cci_set_element_type

설명

[cci_get_data\(\)](#)에 대해 CCI_A_TYPE_SET에 의해 가져온 set에 대해 엘리먼트의 타입을 가져온다.

구문

```
int cci_set_element_type(T_CCI_SET set)
```

- *set* : (IN) cci set 포인터

리턴 값

- 타입

cci_set_free

설명

[cci_get_data\(\)](#)에 대해 CCI_A_TYPE_SET에 의해 가져온 T_CCI_SET 에 할당된 메모리를 제거한다.

구문

```
void cci_set_free(T_CCI_SET set)
```

- *set* : (IN) cci set 포인터

리턴 값

- 없음

cci_set_get

설명

[cci_get_data\(\)](#)에 대해 **CCI_A_TYPE_SET**에 의해 가져온 set에 대해 *index* 번째의 데이터를 가져온다.

*a_type*에 대한 *value*의 타입은 다음과 같다.

a_type	value 타입
CCI_A_TYPE_STR	char**
CCI_A_TYPE_INT	int*
CCI_A_TYPE_FLOAT	float*
CCI_A_TYPE_DOUBLE	double*
CCI_A_TYPE_BIT	T_CCI_BIT*
CCI_A_TYPE_DATE	T_CCI_DATE*
CCI_A_TYPE_BIGINT	int64_t* (Windows 는 __int64*)

구문

```
int cci_set_get(T_CCI_SET set, int index, T_CCI_A_TYPE a_type, void *value, int *indicator)
```

- *set* : (IN) cci set 포인터
- *index* : (IN) set 인덱스 (base : 1)
- *a_type* : (IN) 타입
- *value* : (OUT) 결과 버퍼
- *indicator* : (OUT) null 표시(indicator)

리턴 값

- 에러 코드

cci_set_isolation_level

설명

연결에 대한 트랜잭션 격리 수준(transaction isolation level)을 설정한다. 이 후 주어진 연결에 대한 모든 트랜잭션은 *new_isolation_level*로서 동작한다.

참고 `cci_set_db_parameter()`에 의해 트랜잭션 격리 수준이 설정된 경우에는 현재의 트랜잭션에 대해서만 영향을 주고 트랜잭션이 끝나면 CAS에서 설정된 트랜잭션 격리 수준으로 복귀된다. 연결 전체에 대해서 트랜잭션 격리 수준을 설정할 경우는 반드시 `cci_set_isolation_level()`을 사용해야 한다.

구문

```
int cci_set_isolation_level(int conn handle, T_CCI_TRAN_ISOLATION new_isolation_level,
T_CCI_ERROR *err_buf)
```

- `conn_handle`: (IN) 연결 핸들
- `new_isolation_level`: (IN) 격리 수준(isolation level)
- `err_buf`: (OUT) 데이터베이스 에러 버퍼

리턴 값

- 에러 코드

에러 코드

- `CCI_ER_CON_HANDLE`
- `CCI_ER_CONNECT`
- `CCI_ER_ISOLATION_LEVEL`
- `CCI_ER_DBMS`

cci_set_make

설명

새로운 `CCI_A_TYPE_SET` 타입의 set을 만든다. 만들어진 set은 `cci_bind_param()`을 통해 `CCI_A_TYPE_SET`으로 서버에 전달된다. `cci_set_make()`에 의해 만들어진 set은 반드시 `cci_set_free()`를 통해 사용된 메모리를 제거해야 한다. `u_type`에 따른 `value`의 타입은 `cci_bind_param`을 참고한다.

구문

```
int cci_set_make(T_CCI_SET *set, T_CCI_U_TYPE u_type, int size, void *value, int
*indicator)
```

- `set`: (IN) cci set 포인터
- `u_type`: (IN) 엘리먼트 타입
- `size`: (IN) set 크기
- `value`: (IN) set 엘리먼트
- `indicator`: (IN) null 표시 배열(indicator array)

리턴 값

- 에러 코드

cci_set_max_row

설명

[cci_execute](#)에 의해 수행되는 **SELECT** 문의 결과에 대해 최대 레코드 수를 지정한다. *max* 값이 0일 경우 지정하지 않은 것과 동일하다.

구문

```
int cci_set_max_row(int req_handle, int max)
```

- *req_handle*: (IN) 요청 핸들
- *max*: (IN) 최대 행

리턴 값

- 에러 코드

예제

```
req = cci_prepare( ... );  
cci_set_max_row(req, 1);  
cci_execute( ... );
```

cci_set_size

설명

[cci_get_data](#)() 함수에 대해 CCI_A_TYPE_SET에 의해 가져온 set에 대해 엘리먼트의 개수를 가져온다.

구문

```
int cci_set_size(T_CCI_SET set)
```

- *set*: (IN) cci set 포인터

리턴 값

- 크기

CUBRID 매니저

CUBRID 매니저는 CUBRID 데이터베이스 관리 및 질의 기능을 GUI 환경에서 제공하는 CUBRID 데이터베이스 전용 관리 도구이다. CUBRID 매니저를 사용하면 다양한 관리 작업을 쉽게 할 수 있다. CUBRID 매니저는 데이터베이스 관리 및 응용 개발에 필수적인 질의 편집기 기능을 제공한다.

CUBRID 매니저는 데이터베이스 서버, 브로커가 설치된 호스트에서 동작하는 매니저 서버와 GUI 기반의 클라이언트로 구성되어 있다. CUBRID 매니저 클라이언트 프로그램은 Java 기반으로 작성되어 Java가 실행될 수 있는 환경이면 사용 가능하다.

이 장에서는 설명하는 주요 내용은 다음과 같다.

- CUBRID 매니저 소개
- CUBRID 매니저 실행
- CUBRID 매니저 클라이언트 기능

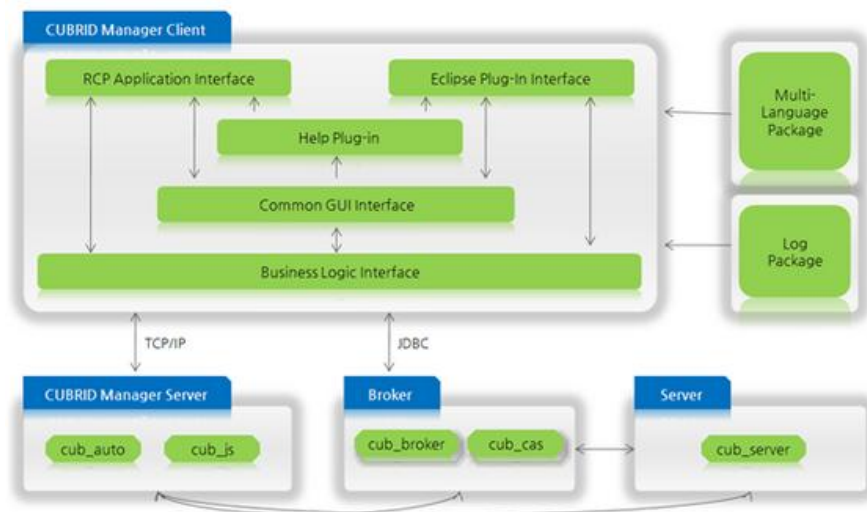
CUBRID 매니저 소개

CUBRID 매니저 구조

CUBRID 매니저의 구조는 아래 그림과 같다. 데이터베이스 서버와 브로커가 설치된 호스트에 CUBRID 매니저 서버가 동작하고, 이 CUBRID 매니저 서버에 CUBRID 매니저 클라이언트가 접속한다.

CUBRID 매니저 서버는 **cub_auto** 프로세스와 **cub_js** 프로세스로 구성된다. CUBRID 매니저 클라이언트에서 CUBRID 매니저 서버에 접속하기 위해서는 반드시 CUBRID 매니저 서버가 구동되어 있어야 하며, **cub_auto** 프로세스와 **cub_js** 프로세스와 상응하는 네트워크(TCP/IP) 포트가 설정되어 있어야 한다. 질의를 수행하기 위해서는 JDBC 연결을 위한 브로커도 구동되어 있어야 한다.

- **cub_auto** 프로세스는 CUBRID 매니저 클라이언트 사용자의 인증처리 및 주기적인 자동화 작업과 진단 정보를 수집하는 기능을 수행한다.
- **cub_js** 프로세스는 CUBRID 매니저 클라이언트로부터 전송된 사용자의 요구를 수행한다.



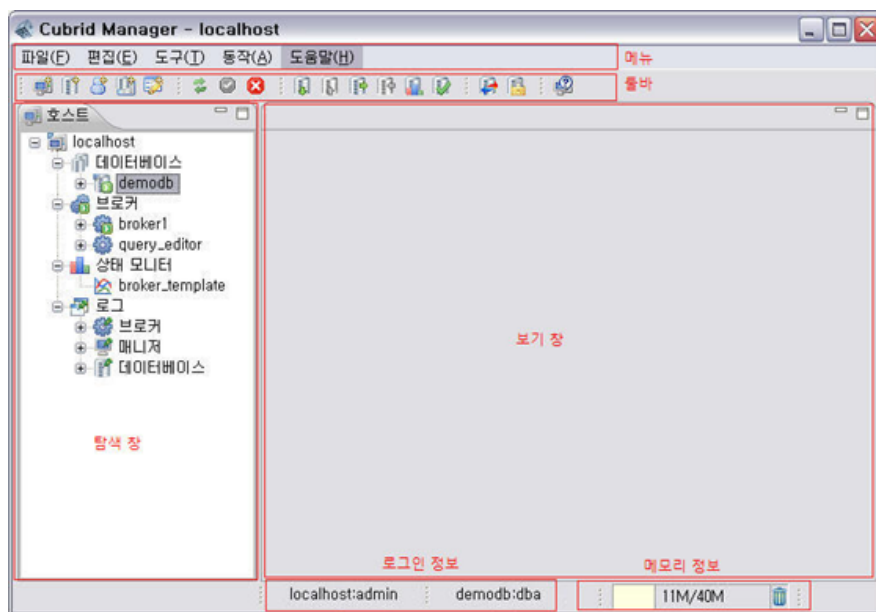
서버 구동에 대한 상세한 정보는 [CUBRID 매니저 서버 환경 설정](#)을 참고한다.

CUBRID 매니저 클라이언트

RCP 응용 프로그램

화면 구성

클라이언트는 메뉴, 탐색, 보기 그리고 상태 표시줄 4개 영역으로 이루어지며, 각 영역별로 고유의 정보를 제공한다.



파일 메뉴

[파일] 메뉴는 호스트 관리 및 연결 메뉴, 파일 저장 메뉴, CUBRID 매니저 기본 설정 메뉴 등으로 구성되어 있다.

 호스트 추가...	
 호스트 삭제	
 연결...	
 연결 끊기	
닫기(C)	Ctrl+W
모든 파일 닫기(E)	Ctrl+Shift+W
 저장(S)	Ctrl+S
 새 이름으로 저장(A)...	
 모든 파일 저장(L)	Ctrl+Shift+S
기본 설정...	
 새 로고침	
종료(X)	

편집 메뉴

[편집] 메뉴는 CUBRID 매니저 내부에서 사용하는 편집 관련 메뉴로 구성되어 있다.

 실행 취소(U)	Ctrl+Z
 다시 실행(B)	Ctrl+Y
 복사(C)	Ctrl+C
 잘라내기(I)	Ctrl+X
 붙여넣기(P)	Ctrl+V
찾기/바꾸기(F)...	Ctrl+F

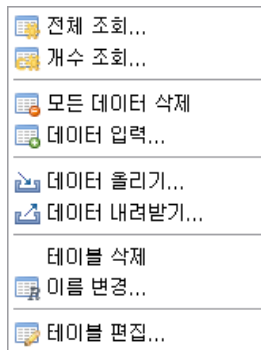
도구 메뉴

[도구] 메뉴는 CUBRID 매니저의 주 메뉴로 서비스 시작/정지를 통제하는 메뉴와 사용자 관리 메뉴로 구성되어 있다.

 새 질의 편집기
 서비스 시작
 서비스 정지
 데이터베이스 시작
 데이터베이스 정지
 브로커 시작...
 브로커 정지...
 비밀번호 변경...
 사용자 관리...

동작 메뉴

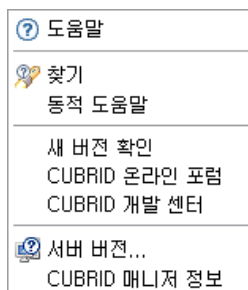
[동작] 메뉴는 동적으로 구성되는 메뉴로 탐색 창에서 마우스 오른쪽 버튼을 클릭해서 수행할 수 있는 기능을 동적으로 보여 주는 메뉴이다. 아래 그림은 테이블을 선택했을 때의 [동작] 메뉴이다.



도움말 메뉴

[도움말] 메뉴는 CUBRID 데이터베이스 매뉴얼을 볼 수 있는 [도움말]과 이를 동적으로 구성하고 찾을 수 있는 메뉴로 구성되어 있다.

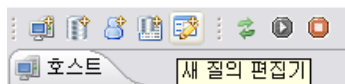
[CUBRID 온라인 포럼] 및 [CUBRID 개발 센터]를 선택하면 CUBRID 사용 관련 팁 및 개발 관련 소식을 접할 수 있다.



툴바

툴바는 전체 기능 중 자주 사용되고 중요한 기능으로 구성되며, 해당 기능이 수행 가능할 때 활성화되고 수행 불가능할 때에는 비활성화된다.

툴바의 아이콘에 마우스 포인터를 대면 아래 그림과 같이 툴팁 도움말이 나타난다.



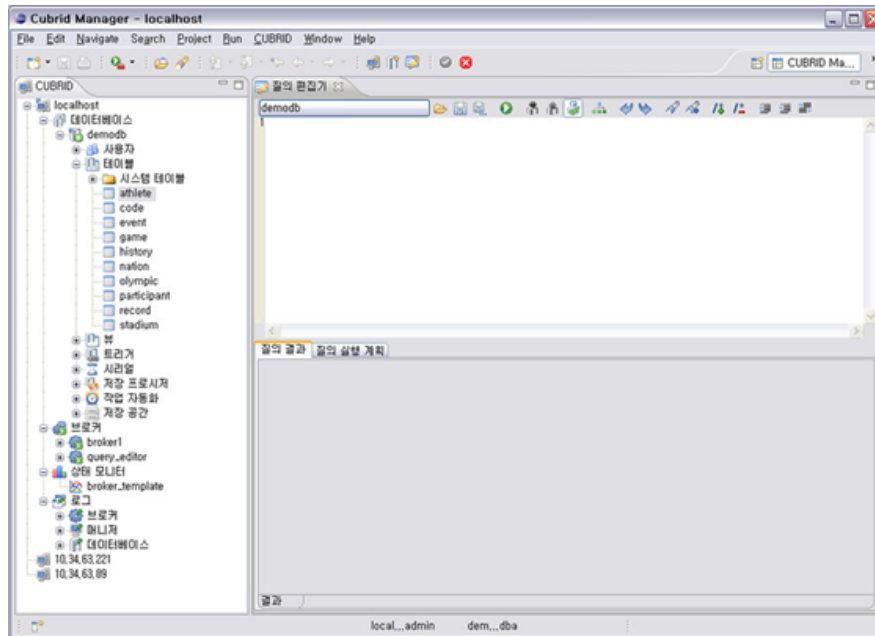
툴바에서 제공하는 기능은 다음과 같다.



Eclipse 플러그인

Eclipse 플러그인으로 CUBRID 매니저 클라이언트에 접속하면, [CUBRID] 메뉴에서 기본 메뉴를 제공하고 트리 창의 기능은 RCP 응용 프로그램과 동일하다. CUBRID 매니저 플러그인을 실행하려면, Eclipse에서 [Window] > [Open Perspective] > [Other]를 선택한 후, [CUBRID Manager]를 선택한다.





CUBRID 메뉴

[CUBRID] 메뉴는 구동 및 설정 그리고 추가에 대한 기본 메뉴만 제공한다. 다른 기능은 RCP 응용 프로그램과 같이 트리 창에서 마우스 오른쪽 버튼을 클릭하여 실행할 수 있다.



툴바

CUBRID 매니저 Eclipse 플러그인의 툴바는 호스트 추가, 데이터베이스 생성, 질의 편집기, 그리고 서비스 시작/정지 기능을 제공한다.



CUBRID 매니저 실행

CUBRID 매니저 서버 환경 설정

CUBRID 매니저 서버의 환경 설정 파일은 **\$CUBRID/conf**에 위치하며, 파일 이름은 **cm.conf**이다.

CUBRID 매니저의 환경 설정 파일에서 주석은 **"#"**으로 처리되며, 매개 변수 이름과 값이 저장된다. 매개 변수 이름과 값 사이에는 공백으로 구분한다. **cm.conf**에서 설정할 수 있는 매개 변수는 다음과 같다.

cm_port

CUBRID 매니저 서버와 클라이언트 사이의 통신 포트를 설정하는 매개 변수로, 디폴트 값은 **8001**로 설정된다. **cm_port**는 **cub_auto**가 사용하는 포트이며, **cm_js**는 자동으로 **cm_port**로 설정한 값보다 1만큼 큰 값을 사용한다. 예를 들어, **cm_port**가 8001로 설정된 경우 **cub_auto**는 8001 포트를 사용하고, **cub_js**는 8002 포트를 사용한다. 따라서 방화벽이 설정된 환경에서 CUBRID 매니저를 구동하려면 반드시 실제로 사용되는 두 개의 포트를 열어야 한다.

monitor_interval

cub_auto의 모니터링 주기를 초 단위로 설정하는 매개 변수로, 디폴트 값은 **5**이다.

allow_user_multi_connection

CUBRID 매니저 서버에 클라이언트가 중복 접속하는 것을 허용하기 위한 매개 변수로, 디폴트 값은 **YES**이다. 따라서 CUBRID 매니저 서버에는 두 개 이상의 CUBRID 매니저 클라이언트가 접속할 수 있으며, 같은 사용자 이름으로 접속할 수도 있다.

server_long_query_time

서버의 진단 항목 중 **slow_query** 항목을 설정할 경우 몇 초 이상을 늦은 질의로 판별할지 결정하는 매개 변수로, 기본 값은 **10**이다. 서버에서 수행된 질의 수행 시간이 매개 변수 설정 값보다 큰 경우, **slow_query**의 개수가 증가한다.

cm_target

브로커와 데이터베이스 서버가 분리된 구조에서 매니저의 메뉴를 해당 서비스에 맞게 출력할 목적으로 지원되는 매개 변수이다. 디폴트 값은 브로커와 데이터베이스 서버가 같이 설치되어 있는 환경을 의미하며, 다음과 같이 설정할 수 있다.

- **cm_target broker, server** : 브로커와 데이터베이스 서버가 같이 있을 경우
- **cm_target broker** : 브로커만 있을 경우
- **cm_target server** : 데이터베이스 서버만 있을 경우

브로커만 설정하면 매니저에서 브로커 관련 메뉴만 출력되고, 데이터베이스 서버만 설정하면 서버 관련 메뉴만 출력된다.

탐색 트리에서 호스트를 마우스 오른쪽 버튼 클릭하고 [속성]을 선택하면, 설정된 정보를 [호스트 정보]에서 확인할 수 있다.

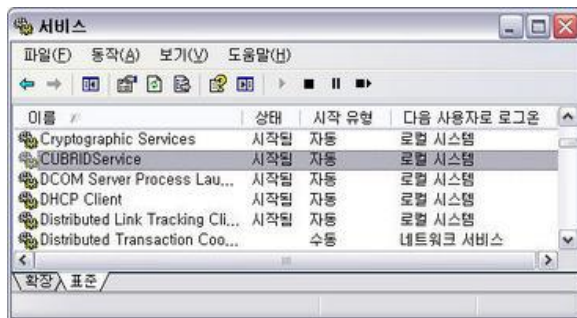


CUBRID 매니저 서버 실행

CUBRID 매니저는 서버와 클라이언트로 구성되며, CUBRID 매니저 서버가 실행 중일 때에만 CUBRID 매니저 클라이언트를 실행할 수 있다.

Windows 환경에서 실행

- [제어판] > [성능 및 유지 관리] > [관리도구] > [서비스]에서 CUBRIDService를 통해 실행할 수 있다.



- CUBRID 서비스 트레이(CUBRID Service Tray)를 클릭하고 [CUBRID Server] > [Start]를 선택한다.



참고 CUBRID가 실행 중일 때, CUBRID 서비스 트레이에서 [Exit]를 선택하면 해당 서버에서 실행 중인 모든 서비스와 프로세스가 중지되므로 주의한다.

Linux 환경에서 실행

셸에서 다음의 명령 중 하나를 입력하면 CUBRID 매니저 서버가 실행된다.

- %cubrid service start
- %cubrid manager start

CUBRID 매니저 서버 로그

CUBRID 매니저 서버를 통해 이루어지는 작업이나 오류에 관한 로그는 **\$CUBRID/log/manager**에 기록되며, 실행 중인 서버의 PID는 **\$CUBRID/var/manager**에 기록된다. 기록되는 파일은 다음과 같다.

- **cub_js.access.log** : cub_js가 처리하는 작업을 기록한다.
- **cub_js.error.log** : cub_js의 작업 처리 중 발생한 에러를 기록한다.
- **cub_auto.access.log** : 서버에 로그인한 클라이언트에 대한 정보를 기록한다.
- **cub_auto.error.log** : 서버 로그인에 실패한 클라이언트에 대한 정보를 기록한다.
- **connlist** : 현재 CUBRID 매니저 서버에 접속 중인 클라이언트들의 리스트를 기록한다.

CUBRID 매니저 클라이언트 실행

CUBRID 매니저는 서버와 클라이언트로 구성되며, CUBRID 매니저 서버가 실행 중일 때에만 CUBRID 매니저 클라이언트를 실행할 수 있다.

CUBRID 매니저 클라이언트는 Java 응용 프로그램으로 JRE 혹은 JDK 버전 1.5 이상에서만 실행되며, 낮은 버전에서 실행할 경우 "지원하지 않는 JRE 버전입니다. CUBRID 매니저를 사용하기 위해서는 JRE 1.5 이상 버전을 사용해야 합니다."라는 오류 메시지가 출력된다.

Windows 환경에서 실행

- [시작] > [모든 프로그램] > [CUBRID] > [CUBRID Manager Client]를 선택한다.



- CUBRID 서비스 트레이를 클릭하고 [Tools] > [CUBRID Manager]를 선택한다.



Linux 환경에서 실행

셸에서 다음 중 하나의 명령을 입력하여 CUBRID 매니저 클라이언트를 실행할 수 있다.

- `%cd $CUBRID/cubridmanager`
- `%. /cubridmanager`

CUBRID 매니저 클라이언트 로그

CUBRID 매니저 클라이언트를 수행 중에 발생한 오류 로그는 **\$CUBRID/cubridmanager/logs**에 **cubridmanager.log**라는 파일로 생성된다. 사용자는 CUBRID 매니저 개선 및 오류 분석 시 해당 오류 로그와 현상을 CUBRID 매니저 개발 사이트 이슈로 등록하여 CUBRID 매니저 개선 활동에 참여할 수 있다.

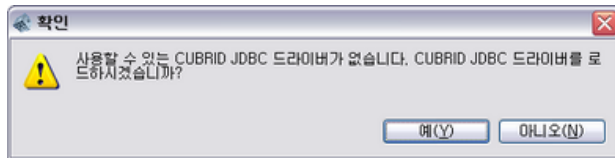
호스트 관리

기본 호스트 정보

CUBRID 매니저를 설치 후 처음 실행하게 되면 호스트 탐색 트리에 "localhost"라는 호스트가 기본으로 설정되어 있다. 현재 CUBRID 매니저 클라이언트가 실행되는 PC에 구동 중인 CUBRID 매니저 서버가 있다는 가정 아래 설정된 것이며, CUBRID 매니저 접속에 대한 기본 정보를 알려주기 위해 제공된다.





- **호스트 이름:** 호스트 이름은 CUBRID 매니저 내에서 관리하고자 하는 호스트를 구분하는 구분자로, 유일한 이름을 부여한다. 내부적으로 관리하는 호스트 구분자는 [호스트 주소 + 연결 포트]이다. 호스트 이름으로는 숫자와 알파벳만 입력할 수 있으며, 공백을 포함할 수 없다. 호스트 이름의 길이는 최소 4자리 최대 32자리이다.
- **호스트 주소:** 호스트 주소는 공백이거나 공백을 포함할 수 없다.
- **연결 포트:** 연결 포트는 1024와 65535 사이의 정수만 입력할 수 있다. 기본 포트 값은 **8001**이며, 접속이 안 될 경우 `$CUBRID/conf/cm.conf` 파일의 `cm_port` 값을 확인한다.
- **사용자 이름:** CUBRID 매니저의 관리자 권한을 가진 사용자는 **admin**이며, 설치 후 처음 사용할 때에는 반드시 **admin** 계정으로 접속해야 한다. 사용자 이름으로는 숫자와 알파벳만 입력할 수 있으며, 공백을 포함할 수 없다. 사용자 이름의 길이는 최소 4자리 최대 32자리이다.
- **비밀번호:** CUBRID 매니저의 관리자 권한을 가진 **admin** 사용자의 최초 비밀번호는 "admin"이다. **admin** 사용자가 최초 연결할 때 비밀번호를 재설정하지 않으면 접속할 수 없다. 이때 재설정된 **admin** 사용자의 비밀번호는 분실하지 않도록 주의한다. 비밀번호는 "admin"으로 재설정할 수 없다. 비밀번호는 공백을 포함할 수 없으며, 비밀번호는 길이는 최소 4자리 최대 32자리이다.
- **드라이버 버전:** CUBRID 매니저에서 사용할 CUBRID JDBC 드라이버를 선택할 수 있다. 디폴트 값은 "Auto Detect"로, 먼저 서버의 버전을 확인한 후 CUBRID 매니저에 등록된 JDBC 드라이버 중 동일한 버전이 있는가를 확인하여 자동으로 연결한다. 연결할 수 있는 JDBC 드라이버가 없으면 다음과 같이 JDBC를 선택하여 등록하라는 확인 메시지가 출력된다.

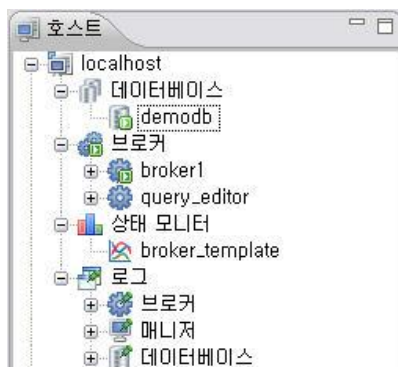


호스트 연결 및 연결 해제

호스트 연결

호스트 탐색 트리에 등록된 호스트를 더블 클릭하면 해당 호스트로 연결된다. 또는 해당 호스트를 마우스 오른쪽 버튼 클릭하여 [연결]을 선택하고, 사용자 이름을 확인한 후 [연결]을 클릭하면 호스트에 연결된다.

연결이 성공하면 호스트 아이콘이 에서 로 변경된다. 연결 후 해당 호스트의 초기 화면은 다음과 같다.



호스트 연결 해제

호스트 탐색 트리에서 호스트를 마우스 오른쪽 버튼 클릭하여 [연결 끊기]를 선택한다.



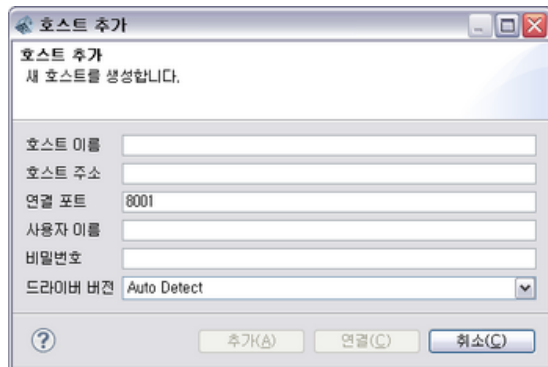
호스트 추가 및 삭제

호스트 추가

호스트를 추가하여, 여러 호스트에 존재하는 데이터베이스를 하나의 매니저에서 관리할 수 있다. 호스트를 추가하려면 다음 중 하나를 실행한다.

- 툴바에서 [호스트 추가 ]를 클릭한다.

- 메뉴에서 [파일] > [호스트 추가]를 선택한다.
- 호스트 탐색 트리에서 마우스 오른쪽 버튼을 클릭하고 [호스트 추가]를 선택한다.



호스트 이름, 호스트 주소, 연결 포트, 사용자 이름 그리고 비밀번호에 대한 자세한 내용은 [기본 호스트 정보](#)를 참조한다.

정보를 입력하고 [추가] 버튼을 클릭하면 호스트 탐색 트리에 추가만 하고 연결은 하지 않는다. 반면, [연결]을 클릭하면 탐색 트리에 추가하고 해당 호스트로 연결한다.

호스트 삭제

호스트 탐색 트리에 등록된 호스트 정보를 삭제한다. 호스트 삭제 시 하부 노드의 모든 정보가 같이 삭제된다. 단, CUBRID 매니저의 클라이언트 구성 정보만 삭제되는 것으로, CUBRID 매니저 서버에 구성된 정보까지 삭제되는 것은 아니다.



서비스 관리

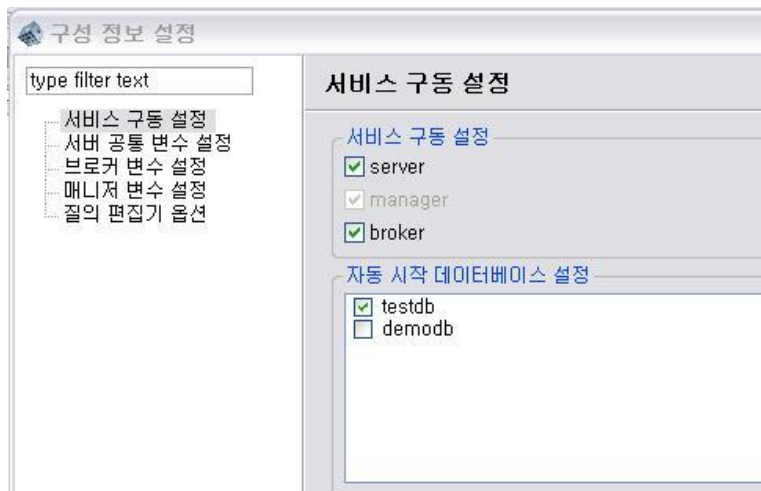
CUBRID 매니저로 CUBRID 관련 서비스를 시작/정지할 수 있다. 이것은 CUBRID 데이터베이스 명령어인 cubrid 명령과 동일한 기능이다. 단, CUBRID 매니저 서버는 CUBRID 매니저 클라이언트를 통해서 구동/정지할 수 없다.



서비스 시작

메뉴에서 [도구] > [서비스 시작]을 선택하면, [서비스 구동 설정]에서 선택한 서비스와 데이터베이스를 시작한다. 시작하는 서비스와 데이터베이스는, 호스트를 마우스 오른쪽 버튼 클릭하고 [속성]을 선택하여 [서비스 구동 설정]에서 선택한다.

[자동 시작 데이터베이스 설정]은 [서비스 구동 설정]에서 [server]를 선택했을 경우에만 활성화된다.





서비스 정지

메뉴에서 [도구] > [서비스 정지]를 선택하면, [서비스 구동 설정]에서 선택한 서비스와 데이터베이스를 정지한다.



데이터베이스 시작/정지

선택한 데이터베이스를 시작 및 정지하려면, 다음 중 한 가지를 실행한다.

- 툴바에서 [시작 ] 또는 [정지 ]를 클릭한다.
- 데이터베이스를 마우스 오른쪽 버튼 클릭하고 [데이터베이스 시작] 또는 [데이터베이스 정지]를 선택한다.
- 메뉴에서 [동작] > [데이터베이스 시작] 또는 [동작] > [데이터베이스 정지]를 선택한다.

브로커 시작/정지

선택한 브로커를 시작 및 정지하려면, 다음 중 한 가지를 실행한다.

- 툴바에서 [시작 ] 또는 [정지 ]를 클릭한다.
- 브로커를 마우스 오른쪽 버튼 클릭하고 [브로커 시작] 또는 [브로커 정지]를 선택한다.
- 메뉴에서 [동작] > [브로커 시작] 또는 [동작] > [브로커 정지]를 선택한다.

사용자 관리

사용자 권한 설정

하나의 데이터베이스에 여러 사용자를 등록하고 사용자 권한을 관리하듯이, 매니저 서버에도 여러 사용자를 등록하고 사용자 권한을 관리할 수 있다. 매니저 사용자 정보는 CUBRID 매니저 서버에서 저장되며, 해당 호스트에 접속하기 위해서는 반드시 사전에 CUBRID 매니저 사용 권한을 취득해야 한다.

관리자 계정인 **admin**과, **admin**이 등록한 일반 사용자만 CUBRID 매니저에 접속할 수 있다. CUBRID 매니저 사용자 관리 기능은 **admin**에게만 허용되며, [도구] > [사용자 관리]를 선택해서 실행할 수 있다.

사용자 관리에서는 각 사용자에게 다음의 권한을 설정할 수 있다.



DB 생성 권한

- **admin** : 새 데이터베이스를 생성할 수 있는 권한을 의미하며, 오직 관리자 계정인 **admin** 사용자에게만 부여된다.
- **none** : 권한이 없음을 의미한다.

브로커 권한

- **admin** : 브로커를 시작/정지할 수 있고 브로커를 추가, 편집, 삭제할 수 있는 관리 권한을 의미한다.
- **monitor** : 브로커의 상태 보기 기능을 통해 브로커의 진행 상황을 모니터링할 수 있는 권한을 의미한다.
- **none** : 권한이 없음을 의미한다.

상태 모니터 권한

- **admin** : 상태 모니터를 수행할 수 있고 추가, 편집, 삭제할 수 있는 관리 권한을 의미한다.
- **monitor** : 상태 모니터를 수행하여 모니터링할 수 있는 권한을 의미한다.
- **none** : 권한이 없음을 의미한다.

사용자 추가, 편집 및 삭제

사용자 추가

사용자 추가는 사용자 계정 정보 및 권한 설정과 데이터베이스 접속 권한 설정으로 이루어진다.

사용자 계정 및 권한 설정

사용자 추가

새 사용자 계정을 생성합니다.

사용자 이름:

비밀번호:

새 비밀번호 확인:

DB 생성 권한: none

브로커 권한: none

상태 모니터 권한: none

< 이전(B) 다음(N) > 완료(F) 취소

- **사용자 이름**: 사용자 이름의 길이는 최소 4자리 최대 32자리이다. 사용자 이름으로는 숫자와 알파벳만 입력할 수 있으며, 공백을 포함할 수 없다. 사용자 이름으로 "admin"을 사용할 수 없으며, 사용자 이름은 해당 호스트 내에서 유일해야 한다.
- **비밀번호**: 비밀번호의 길이는 최소 4바이트 최대 32바이트이며, 공백을 포함할 수 없다. 비밀번호로 "admin"을 사용할 수 없다.

데이터베이스 권한 설정

데이터베이스 이름	연결 권한	DB 계정	브로커 주소	브로커 포트
demodb	Yes	public	10.34.63.89	broker1[33000/ON]
testdb	No			query_editor[30000/ON]
				broker1[33000/ON]

- **연결 권한:** 추가 중인 CUBRID 매니저 사용자가 연결할 수 있는 데이터베이스를 선택한다. [Yes]를 선택한 데이터베이스만 호스트 탐색 트리에 출력된다.
- **DB 계정:** 추가 중인 CUBRID 매니저 사용자가 해당 데이터베이스에 접속할 때 사용하는 데이터베이스 계정 정보를 입력한다. "dba" 또는 "public"과 같은 값을 입력할 수 있다.
- **브로커 주소:** 해당 데이터베이스에 접속하기 위해 사용할 브로커 주소를 입력한다. 디폴트 값은 데이터베이스 서버의 주소와 동일하다. 브로커 서버가 분리되어 있다면 이 값을 변경하여 접속 정보를 부여할 수 있다.
- **브로커 포트:** 추가 중인 CUBRID 매니저 사용자가 해당 데이터베이스에 접속할 때 사용할 브로커 포트를 정의한다. 브로커 포트는 현재 정의되어 있는 브로커 정보를 통해서 선택할 수 있다. 브로커 정보는 "브로커 이름[포트/구동상태]"로 구성되어 있다.

사용자 편집

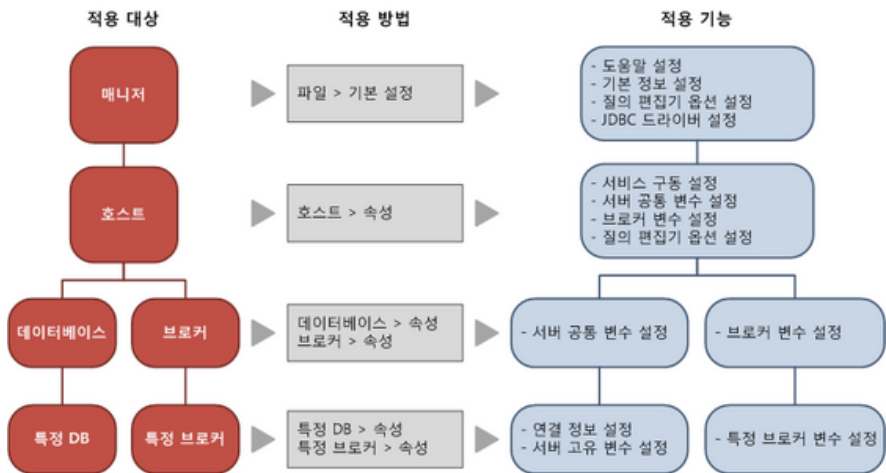
편집할 사용자를 리스트에서 선택한 후, 사용자 추가와 동일한 방법으로 사용자를 편집할 수 있다. 단, **admin** 계정에 대한 편집은 비밀번호만 변경할 수 있으며, 나머지 권한은 **admin** 권한에서 다른 권한으로 변경할 수 없다.

사용자 삭제

삭제할 사용자를 리스트에서 선택한 후 [삭제] 버튼을 클릭하면 사용자를 삭제할 수 있다. 단, **admin** 계정은 삭제할 수 없다.

속성 관리

CUBRID 매니저를 통해서 서비스, 데이터베이스, 브로커, 매니저 서버 그리고 질의 편집기의 구동 환경을 설정할 수 있다. 환경 설정은 각 단계별로 설정할 수 있으며 다음과 같은 구성을 가진다.



도움말 설정

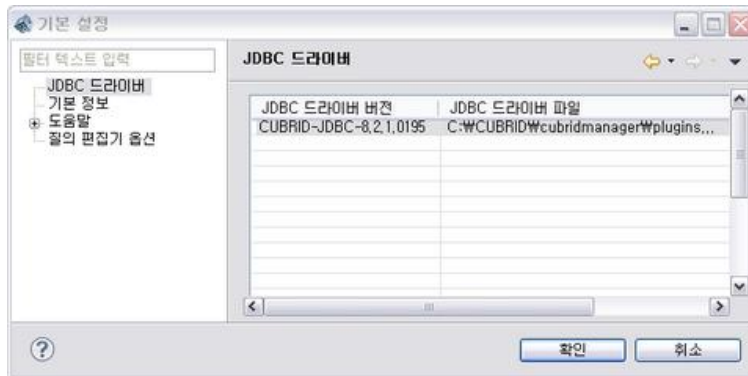
내부 도움말 창을 통해 도움말을 볼지 또는 외부 브라우저로 도움말을 볼지 선택할 수 있으며, Eclipse 플러그인으로 실행했을 때 도움말 열기 모드를 선택할 수 있다.

기본 정보 설정

기본 정보 설정에서는 다음과 같은 내용을 설정할 수 있다.

- CUBRID 매니저 창을 최대 크기로 열거나 또는 이전 크기로 열도록 설정
- 시작할 때 CUBRID 소식 보기 여부 설정
- 탐색 트리에서 정보 창을 열 때 마우스 한 번 클릭으로 열지 또는 두 번 클릭으로 열지 설정(디폴트 값은 두 번 클릭)

JDBC 드라이버 설정



CUBRID 2008 R2.1 버전부터는 다중 JDBC 드라이버를 지원하며, 하위 버전 데이터베이스 서버에 접근하기 위하여 해당 데이터베이스 서버의 JDBC 드라이버를 추가로 설정할 수 있다. CUBRID JDBC 드라이버는 cubridmanager/plugins에 포함되어 배포되며, 원칙적으로 CUBRID 2008 R2.1 버전의 매니저는 CUBRID-JDBC-8.2.1.x 드라이버 버전을 사용하여 해당 버전의 데이터베이스 서버에 접근할 수 있다. 그러나 다중 JDBC 드라이버가 지원되면서 사용자는 CUBRID 2008 R2.1 이상 버전의 매니저를 이용하여 다른 버전의 데이터베이스에 접속하여 질의를 수행할 수 있다. 다른 버전의 데이터베이스에 접속하여 질의를 수행하고자 한다면 [기본 설정] > [JDBC 드라이버]에서 해당 버전의 CUBRID JDBC 드라이버를 추가로 설정해야 한다.

단, 접속하고자 하는 데이터베이스 서버의 버전에 따라 일부 기능이 제한될 수 있다. 예를 들어, 왼쪽 트리 구조에서 관리할 수 있는 기능은 CUBRID 2008 R2.0 이상의 데이터베이스에 대해서는 정상적으로 수행되지만 이전 버전에 대해서는 제한되며, 질의 편집 및 수행 기능은 CUBRID 2008 R1.4 이상의 데이터베이스에 대해서는 정상 수행되지만 이전 버전에 대해서는 제한된다.

질의 편집기 옵션

질의 편집기를 통해서 데이터베이스와 접속할 때 적용되는 매개 변수를 설정한다. 자세한 것은 [질의 편집기 옵션](#)을 참조한다.

서비스 구동 설정

서비스 구동과 관련된 옵션을 설정한다. 데이터베이스 자동 시작을 설정할 수 있다.

서버 공통/고유 변수 설정

데이터베이스 공통 변수를 설정한다. 자세한 것은 [데이터베이스 설정](#)을 참조한다.

특정 브로커 변수 설정


브로커 정보를 설정한다. 자세한 것은 [브로커 메뉴](#)를 참조한다.

연결 정보 설정

연결 정보 설정을 통해 데이터베이스에 접속하는 포트와 문제 세트를 설정할 수 있다.

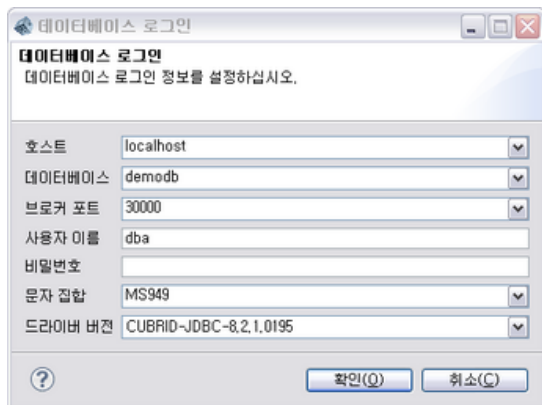
질의 편집기

질의 편집기 구조

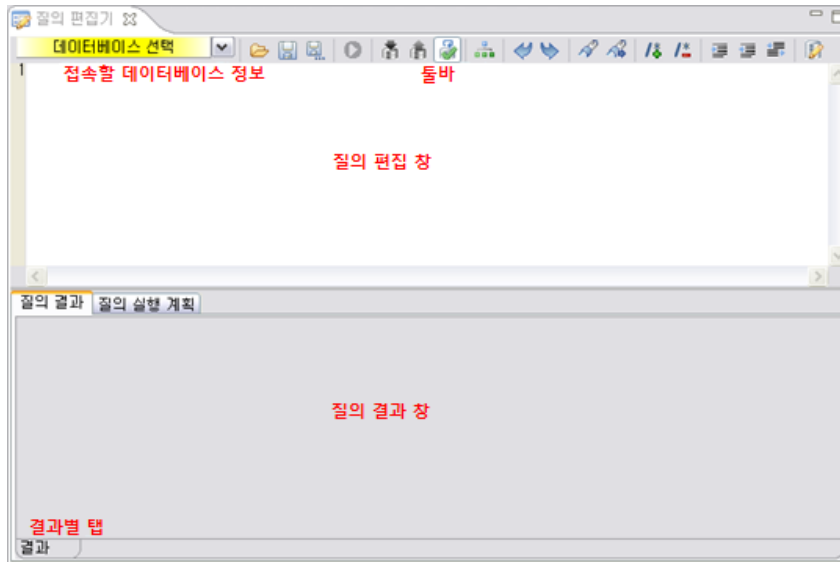
CUBRID 매니저 질의 편집기는 모든 **DML**, **DDL**, **DCL** 문의 수행을 지원하는 질의 도구로서, 보다 쉽고 편리하게 질의를 편집하고 실행할 수 있다. 질의 편집기를 실행하려면 메뉴에서 [도구] > [새 질의 편집기]를 선택하거나, 툴바에서 [새 질의 편집기] 를 클릭한다. 또는 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하여 [새 질의 편집기]를 선택한다.

데이터베이스를 마우스 오른쪽 버튼 클릭하여 [새 질의 편집기]를 선택하면, 해당 데이터베이스에 로그인한 기본 정보로 질의 편집기를 실행한다.

메뉴 또는 툴바에서 [새 질의 편집기]를 선택하면 아래 그림과 같이 사용자가 직접 접속하고자 하는 데이터베이스 로그인 정보, 문자 집합(character set) 및 JDBC 드라이버 버전 정보를 설정하여 로그인할 수 있다.



질의 편집기 화면은 위쪽의 질의 편집 창과 아래쪽의 질의 결과 창으로 구분할 수 있다. 질의 편집 창에서는 수행할 질의를 입력하고 편집하며, 질의 편집기에서 주로 사용하는 기능을 아이콘으로 보여주는 툴바를 포함한다. 질의 결과 창에서는 수행된 질의별 결과를 탭 형식으로 보여주며, 수행한 질의 및 해당 질의의 수행 시간을 확인할 수 있다.

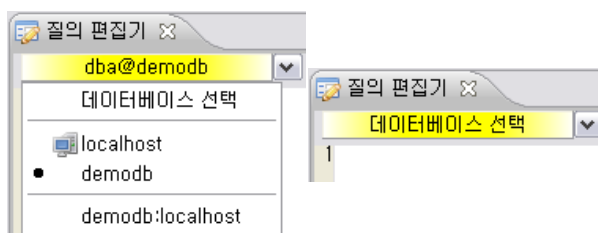


툴바 기능

데이터베이스 선택

하나의 질의 편집기에서 다중 호스트의 다중 데이터베이스를 접속할 수 있도록 지원한다.

현재 접속 중인 데이터베이스를 표시하며, 현재 CUBRID 매니저에서 접속한 다른 호스트의 데이터베이스도 선택하여 접속할 수 있다. 아래 그림과 같이 DBA로 로그인한 경우와 데이터베이스를 선택하지 않은 경우는 노란색으로 강조하여, 현재 어떤 계정으로 로그인하여 작업 중인지 확인하고, 데이터베이스 연결 정보가 없을 때 선택할 수 있도록 지원한다.



만약 현재 접속 중인 데이터베이스의 트랜잭션을 종료하지 않고 다른 데이터베이스를 선택하면 아래와 같이 수행 중인 트랜잭션을 커밋할 것인지 아니면 롤백할 것인지 묻는다.



툴바 기능

질의 편집기 툴바의 기능은 다음과 같으며, 편집 기능은 메뉴의 [편집] 메뉴와 동기화되어 있다. 중요 기능은 단축키를 제공한다.



기능	아이콘	단축키	설명
열기			텍스트 형태의 SQL 문을 열어서 질의 편집창에 보여 준다.
저장			질의 편집창의 내용을 저장한다.
새 이름으로 저장			질의 편집창의 내용을 다른 이름으로 저장한다.
실행		F5	질의 편집창의 모든 질의를 수행한다. 또는 블록으로 선택한 질의만 수행한다. 수행할 수 없는 상태이면 아이콘이 비활성화()된다.
커밋			자동 커밋 모드에서는 비활성화() 상태를 유지하고, 자동 커밋 모드가 아닌 경우에 트랜잭션이 발생하면 활성화()되어 사용자가 명시적으로 커밋을 수행할 수 있다.
롤백			자동 커밋 모드에서는 비활성화() 상태를 유지하고, 자동 커밋 모드가 아닌 경우에 트랜잭션이 발생하면 활성화()되어 사용자가 명시적으로 롤백을 수행할 수 있다.
자동 커밋			질의 편집창에서 수행하는 질의를 자동 커밋한다. 아이콘은 툴 형태로 표시되며, 자동 커밋 설정 상태는 이고 해제 상태는 이다. 단, 해당 설정은 해당 질의 편집기에만 적용되며, 해당 데이터베이스 질의 편집기 옵션의 디폴트 값은 변경되지 않는다.
질의 실행 계획		F6	선택한 질의의 질의 실행 계획을 본다. 자세한 내용은 질의 실행 계획 보기 를 참조한다.
실행 취소		Ctrl+Z	편집 기능으로 편집한 내용을 취소한다.

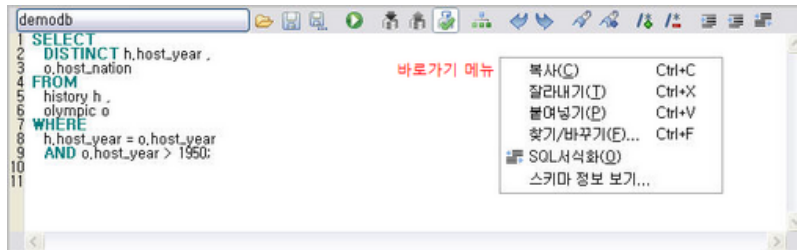
다시 실행		Ctrl+Y	편집 기능으로 실행 취소한 내용을 다시 실행한다.
찾기/바꾸기		Ctrl+F	질의 편집창 내에서 찾고 바꿀 수 있는 기능을 제공한다.
다음 찾기		F3	한번 찾은 내용에 대해 다음 찾기 기능을 제공한다.
주석 추가		Ctrl+/	질의 편집 창에서 사용자가 선택한 영역이나 커서가 있는 라인의 주석을 삽입한다. 주석은 '--'로 삽입된다.
주석 삭제		Ctrl+/	질의 편집 창에서 사용자가 선택한 영역이나 커서가 있는 라인의 주석을 제거한다.
들여 쓰기		Tab	질의 편집 창에서 사용자가 선택한 영역에 들여 쓰기를 한다.
내어 쓰기		Shift+Tab	질의 편집 창에서 사용자가 선택한 영역에 내어 쓰기를 한다.
SQL 서식화		Ctrl+Shift+F	질의 편집 창에서 선택한 SQL 문에 대해 서식화를 수행한다.
OID 정보 보기			[질의 편집기 옵션]의 [OID 정보 보기]를 툴바에서 설정할 수 있다. [OID 정보 보기]를 선택하면, 이후 수행하는 질의부터 질의 결과 창에서 데이터를 직접 수정/삭제할 수 있다. 단, 툴바에서 이 기능을 설정하면 해당 질의 편집기에만 적용되며, 해당 데이터베이스 질의 편집기 옵션의 값은 변경되지 않는다.

질의 편집 창

질의 편집 창에서는 데이터베이스 조작을 위한 질의를 입력하고 편집할 수 있으며, 툴바에서 지원하는 모든 기능을 사용할 수 있다. 질의 편집 창은 구문 자동 완성 기능 및 팝업 메뉴를 통한 편집 및 스키마 정보 보기 기능을 제공한다.

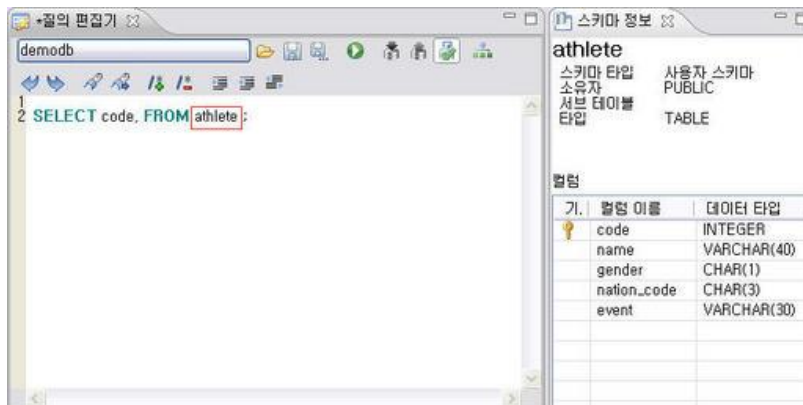
바로가기 메뉴

질의 편집 창에서 마우스 오른쪽 버튼을 클릭하여 복사, 잘라내기, 붙여넣기, 찾기/바꾸기, SQL서식화, 스키마 정보 보기를 실행할 수 있다. 편집 기능은 메뉴의 [편집] 기능과 동기화되어 있다.



스키마 정보 보기

테이블 이름을 선택한 후 [스키마 정보 보기]를 실행하면 질의 편집기 뷰와 스키마 정보 뷰를 한 화면에서 보면서 질의를 작성할 수 있다.



구문 자동 완성

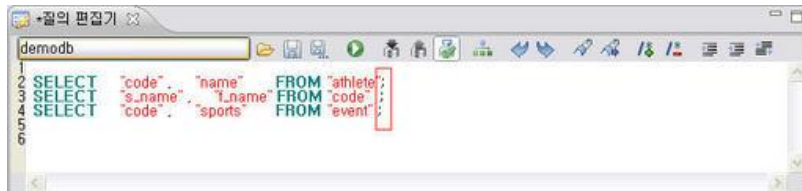
큐브리드 데이터베이스 키워드에 대해서 입력을 하면 자동으로 구문을 찾아 완성시켜 주므로 사용의 편의성을 높이고 사용자의 접근성을 높여 준다.



다중 질의 수행 기능

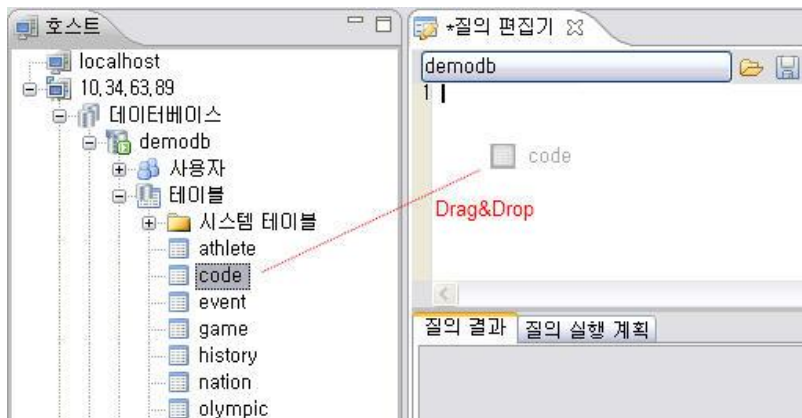
질의문 마지막에 세미콜론(;)을 입력하여, 하나의 질의가 끝나고 다음 질의가 시작됨을 명시한다. 여러 개의 질의가 존재할 경우, 각 질의는 순차적으로 수행되며, 질의별로 질의 결과 창에 결과 탭이 형성된다.

세미콜론으로 구분하지 않고 다중 질의를 수행하면, 첫 번째 질의만 수행되고 이후의 질의는 무시된다.



끌어서 놓기

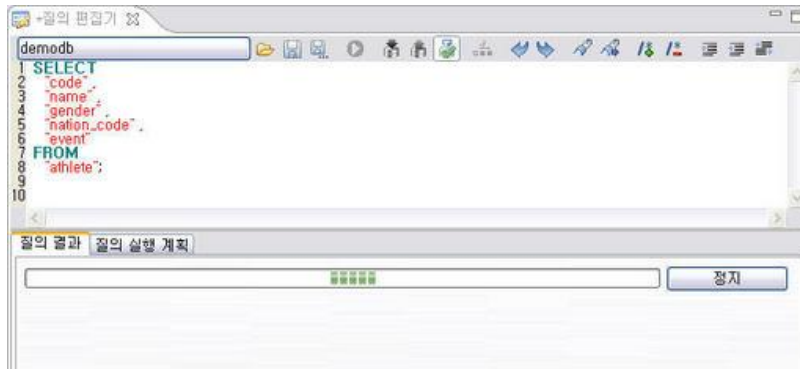
조회하려는 테이블을 호스트 탐색 트리에서 질의 편집기의 편집 창으로 끌어서 놓으면 해당 테이블에 대한 SELECT 질의문이 자동 생성된다.



질의 취소 기능

수행 중인 질의를 중지시킬 수 있는 기능이다. 정지 기능은 두 가지 기능으로 나누어 볼 수 있다.

- **다중 질의 수행 시** : 자동 커밋이 설정되어 있을 때 다중 질의 수행 시 정지 버튼을 클릭하면, 현재 취소되는 질의 이전에 처리한 질의들은 정상적으로 반영되고, 현재 취소된 질의와 이후의 질의는 반영되지 않는다. 자동 커밋이 설정되어 있지 않으면, 수행된 모든 질의를 반영하지 않는다.
- **긴 트랜잭션 수행 시** : 정지 호출 시 CUBRID 매니저 > JDBC > 브로커 > 서버의 각 단계로 질의 취소 명령이 전달되어 최종적으로 데이터베이스 서버에서 해당 질의가 정지되어야 실제 질의가 취소된다. 단, 윈도우 버전의 경우, CUBRID R2.2 이상 버전에서만 질의 취소 기능을 지원한다.



질의 결과 창

질의 결과 창은 수행된 질의의 결과를 보여주는 창으로, 여러 개의 질의가 수행된 경우 질의 결과 창 아래에 질의별 탭으로 구분된다. 각 탭을 선택하여 해당 질의의 수행 결과를 확인할 수 있다.

질의 결과 창은 결과 탐색, 수행 질의 정보, 수행 시간 및 검색 건수를 제공하는 영역으로 구분된다.



SELECT 문을 제외한 다른 SQL문을 수행하거나, 수행한 **SELECT** 문에 오류가 있으면 로그 탭에 수행 질의 정보와 오류 메시지 또는 수행 시간 정보가 출력된다.



질의 결과 창 구조

- **결과 탐색:** 전체 검색 결과에서 [질의 편집기 옵션]의 [결과 창의 페이징 단위 설정]에서 설정한 값만큼 이동하며 탐색한다.
- **수행 질의 정보:** 현재 질의 결과가 어떤 질의를 수행한 결과인지 알려 준다.
- **수행 시간 및 검색 건수:** 현재 질의 결과를 얻는데 소요된 서버의 수행 시간과 전체 검색 건수 정보를 제공한다.

질의 결과 창 기능

질의 결과 창에는 다음 그림의 바로가기 메뉴와 같이 복사, 수정, 삭제, OID 탐색, 상세 보기, 전체 내보내기 그리고 선택된 행 내보내기와 같은 기능이 있다.

NO	code	name	gender	nation_code	event
1	10002	Abanda Patrice	M	CMR	Football
2	10003	Abarca Jose Maria	M	ESP	Water Polo
3	10004	Abay Peter	M	HUN	Fencing
4	10005	Abbagnale Agostino	M	ITA	Rowing
5	10006	Abbagnale Carmine	M	ITA	Rowing
6	10007	Abbagnale Giuseppe	M	ITA	Rowing
7	10008	Abdullaev Mahamad	M	UZB	Boxing
8	10009	Abdullaev Namik	M	AZE	Wrestling
9	10010	Abdur-Rahim Shareef	M	USA	Basketball
10	10011	Abduvaliev Andrei	M	EUN	Athletics
11	10012	Abebe Addis	M	ETH	Athletics
12	10013	Abera Gezahgne	M	ETH	Athletics
13	10014	Abernathy Brent	M	USA	Baseball

- **복사:** 한 행 전체를 복사한다. 한 행의 특정 컬럼 값만 복사하려면 [상세 보기]를 선택한다.
- **수정:** 질의 결과 창에서 데이터를 직접 수정할 수 있다. 수정하고자 하는 데이터를 더블 클릭하여 편집할 수 있는 상태로 바꾼 후 데이터를 수정한다. 단, 이 수정 기능은 해당 호스트의 속성()의 [질의 편집기 옵션] 대화 상자에서 [OID 정보 보기]가 선택되어 있거나, 질의 편집기 툴바의 OID 정보 보기()가 선택되어 있어야만 가능하다.

NO	OID	code	name	gender	nation_code	event
1	@4201111	10002	Abanda Patrice	M	CMR	Football
2	@4201211	10003	Abarca Jose Maria	M	ESP	Water Polo
3	@4201311	10004	Abay Peter	M	HUN	Fencing
4	@4201411	10005	Abbagnale Agostino	M	ITA	Rowing
5	@4201511	10006	Abbagnale Carmine	M	ITA	Rowing
6	@4201611	10007	Abbagnale Giuseppe	M	ITA	Rowing
7	@4201711	10008	Abdullaev Mahamadkadyz	M	UZB	Boxing
8	@4201811	10009	Abdullaev Namik	M	AZE	Wrestling
9	@4201911	10010	Abdur-Rahim Shareef	M	USA	Basketball
10	@42011011	10011	Abduvaliev Andrei	M	EUN	Athletics

- **삭제:** 질의 결과 창에서 데이터를 직접 삭제할 수 있으며, 삭제할 행을 마우스 오른쪽 버튼 클릭하고 [삭제]를 선택하여 수행한다. 단, 삭제 기능은 수정과 마찬가지로 [OID 정보 보기]가 선택되어 있어야만 가능하다.
- **OID 탐색기:** OID 데이터를 마우스 오른쪽 버튼 클릭하면 활성화된다. 해당 OID에 대해 직접 탐색할 수 있는 기능을 제공한다.

- **상세 보기:** 행으로 출력되는 데이터가 많거나 클 경우 한 행의 데이터를 상세히 보기 어렵다. 이 때 해당 행을 마우스 오른쪽 버튼 클릭하고 [상세 보기]를 선택하여 한 행의 모든 데이터를 상세히 볼 수 있다. [OID 정보 보기]가 선택되어 있다면, 상세 보기 창에서 직접 수정할 수도 있다. 또한 컬럼 별로 값을 복사할 수 있다.



- **전체 내보내기:** 결과 창의 모든 데이터를 엑셀 또는 CSV 파일로 내보낸다.
- **선택된 행 내보내기:** 결과 창에서 선택한 행의 데이터만 엑셀 또는 CSV 파일로 내보낸다.

결과 창에서 컬럼명에 마우스 포인터를 올리면 해당 컬럼의 데이터 타입을 알 수 있다.

질의 결과		질의 실행 계획			
NO	code	name	gender	nation_code	event
1	10002	Abanda Patrice	M	CMR	Football
2	10003	Abarca Jose M	M	ESP	Water Polo
3	10004	Abay Peter	M	HUN	Fencing
4	10005	Abbagnale Agostino	M	ITA	Rowing
5	10006	Abbagnale Carmine	M	ITA	Rowing

질의 편집기 옵션

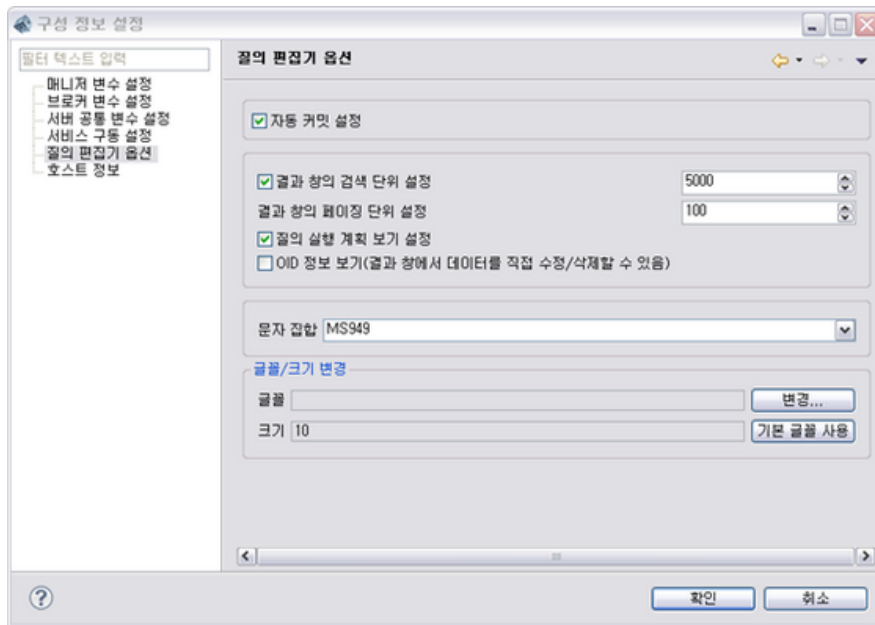
질의 편집기에 대한 옵션은 다음과 같이 설정할 수 있으며, 다음과 같은 적용의 우선 순위를 가진다.

구분	적용	내용
[파일] > [기본 설정] > [질의 옵션]	3 순위	현재 CUBRID 매니저의 기본 정보이다. 각 호스트와 데이터베이스에 적용한 값이 없을 경우 최후에 적용되는 기본 값이다.
[호스트] > [속성] > [질의 편집기 옵션]	2 순위	[파일] > [기본 설정]보다 우선 적용되며, 해당 호스트 내에 공통으로 적용된다.
[데이터베이스] > [속성] > [연결 정보]	1 순위	[호스트] > [설정]보다 우선 적용되며, 설정할 수 있는 정보는 연결 포트와 문자 세트다. 이 정보는 수정 후 여는 질의 편집기부터 적용된다.







[파일] > [기본 설정] > [질의 옵션]

현재 사용하는 CUBRID 매니저에 대한 기본 정보를 설정한다. 설정 항목은 호스트의 설정과 동일하다.

[호스트] > [속성] > [질의 편집기 옵션]



[질의 편집기 옵션] 대화 상자에서는 다음과 같은 옵션을 설정할 수 있다.

- **자동 커밋 설정:** 질의 편집 창에서 질의를 실행한 후 자동으로 커밋을 수행하도록 기본값으로 설정할 수 있다. [자동 커밋 설정]이 선택되어 있어도 질의 편집기의 톨바에서 자동 커밋 기능을 설정() / 해제()할 수 있다.
- **결과 창의 검색 단위 설정:** 질의를 실행한 후, 데이터베이스로부터 한 번에 가져올 결과 행의 개수를 지정한다. 즉, 5,000개로 설정한 상태에서 검색 결과 행 개수가 7,000개라면, 5,000개의 데이터를 가져온 후에 계속 검색 결과를 가져올지 선택할 수 있다.
또한, 다음의 조건을 충족하는 질의를 수행할 경우에는 자동으로 이 설정 값을 기준으로 **ROWNUM**을 추가하여 불필요하게 서버 자원을 많이 사용하는 것을 방지한다.
- **WHERE** 조건 절이 없는 경우
- **GROUP BY**를 사용하지 않는 경우
- **ORDER BY**를 사용하지 않는 경우
- 집합 함수(**SUM**, **COUNT**, **MIN**, **MAX**, **AVG**, **STDDEV**, **VARIANCE**)를 사용하지 않는 경우
- 계층적 질의(Hierarchical Query)를 사용하지 않는 경우
- **결과 창의 페이지징 단위 설정:** 질의 결과 창에서 지정된 데이터 건수 만큼 페이지징(, , , )하여 탐색할 수 있다.


- **질의 실행 계획 보기 설정:** 질의를 실행하기 전후에 해당 질의에 대한 실행 계획을 확인할 수 있다. 이 옵션을 선택하면, 이후 수행하는 질의에 대해 미리 플랜 정보를 생성하게 되므로 질의 수행 시간이 약간의 영향을 받을 수 있다.
- **OID 정보 보기:** 질의 수행 시 OID 정보를 가져온다. 이를 통해 질의 결과 창에서 데이터를 직접 수정/삭제 할 수 있다. 단, 조인 질의와 같이 OID를 가져올 수 없는 경우는 "NONE"이라고 출력한다. 이 옵션을 선택하면, 선택을 안 했을 때보다 질의 수행 시간이 길어질 수 있다.
- **문자 집합:** 해당 데이터베이스에 접속하기 위한 기본 문자 세트를 설정한다. 디폴트 값은 CUBRID 매니저가 수행 중인 시스템(PC)에 설정된 문자 세트이다.
- **글꼴/크기 변경:** 질의 편집기에서 사용하는 글꼴과 글꼴 크기를 변경할 수 있다.


[데이터베이스] > [속성] > [연결 정보]




- **브로커 주소:** 데이터베이스에 접속하기 위해 사용하는 브로커 주소는 기본적으로 DB 서버의 IP로 설정된다. 만약, 브로커 서버가 분리되어 있을 경우, 이 값을 변경하여 해당 브로커를 통해 접속할 수 있다. 단, 브로커 주소는 CUBRID 매니저 **admin** 사용자만 변경할 수 있으며, 그 밖의 사용자는 **admin** 사용자가 지정한 브로커 주소를 통해서만 접속할 수 있다.
- **브로커 포트:** 데이터베이스에 접속하기 위해 사용하는 브로커 포트로 CUBRID 매니저 **admin** 사용자에 의해 부여된 포트가 출력된다. 단, **admin** 사용자만 지정된 포트를 변경하여 사용할 수 있으며, 그 밖의 사용자는 **admin** 사용자가 지정한 브로커 포트를 통해서만 접속할 수 있다.
- **문자 집합:** 해당 데이터베이스에 적용된 문자 세트로 설정할 수 있다. 한 데이터베이스에 하나의 문자 세트만 사용하는 것을 권장한다. 디폴트 값은 CUBRID 매니저가 수행 중인 시스템(PC)에 설정된 문자 세트이다.

질의 실행 계획 보기

질의 편집기 옵션에서 [질의 실행 계획 보기 설정]을 선택하면, 질의 편집기의 툴바에서 [질의 실행 계획 보기] 가 활성화된다.


사용자는 툴바에서 [질의 실행 계획 보기] 를 클릭하거나 <Ctrl+L>을 눌러서, 질의를 실제로 수행하지 않고도 선택된 질의가 어떻게 수행될지에 대한 실행 계획을 확인할 수 있다. 이미 수행된 질의에 대한 실행 계획도 확인할 수 있다.

질의 편집기 하단에는 질의 실행 계획이 항상 보이므로 데이터베이스에 접속하지 않은 상태에서도 실행 계획 이력 파일을 열어서 기존의 실행 계획을 점검해 볼 수 있다.

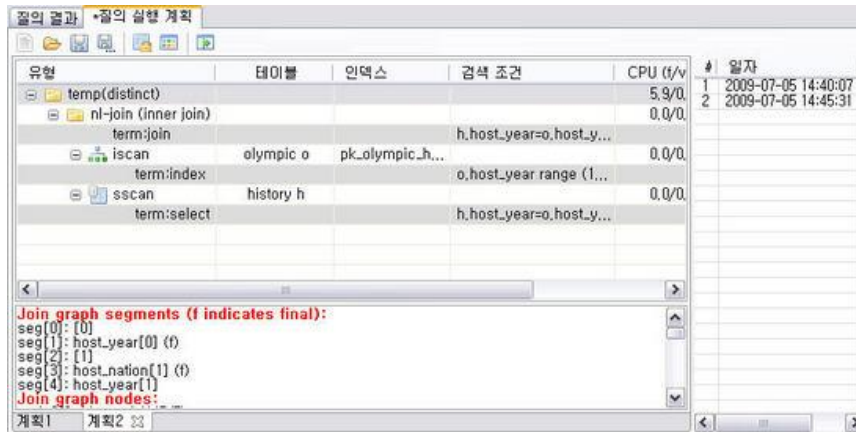
질의 실행 계획 기능은 SQL의 실행 계획을 조회하는 것으로, 일회성이 아닌 실행 계획을 지속적으로 관리하고 조회하기 위한 수집 목적으로 사용한다. 실행 계획을 조회할 때마다 질의 실행 계획 이력에 누적이 되고, 누적된 자료를 .xml 파일로 저장할 수 있다. 저장된 .xml 파일을 열면 당시의 실행 계획과 실행된 SQL을 조회해 볼 수 있다. 일회성 조회가 목적이라면, 툴바에서 [수집 이력 안 남기기] 를 선택하여 이력을 남기지 않고 현재 실행 계획을 임시로 조회할 수 있다.

질의 실행 계획은 툴바, 실행 계획 출력 창, 선택된 실행 계획에 대한 원문 출력 창 그리고 실행 계획 이력 창으로 구성되어 있다.

질의 실행 계획 탭



질을 선택하고 툴바에서 [질의 실행 계획 보기] 를 클릭하면 아래 그림과 같이 [질의 실행 계획] 탭에 질의 수행 계획이 출력된다. [질의 실행 계획] 탭은 해당 질의에 대한 실행 계획을 트리 구조로 요약하여 보여준다.








[질의 실행 계획] 탭은 [질의 결과] 탭 오른쪽에 있으며, 질의 결과를 보다가 [질의 실행 계획] 탭으로 전환할 수 있다.

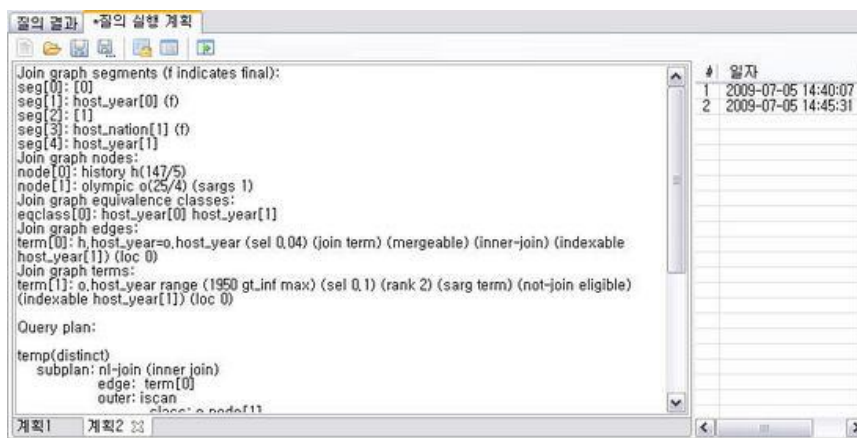




질의 실행 계획 툴바

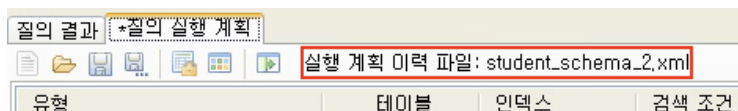
질의 실행 계획 툴바에는 다음과 같은 기능이 있다.

- **새 파일** : 현재까지 수집하고 조회하던 질의 실행 계획 이력을 모두 초기화하고 다시 수집한다. 기존에 하던 작업을 초기화할 때 사용한다.
- **파일 열기** : 기존에 저장된 실행 계획 이력 파일을 불러온다. .xml 파일을 열면 당시의 실행 계획을 조회해 볼 수 있다.

- **질의 실행 계획 저장** , **질의 실행 계획 새 이름으로 저장** : 수집한 실행 계획을 외부 파일로 저장할 수 있다. 파일의 확장자는 **xml**이다.
- **수집 이력 안 남기기** : 질의 편집기에서 질의 실행 계획 보기를 이용하여 실행 계획을 조회하면 항상 실행 계획 이력 창에 이력이 추가된다. 이력에 남기지 않고 임시로 조회하려면 [수집 이력 안 남기기] 를 클릭한다. 실행 계획 출력 창 아래의 탭 이름은 "계획"으로만 출력된다.
- **실행 계획 형식** : 클릭하면 텍스트형  또는 계층형 으로 전환된다. 데이터베이스에서 생성하는 가공되지 않은 실행 계획 소스를 텍스트 형식으로 볼 수 있다.



- **수집 이력창 출력/숨김**  : 오른쪽의 수집 이력 창을 보거나 숨길 수 있다.
- **실행 계획 이력 파일**: 실행 계획 이력을 파일로 저장하였거나 기존에 저장된 파일을 열었을 경우에는 현재 이용중인 실행 이력 파일 이름이 출력된다.



실행 계획 출력 창

실행 계획 출력 창에는 단계별로 실행된 실행계획이 계층형(tree)으로 출력된다.

세로축의 각 항목을 노드(node)라고 부르며 각 노드마다 데이터의 내용은 다르다. 위에서 아래로 내려가면서 본다.

가로축은 항목(item)이라고 부르며 유형, 테이블, 인덱스, 검색조건, CPU I/O 비용, Disk I/O 비용, 전체(ROW/PAGE)으로 구분된다.

유형	테이블	인덱스	검색 조건	CPU (t/v)	IO (t/v)	전체 (r/p)
temp(distinct)				5.9/0.0	7.0/1.0	
nl-join (inner join)				0.0/0.9	1.0/6.0	
term:join			h.host_year=o.host_y...			
iscan	olympic o	pk_olympic_h...		0.0/0.0	1.0/1.0	25/4
term:index			o.host_year range (1...			
sscan	history h			0.0/0.4	0.0/5.0	147/5
term:select			h.host_year=o.host_y...			

- **유형**: sscan, iscan, idx-join 등과 같이 실행 계획상의 각 노드별 질의 최적화기의 실행 방법을 의미한다.
- **테이블**: 해당 노드가 실행될 때 참조하는 테이블(class), 뷰(virtual class) 이름과 별명(alias)이 함께 출력된다.
- **인덱스**: iscan 유형의 경우 사용되는 인덱스명이 출력된다.
- **검색 조건**: 결합(join) 조건, 필터 조건 등이 출력된다. 가독성을 위해 내용은 숨겨져 있으며 +를 클릭하여 하위 노드를 펼치면 상세히 출력된다. 또한 검색 조건별로 별도의 색깔로 볼 수 있다.
- **비용**: 실행 계획에서 출력하는 비용은 CPU, Disk I/O 비용이다. 각각 고정 비용과 가변 비용으로 구분하여 출력된다.
- **전체(r/p)**: 가져올(fetch) 데이터를 만들기 위해 사용되는 테이블의 전체 행의 개수(rows)와 페이지 수(pages)가 출력된다.

질의 실행 계획 출력 창의 아래쪽에는 실행 계획 창에서 선택한 실행 계획의 원문이 출력된다.

```
Join graph segments (f indicates final):
seg[0]: [0]
seg[1]: host_year[0] (f)
seg[2]: [1]
seg[3]: host_nation[1] (f)
seg[4]: host_year[1]
Join graph nodes:
```

실행 계획 이력 창

실행 계획 이력 창에는 실행 계획을 수행할 때마다 이력이 누적되어 출력된다. #은 누적 순번이며 실행 계획 출력 창 아래의 탭 번호와 일치한다. 일자는 실행계획을 실행한 일시이며, 총 비용은 CPU와 Disk I/O의 합이다. 이 실행 계획 이력 창의 항목을 더블 클릭을 하면 해당 실행 계획을 실행 계획 출력 창에서 다시 볼 수 있다.

#	일자	총 비용
1	2009-07-05 14:40:07	3.0
2	2009-07-05 14:45:31	14.0

질의 실행 계획 활용

질의 실행 계획 기능을 활용하면 실행 계획과 해당 테이블의 스키마를 동시에 보면서 분석할 수 있다.

질의 실행 계획 출력 창에서 테이블이 있는 행을 마우스 오른쪽 버튼 클릭하여 [스키마 정보 보기]를 선택하면, 해당 테이블의 스키마 정보를 별도로 열어서 같이 볼 수 있다.

유형	테이블	인덱스	검색 조건	CPU (t/v)	IO (t/v)	전체 (r/p)
temp(distinct)				5.9/0.0	7.0/1.0	
nl-join (inner join)				0.0/0.9	1.0/6.0	
term:join			h.host_year=o.host_y...			
iscan	olymp			0.0/0.0	1.0/1.0	25/4
term:index			h.host_year range (1...			
sscan	history h			0.0/0.4	0.0/5.0	147/5
term:select			h.host_year=o.host_y...			

스키마 정보 창을 CUBRID 매니저 밖으로 끌어서 놓으면 별도의 윈도우로 볼 수도 있다. 이는 다중 모니터를 사용하는 환경에서 편리하게 활용할 수 있다.

demodb

```

1 SELECT
2   DISTINCT h.host_year ,
3   o.host_nation
4 FROM
5   history h
6   olympic o
7 WHERE
8   h.host_year = o.host_year
9   AND o.host_year > 1950;
10
11

```

질의 결과

유형	테이블	인덱스	검색 조건
temp(distinct)			
nl-join (inner join)			
term:join			h.host_year=o,ho
iscan	olympic o	pk_olympic_h...	
term:index			o.host_year rang
sscan	history h		
term:select			h.host_year=o,ho

Join graph segments (f indicates final):
seg[0]: [0]
seg[1]: host_year[0] (f)
seg[2]: [1]
seg[3]: host_nation[1] (f)
seg[4]: host_year[1]
Join graph nodes:
node[0]: history h(147/5)
node[1]: olympic o(25/4)

계획1 계획2

스키마 정보

olympic

스키마 타입 사용자 스키마
소유자 PUBLIC
서브 테이블 TABLE
타입

기.	컬럼 이름	데이터 타입
	host_year	INTEGER
	host_nation	VARCHAR(40)
	host_city	VARCHAR(20)
	opening_date	DATE
	closing_date	DATE
	mascot	VARCHAR(20)
	slogan	VARCHAR(40)
	introduction	VARCHAR(1500)

외래 키

외래키 이름	컬럼 이름
--------	-------

인덱스

인덱스 이름	인덱스 ...	키 컬럼
--------	---------	------

데이터베이스

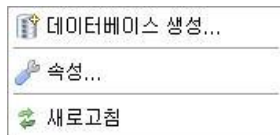
데이터베이스 기능 구조

데이터베이스 관리를 위한 CUBRID 매니저 기능은 크게 전체 관리 기능과 개별 데이터베이스 관리 및 개발 기능으로 나누어 볼 수 있다.

전체 관리 기능

호스트 탐색 트리의 데이터베이스를 마우스 오른쪽 버튼 클릭하여 실행할 수 있다.

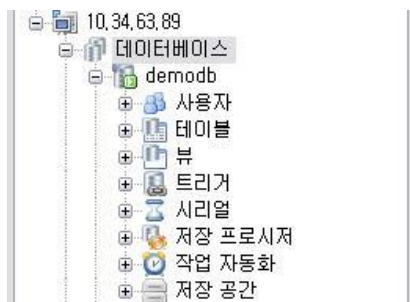
데이터베이스를 생성할 수 있으며, [속성]을 선택하면 해당 호스트 전체에 영향을 주는 데이터베이스 공통 매개 변수를 설정할 수 있다.



개별 데이터베이스 관리 및 개발 기능

개별 데이터베이스 관리 기능은 탐색 트리와 바로가기 메뉴로 제공된다.

데이터베이스 내의 객체와 작업 자동화 관련 기능이 탐색 트리로 제공된다. 탐색 트리로 제공되는 기능은 데이터베이스가 구동 중일 때에만 사용할 수 있다.

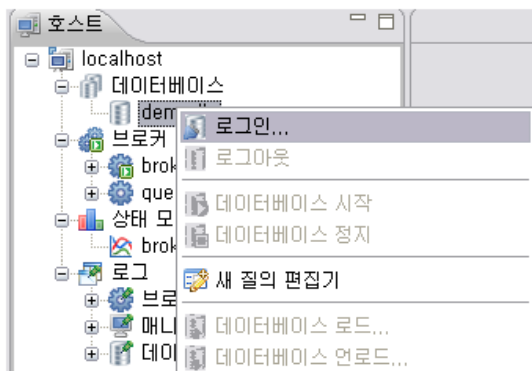


바로가기 메뉴는 운영에 관련된 모든 기능을 제공하며, OID 탐색 기능을 제외한 모든 기능은 데이터베이스가 정지된 상태에서도 사용할 수 있다.

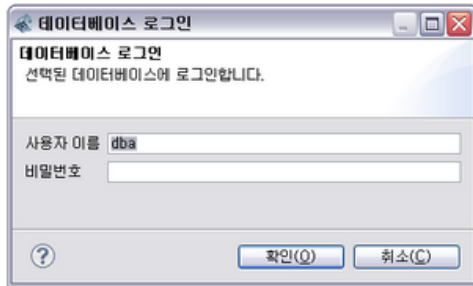


데이터베이스 연결

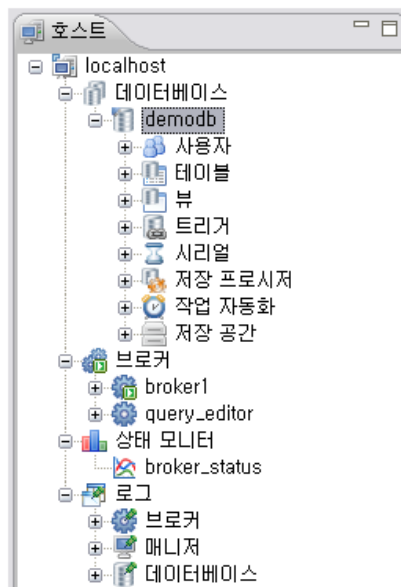
CUBRID 매니저에서 데이터베이스를 구동하려면 데이터베이스에 로그인해야 한다. 호스트 탐색 트리에서 데이터베이스 항목을 마우스 오른쪽 버튼 클릭하고 [로그인]을 선택하거나, 더블 클릭한다.



[데이터베이스 로그인] 창에서 사용자 이름과 비밀번호를 입력한다. 기본 설정된 사용자 이름은 **dba**이며, 비밀번호는 없다.



데이터베이스 로그인 과정에서는 사용자 권한을 검사하고, 로그인에 성공하면 해당 사용자 권한에 맞는 트리 메뉴가 나타난다. 해당 데이터베이스를 구동하기 위해서는 왼쪽 탐색 트리에서 데이터베이스 항목을 마우스 오른쪽 버튼을 클릭하고 [데이터베이스 시작]을 선택한다. 데이터베이스가 구동된 상태에서 [사용자]를 클릭하면 사용자를 추가하거나 기존 사용자의 비밀번호를 변경할 수 있다.



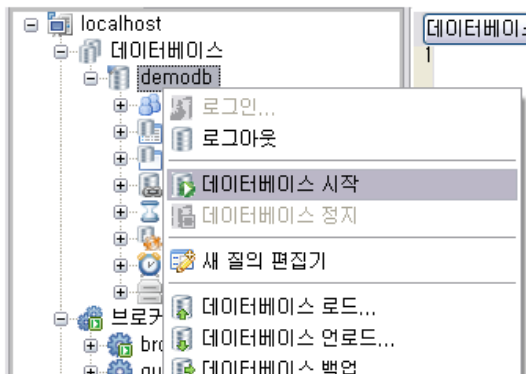
CUBRID 매니저가 정상적으로 구동되지 않는 경우, "서버에 접속할 수 없습니다. CUBRID 매니저 서버의 구동 및 기타 접속 환경을 점검하시기 바랍니다."라는 오류 메시지가 나타난다. 이 때, 사용자는 다음 사항을 확인하여 문제를 해결할 수 있다.

- CUBRID 매니저 서버가 구동 중인지 확인한다.

- CUBRID 매니저 서버의 환경 설정 파일을 열고 **cm_port** 파라미터의 값이 등록된 통신 포트와 같은지 확인한다. [CUBRID 매니저 서버 환경 설정](#)을 참고한다.
- 매니저 클라이언트가 구동되는 현재 시스템에 방화벽이 설정되어 있다면, 매니저 클라이언트 접속을 위한 통신 포트(**cm_port**, **cm_port+1**)를 모두 연다. 예를 들어, **cm_port**가 8001이면, 8001 포트와 8002 포트를 열어야 한다.
- "Cannot execute the current operation because the previous operation is already running."이라는 메시지가 나타나면, 서버가 이미 동일한 작업을 수행하고 있는 것이다. 이 메시지가 나타나면 해당 작업을 재시도한다.

데이터베이스 시작

CUBRID 매니저에서 질의를 실행하려면 데이터베이스를 시작해야 한다. 데이터베이스를 선택한 후, 툴바에서 [시작]을 클릭한다.



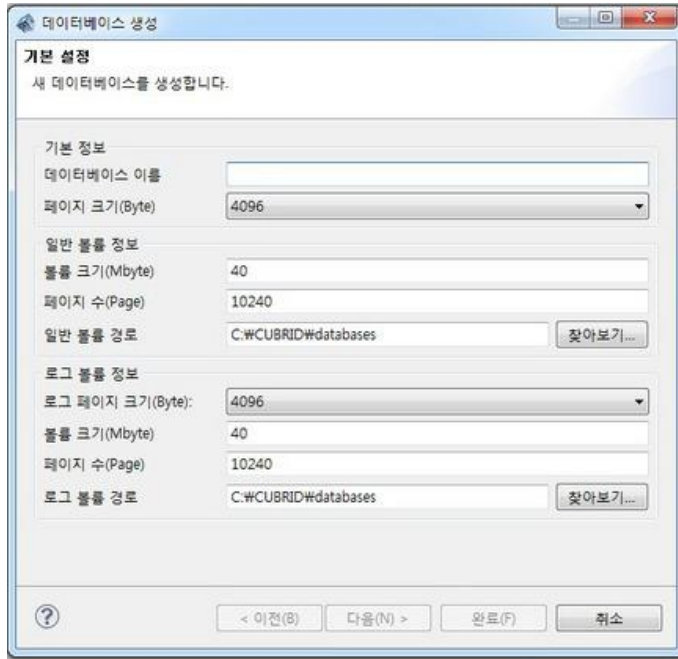
데이터베이스 생성

데이터베이스 생성은 바로가기 메뉴 또는 상단 툴바에서 [데이터베이스 생성]을 클릭하여 수행 할 수 있다. 데이터베이스 생성 마법사는 총 5단계로 구성되어 있다.

CUBRID 매니저 **admin** 계정만 데이터베이스를 생성할 수 있으며, 이에 대한 자세한 사항은 [사용자 권한 설정](#)을 참조한다.

1 단계: 기본 설정

데이터베이스 이름, 일반 볼륨 정보 로그 볼륨 정보와 같은 기본 정보를 설정한다.



- **데이터베이스 이름:** 데이터베이스 이름은 해당 호스트 내에서 유일해야 하며, 같은 이름이 있으면 경고 메시지를 출력한다. 데이터베이스 이름은 다음과 같은 규칙을 지켜야 한다.
- 데이터베이스 이름으로는 숫자와 알파벳만 입력할 수 있으며, 최대 길이는 16자이다.
- Linux (Unix) 파일명으로 사용할 수 없는 글자(공백, *, &, %, \$, |, ^, /, ~, ₩)와 특수 문자는 사용할 수 없다.
- "."과 "-"은 데이터베이스 이름으로 사용할 수 없다.
- **페이지 크기:** 1024, 2048, 4096, 8192, 16384 중에서 선택한다. 디폴트 값은 **4096**이다. 데이터베이스를 생성한 후에는 변경할 수 없으므로 생성하는 데이터베이스의 사용 목적에 따라 조정한다. 특별한 경우가 아니면 디폴트 값을 사용하는 것을 권장한다.
- **일반 볼륨 정보:** 일반 볼륨(generic volume)의 볼륨 크기를 Mbyte 단위로 입력하면, 생성하고자 하는 볼륨의 페이지 수를 자동으로 연산하여 보여 준다. 일반 볼륨 경로의 디폴트 값은 데이터베이스 위치 정보 파일(\$CUBRID_DATABASES/databases.txt)에 설정된 값이다. CUBRID 매니저와 동일한 서버(PC)에 데이터베이스를 생성한다면, [찾아보기]를 클릭해서 디렉토리를 선택할 수 있다.
- **로그 볼륨 정보:** 로그 볼륨(log volume)의 볼륨 크기 및 로그 볼륨 경로를 입력한다. 로그 볼륨 경로의 디폴트 값은 일반 볼륨 경로와 동일한 경로이다.

2 단계: 추가 볼륨 설정

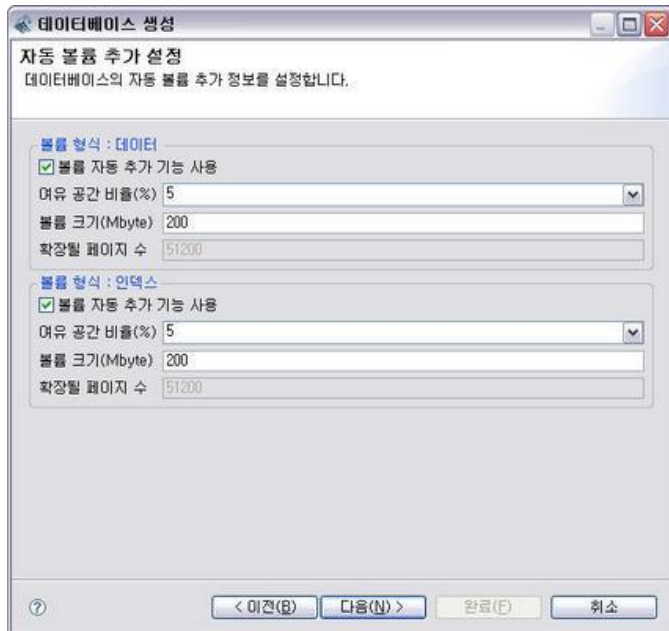
볼륨 형식(generic, data, index, temp)별로 추가 볼륨에 관한 정보를 입력한다. 단, 추가 볼륨 설정에서는 볼륨 형식별 자동 볼륨 추가 설정을 위해 반드시 데이터 볼륨과 인덱스 볼륨을 설정해야 한다.



- **추가 볼륨 정보:** 데이터베이스 생성 시에 추가 용량이 예측되는 경우, 이 단계에서 추가될 볼륨에 관한 정보를 설정한다. 새로 추가할 볼륨의 이름과 경로, 볼륨 타입 및 볼륨 크기를 입력하고, [볼륨 추가]를 클릭하면 입력된 설정 정보에 따라 볼륨이 새로 추가된다. 볼륨 형식은 목적에 따라 data, index, temp, generic 중 하나를 선택할 수 있다. 추가 볼륨에 데이터 볼륨과 인덱스 볼륨은 반드시 추가로 입력한다.
- **볼륨 추가/볼륨 삭제:** [볼륨 추가]를 클릭하면 입력한 설정 정보에 따라 추가될 볼륨이 리스트에 표시된다. 또한, 볼륨 리스트에서 볼륨을 선택한 후, [볼륨 삭제] 버튼을 클릭하면 추가 볼륨 리스트에서 삭제되어 생성되지 않는다.
- **추가 볼륨 리스트:** 데이터베이스 생성 단계에서 추가될 볼륨의 정보를 나열한다.

3 단계: 자동 볼륨 추가 설정

볼륨 형식(data, index)별로 남은 공간이 부족할 경우 자동으로 해당 볼륨을 추가할 수 있는 정보를 설정한다. 데이터 볼륨 또는 인덱스 볼륨에 대해서만 자동 추가 기능을 사용할 수 있다.



- **볼륨 자동 추가 기능 사용:** 선택하면 해당 형식의 볼륨에 대해 자동 추가 기능을 사용한다.
- **여유 공간 비율:** [여유 공간 비율]에 지정한 값과 남은 볼륨이 같을 때, 자동으로 볼륨을 추가한다. 예를 들어, 여유 공간 비율이 5%이면, 기존 볼륨의 여유 공간이 5% 남았을 때, 자동으로 데이터 볼륨을 추가한다. 최소 값은 5이고 최대 값은 30이다.
- **볼륨 크기:** 자동으로 추가할 볼륨의 크기를 설정한다.

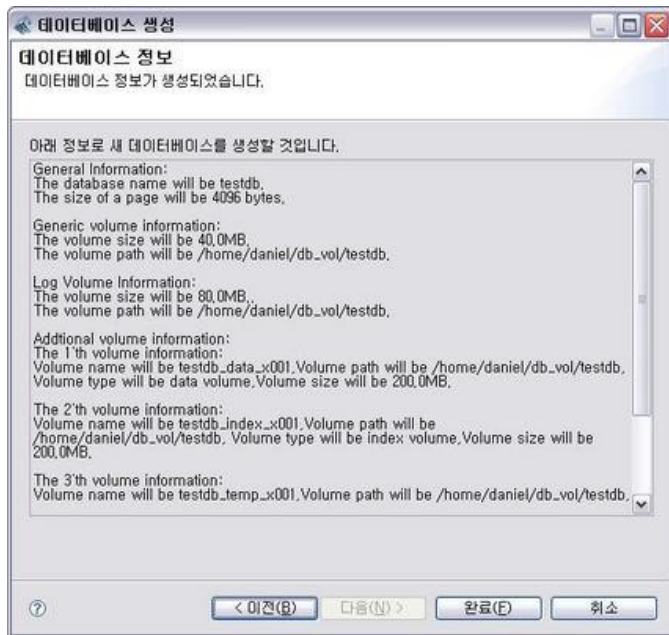
4 단계: DBA 비밀번호 설정

새로 생성하는 데이터베이스의 DBA 계정의 비밀번호를 설정한다. 비밀번호는 공백을 포함할 수 없고 4자리 이상이어야 한다.



5 단계: 생성 정보 확인

4단계까지 설정한 정보를 확인 한 후 데이터베이스를 생성한다. 수정 사항이 있으면 [이전]을 클릭하면 이전 단계로 이동하여 수정할 수 있다.



데이터베이스 설정

데이터베이스 서버 매개 변수를 설정하는 방법은 다음과 같다.

- 호스트 내의 모든 데이터베이스에 공통으로 적용되는 데이터베이스 서버 매개 변수를 설정하려면, 해당 호스트 또는 [데이터베이스]를 마우스 오른쪽 버튼 클릭하고 [속성]을 선택한 후 [서버 공통 변수 설정]에서 설정한다.
- 특정 데이터베이스의 매개 변수를 설정하려면, 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하고 [속성]을 선택한 후, [서버 고유 변수 설정]에서 설정한다.

서버 공통 변수 설정

\$CUBRID/conf/cubrid.conf 파일의 **[common]** 섹션에 대한 공통 매개 변수를 설정한다. 설치 시 기본으로 포함된 매개 변수를 설정하는 [기본 설정] 탭과 나머지 모든 매개 변수를 설정하는 [고급 설정] 탭으로 구성되어 있다. 각 매개 변수에 대한 자세한 설명은 [데이터베이스 서버 설정](#)을 참조한다.

- 기본 설정

서버 공통 변수 설정

기본 설정 고급 설정

data_buffer_pages: 25000

sort_buffer_pages: 16

log_buffer_pages: 50

lock_escalation: 100000

lock_timeout_in_secs: -1

deadlock_detection_interval_in_secs: 1

checkpoint_interval_in_mins: 720

isolation_level: TRAN_REP_CLASS_UNCOMMIT_INSTANCE

Cubrid Port ID: 1523

max_clients: 50

☒ auto_restart_server

☐ replication

☐ java_stored_procedure

• 고급 설정

서버 공통 변수 설정

기본 설정 고급 설정

매개 변수 이름	매개 변수 형식	값 형식	매개 변수 값
async_commit	server	bool(yes no)	no
backup_volume_max_size_bytes	server	int(v>=32*1024)	-1
block_ddl_statement	client	bool(yes no)	no
block_nowhere_statement	client	bool(yes no)	no
call_stack_dump_activation_list	client.server	string	
call_stack_dump_deactivation_list	client.server	string	
call_stack_dump_on_error	client.server	bool(yes no)	no
compactdb_page_reclaim_only	utility only	int	0
compat_numeric_division_scale	client.server	bool(yes no)	no
compat_primary_key	client	bool(yes no)	no
csql_history_num	client	int(v>=1&&v<=200)	50
db_hosts	client	string	
dont_reuse_heap_file	server	bool(yes no)	no
error_log	client.server	string	cubrid.err
file_lock	server	bool(yes no)	yes
garbage_collection	client	bool(yes no)	no
group_commit_interval_in_msecs	server	int(v>=0)	0
hostvar_late_binding	client	bool(yes no)	no
index_scan_in_oid_order	client	bool(yes no)	no
index_scan_old_buffer_pages	server	int(v>=1&&v<=16)	4
insert_execution_mode	client	int(v>=1&&v<=7)	1
intl_mbs_support	client	bool(yes no)	no
lock_timeout_message_type	server	bool(yes no)	0
max_plan_cache_entries	client.server	int(v>=0&&v<=2)	1000

서버 공통 변수 설정에서 설정하는 모든 값은 편집 시 바로 유효성 검사를 수행하여 사용자의 실수를 최소화한다.

기본 설정 고급 설정

매개 변수 이름 매개 변수 형식 값 형식 매개 변수 값

async_commit server bool(yes|no) no

backup_volume_max_size_bytes server int(v>=32*1024) -1

오류

async_commit 매개 변수는 "yes" 또는 "no"만 올 수 있습니다.

확인(O)

서버 고유 변수 설정

서버 고유 변수 설정은 서버 공통 변수 설정과 동일한 기능을 제공한다. 단, 설정한 값은 해당 데이터베이스에만 적용 되는 것으로 \$CUBRID/conf/cubrid.conf 파일의 [@DBNAME] 섹션에만 추가된다.

사용자

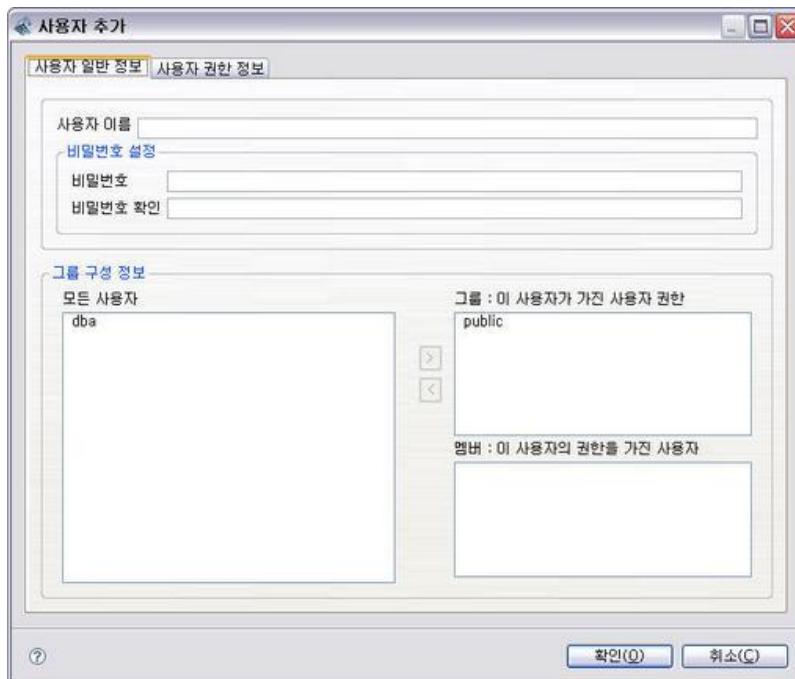
데이터베이스에 로그인한 후 탐색 트리에서 [사용자]를 마우스 오른쪽 클릭하면, 사용자를 추가, 편집, 삭제할 수 있다. [사용자]의 하부 노드에는 현재 데이터베이스의 모든 사용자가 출력된다.



사용자 추가

탐색 트리에서 [사용자]를 마우스 오른쪽 클릭하여 [사용자 추가]를 선택하거나, 톨바 메뉴에서 [사용자 추가] 아이콘을 클릭하여 사용자를 추가할 수 있다.

사용자 일반 정보



- **사용자 이름/비밀번호:** 추가할 사용자의 이름, 비밀번호를 입력한다. 사용자 이름의 길이는 최대 32자리이다.
- **모든 사용자:** 추가할 사용자의 그룹으로 선택할 수 있는 사용자 목록을 보여준다.
- **이 사용자가 가진 권한:** 추가할 사용자가 가입한 그룹 목록을 보여준다. 화살표 버튼을 통해 [모든 사용자] 리스트로부터 추가할 사용자가 가입할 그룹을 지정할 수 있다. 단, **dba**와 **public** 계정은 사용자 권한을 조정할 수 없다.
- **이 사용자의 권한을 가진 사용자:** 사용자의 권한을 가진 사용자 목록을 보여준다.

사용자 권한 정보

[사용자 권한 정보] 탭에서는 테이블 별로 권한을 부여하고 철회할 수 있다. 권한 부여 대상 리스트는 각 컬럼별로 정렬하여 볼 수 있으며, 부여 대상 테이블을 다중 선택하여 권한을 부여할 수 있다.

The screenshot shows a '사용자 추가' (Add User) dialog box. The '사용자 권한 정보' (User Privilege Information) tab is selected. It contains two main tables.

The first table lists existing users:

테이블 이름	스키마 형식	소유자	수퍼 테이블	형식
htable	사용자 스키마	AUSER		테이블
itable	사용자 스키마	AUSER		테이블
jtable	사용자 스키마	AUSER		테이블
htable	사용자 스키마	AUSER		테이블
itable	사용자 스키마	AUSER		테이블
ntable	사용자 스키마	AUSER		테이블
ntable	사용자 스키마	AUSER		테이블

Below this table are buttons for '↓ 권한 부여' (Assign Privilege) and '↑ 권한 제거' (Remove Privilege).

The second table is for assigning permissions to a new user:

테이블 이름	<input type="checkbox"/> SELECT	<input checked="" type="checkbox"/> INSERT	<input type="checkbox"/> UPDATE	<input type="checkbox"/> DELETE	<input type="checkbox"/> ALTER
atable	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
btable	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ctable	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
dtable	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
etable	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
ftable	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
gtable	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

At the bottom of the dialog are buttons for '확인(O)' (OK) and '취소(C)' (Cancel).

사용자 편집

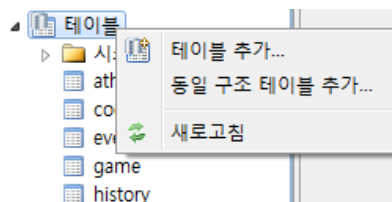
사용자 추가에서 입력한 내용을 편집할 수 있다. 단, 사용자 이름은 변경할 수 없다.

사용자 삭제

선택한 데이터베이스 사용자 계정을 삭제한다.

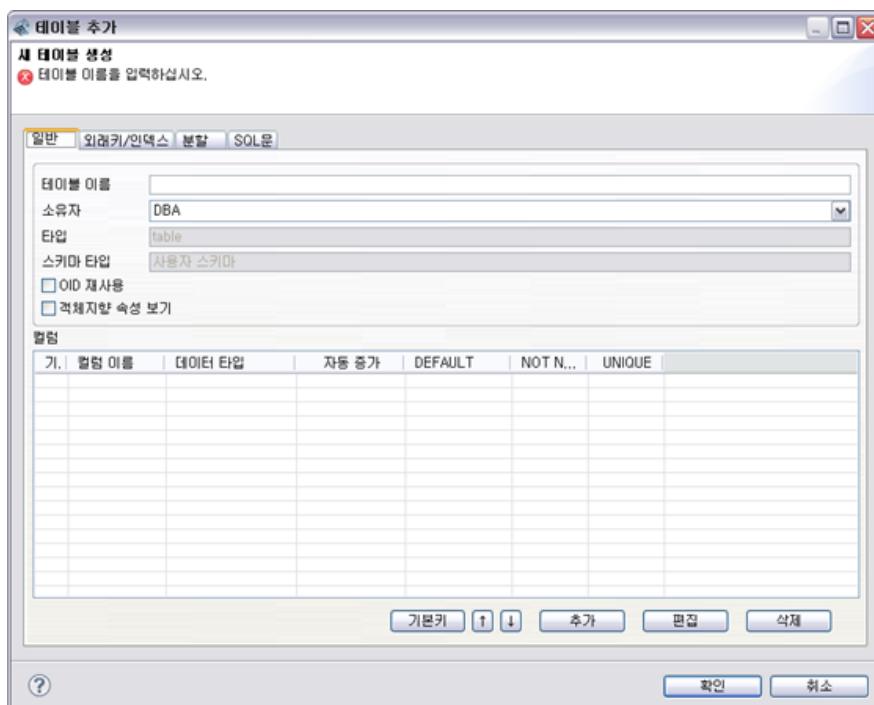
테이블

데이터베이스에 로그인하면, 접근할 수 있는 테이블과 시스템 테이블을 탐색 트리에서 볼 수 있다. 탐색 트리에서 [테이블]을 마우스 오른쪽 버튼 클릭하면, [테이블 추가], [동일 구조 테이블 추가], [새로고침] 메뉴가 나타난다.



테이블 추가

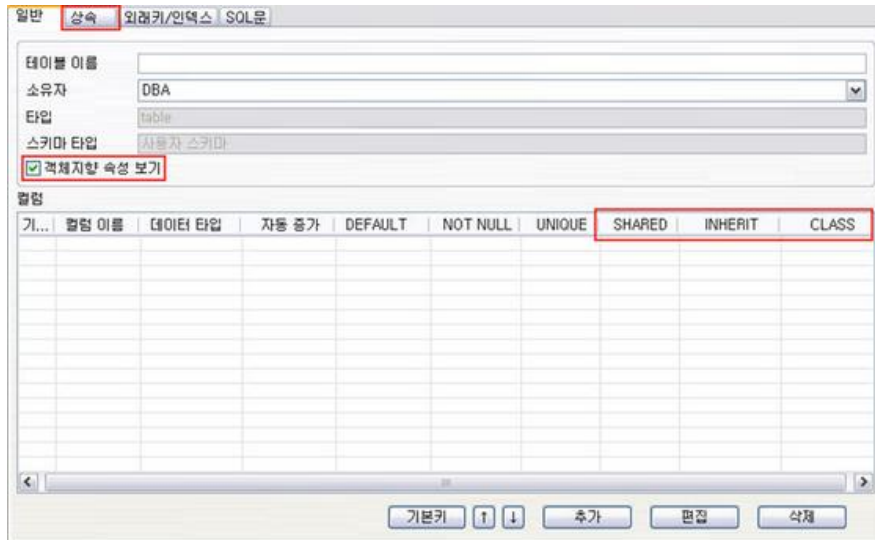
탐색 트리의 [테이블]을 마우스 오른쪽 버튼 클릭해서 [테이블 추가]를 선택하거나, 톨바에서 [테이블 추가] 버튼을 클릭하면, 새 테이블을 생성할 수 있는 마법사가 나타난다.



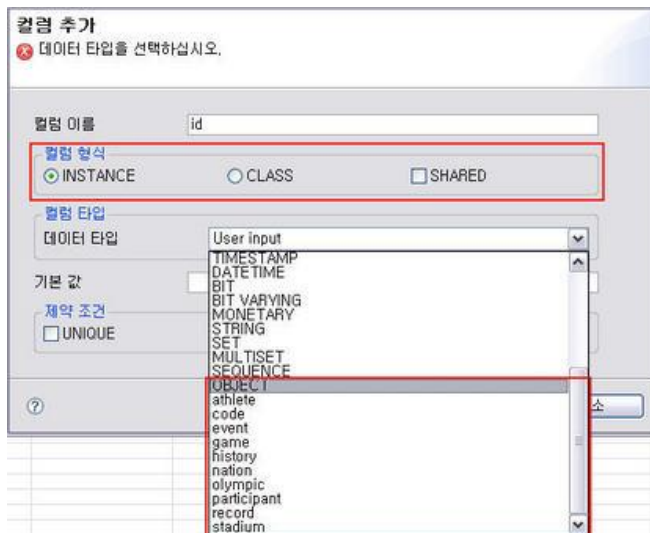
- **[일반] 탭** : 추가할 테이블 이름을 정의하고 컬럼을 추가, 편집, 삭제할 수 있다. 정의된 컬럼에 대해 기본키(PK)를 설정할 수 있으며, 테이블을 생성하기 전에는 [↑], [↓]를 클릭해서 선택한 컬럼의 위치를 조정할 수 있다.
추가할 컬럼 이름, 컬럼 타입, 기본 값 및 컬럼 제약 조건을 컬럼 추가 마법사로 설정할 수 있다. 문법에 맞지 않는 내용이 입력되면 경고 메시지를 출력하거나 문법에 맞지 않는 내용은 선택할 수 없도록 비활성화한다.
- **OID 재사용** : REUSE_OID 테이블 옵션이 적용되어 테이블을 생성하며, 이러한 테이블에 대해서는 OID를 이용한 기능은 제한된다. OID 재사용 테이블에 대한 자세한 내용은 [테이블 옵션\(REUSE_OID\)](#)을 참고한다.
- **[외래키/인덱스] 탭** : 외래키와 인덱스를 설정할 수 있다.
- **외래키 추가** : 외래키 이름, 참조 테이블 이름, 참조 테이블의 기본키를 외래키 추가 마법사로 설정할 수 있으며, 참조 관계 무결성을 유지하기 위한 트리거 동작을 설정할 수 있다. **ON UPDATE, ON DELETE, ON CACHE OBJECT** 옵션을 제공한다.
- **인덱스 추가** : 인덱스 이름, 인덱스 타입, 인덱스를 설정하고자 하는 컬럼을 인덱스 제약 조건 추가 마법사로 설정할 수 있다. 오름차순(asc) 정렬과 내림차순(desc) 정렬은 지원되는 범위 내에서 선택할 수 있다. **REVERSE** 인덱스는 내림차순 정렬만 지원된다.
- **[분할] 탭** : 해당 테이블에 대한 분할 설정 및 변경을 지원한다.
- **분할 추가** : 분할 추가 마법사로 분할 형식과 분할 표현식을 설정할 수 있다. **RANGE, LIST, HASH** 분할을 지원한다.
- **분할 편집/삭제** : 분할 관리 기능 중 분할 편집 및 삭제를 수행할 수 있다.
- **[SQL문] 탭** : [일반] 탭과 [외래키/인덱스] 탭에서 설정한 대로 생성된 SQL 문을 확인하고 복사할 수 있다.

객체 지향 테이블 추가

객체 지향 속성을 가지는 테이블을 추가하려면 [객체 지향 속성 보기]를 선택한다. [객체 지향 속성 보기]를 선택하면, 상속 탭이 추가된다.



- **[일반] 탭:** [객체 지향 속성 보기]를 선택하면, 컬럼 항목으로 SHARED, INHERIT, CLASS 정보가 추가된다. 컬럼 추가 및 편집 시에는 아래 그림과 같이 컬럼 형식을 선택할 수 있고, 데이터 타입으로 OBJECT와 데이터베이스 내에 존재하는 테이블을 선택할 수 있다.



- **[상속] 탭:** 상속하고자 하는 수퍼 테이블을 정의할 수 있으며, 컬럼 이름이 충돌하면 조정할 수 있는 기능을 제공한다.



전체 조회

탐색 트리에서 각 테이블을 마우스 오른쪽 버튼 클릭하여 [전체 조회]를 선택하면 수행할 수 있다. 또는 [질의 편집기]가 열려 있을 때 질의 편집기의 편집 창 또는 결과 창에 해당 테이블을 끌어서 놓아서 수행할 수 있다. 수행하면 새로운 질의 편집기가 열리면서 전체 데이터를 조회한다.

개수 조회

해당 테이블의 전체 데이터 개수를 조회한다. 다음 구문과 같은 기능을 수행한다.

```
SELECT COUNT(*) FROM table_name
```

DELETE ALL

해당 테이블의 전체 데이터를 삭제한다. 다음 구문과 같은 기능을 수행한다.

```
DELETE FROM table_name
```

TRUNCATE TABLE

해당 테이블의 전체 데이터를 삭제한다. 테이블에 정의된 모든 인덱스와 제약 조건을 포함하여 모든 레코드를 한 번에 삭제하기 때문에, [DELETE ALL]보다 빠르다. [TRUNCATE TABLE]을 사용해서 삭제하면 **ON DELETE** 트리거가 활성화되지 않는다. 다음 구문과 같은 기능을 수행한다.

```
TRUNCATE TABLE table_name
```

데이터 입력

해당 테이블의 타입과 제약 조건 등을 보면서 각 컬럼별로 값을 입력할 수 있다.

athlete 테이블 데이터 입력

데이터 입력
컬럼에 대한 값을 입력하십시오.

컬럼	도메인	제약 조건	값
code	INTEGER	Auto Increment, Unique	20001
name	VARCHAR(40)	Not Null	daniel
gender	CHAR(1)		M
nation_code	CHAR(3)		KOR
event	VARCHAR(30)		test

```
insert into "athlete" ("code", "name", "gender", "nation_code", "event") values (20001, 'daniel', 'M', 'KOR', 'test')
// 1 행이 입력되었습니다.
```

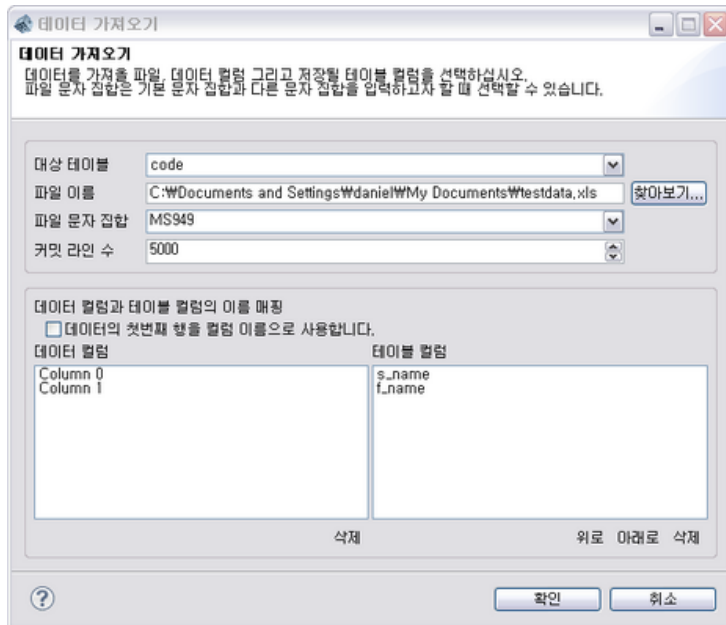
총 1 건이 입력됨

입력 지우기 닫기

- 여러 인스턴스를 추가할 때, 각 질의와 결과 사이에 줄바꿈 문자를 추가하여 구분할 수 있다.
- 각 필드의 값을 입력할 때 <Enter> 키를 누르면 다음 필드로 커서가 이동한다.
- [지우기] 버튼을 클릭하면 입력 상자의 값, 수행 이력까지 초기화한다.
- 수행 이력창은 편집할 수 없다.
- **DATE, TIME, TIMESTAMP, DATETIME** 데이터 타입에 대해서는 타입 별로 시스템 데이터를 입력할 수 있다. 예를 들어, **DATE** 타입 같은 경우 **SYSDATE, SYS_DATE, CURRENT_DATE, DATE** '2009-07-05'와 같은 데이터를 입력할 수 있다.

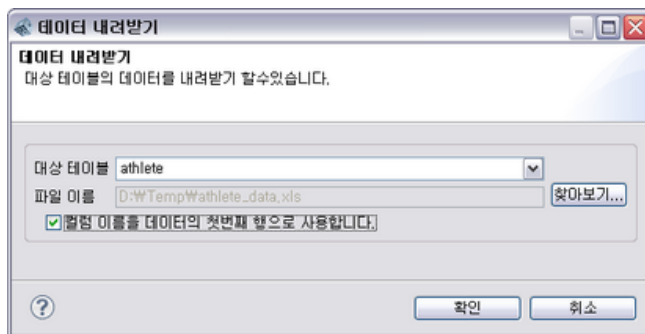
데이터 가져오기

엑셀 또는 CSV 파일의 데이터를 가져와 데이터베이스에 저장할 수 있다. 단, 데이터를 가져올 파일의 문자 세트가 데이터베이스의 문자 세트와 다를 경우, 아래처럼 파일의 문자 세트를 설정한 후 가져오기를 수행할 수 있다.



데이터 내려받기

데이터 내려받기 기능을 사용해서 해당 테이블의 모든 데이터를 엑셀, CSV, SQL, 큐브리드 로드 형식으로 받을 수 있다.



테이블 삭제

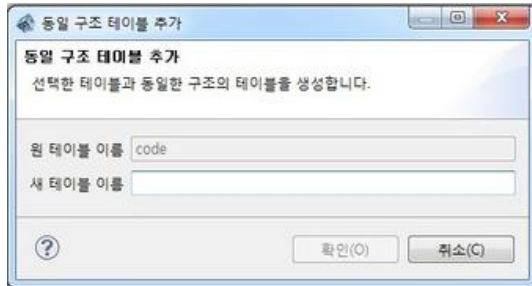
해당 테이블을 삭제한다. **DROP TABLE** 문과 동일하다.

테이블 이름 변경

현재 테이블의 이름을 새로운 이름으로 변경한다. **RENAME TABLE** 문과 동일하다.

동일 구조 테이블 추가

이미 존재하는 테이블과 동일한 스키마 구조를 가지는 빈 테이블을 생성하며, **CREATE TABLE LIKE** 문이 수행된 것과 같다. 자세한 내용은 [CREATE TABLE LIKE](#)를 참고한다.

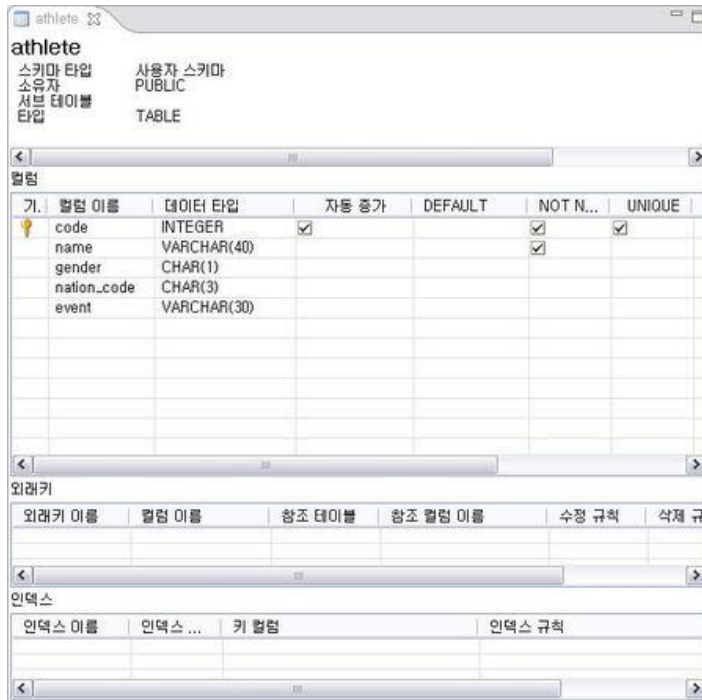


테이블 편집

[테이블 추가]의 모든 기능을 테이블 편집에서도 사용할 수 있다. 단, 테이블 내의 컬럼 순서는 조정할 수 없다.

테이블 정보 보기

해당 테이블을 더블 클릭하면 해당 테이블의 스키마 구성 정보를 조회해 볼 수 있다.



뷰

데이터베이스에 로그인하면, 접근할 수 있는 뷰와 시스템 뷰를 탐색 트리에서 볼 수 있다.



뷰 추가

탐색 트리의 [뷰]를 마우스 오른쪽 클릭해서 [뷰 추가]를 선택한다.



뷰 이름과 소유자를 설정한 후, [추가] 버튼을 클릭하고 새로 생성할 뷰에 대한 질의를 작성한다. 뷰 추가 마법사를 통해 작성한 내용은 [SQL문] 탭에서 SQL 구문으로 확인해 볼 수 있다.



전체 조회

탐색 트리의 각 뷰를 마우스 오른쪽 버튼 클릭하여 [전체 조회]를 선택하면 수행할 수 있다. 또는 [질의 편집기]가 열려 있을 때 질의 편집기의 편집 창 또는 결과 창에 해당 뷰를 끌어서 놓아서 수행할 수 있다. 수행하면 새로운 질의 편집기가 열리면서 전체 데이터를 조회한다.

개수 조회

해당 뷰의 전체 데이터 개수를 조회한다. 다음 구문과 같은 기능을 수행한다.

```
SELECT COUNT(*) FROM view
```

데이터 내려 받기

데이터 내려 받기 기능을 통해서 해당 뷰의 모든 데이터를 엑셀, CSV, SQL, 큐브리드 로드 형식으로 받을 수 있다.

뷰 삭제

해당 뷰를 삭제한다. **DROP VIEW** 문과 동일하다.

뷰 이름 변경

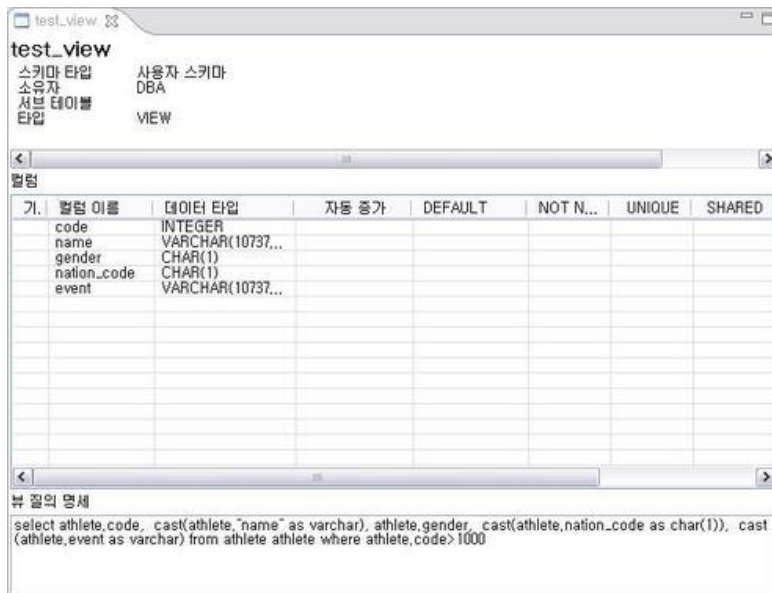
현재 뷰의 이름을 새로운 이름으로 변경한다. **RENAME VIEW** 문과 동일하다.

뷰 편집

[뷰 추가]의 모든 기능을 뷰 편집에서도 사용할 수 있다.

뷰 정보 보기

해당 뷰를 더블 클릭하면 해당 뷰의 스키마 구성 정보를 조회해 볼 수 있다.



트리거

데이터베이스에 로그인하면, 접근할 수 있는 트리거를 탐색 트리에서 볼 수 있다.

트리거 추가

탐색 트리의 [트리거]를 마우스 오른쪽 버튼 클릭해서 [트리거 추가]를 선택한다.

- **이름:** 추가할 트리거의 이름을 입력한다.
- **조건 확인 시점:** 트리거의 조건을 검사하는 시점을 선택한다. **BEFORE**, **AFTER**, **DEFERRED** 중에서 선택할 수 있다.
- **트리거 이벤트:** 발생할 이벤트의 형식을 선택한다. 이벤트 형식에는 **INSERT**, **DELETE**, **UPDATE**, **STATEMENT INSERT**, **STATEMENT DELETE**, **STATEMENT UPDATE**, **COMMIT**, **ROLLBACK**이 있다.
- **대상 테이블 / 컬럼:** 이벤트의 대상 테이블과 컬럼에 대한 정보를 입력한다.
- **수행 조건:** 트리거가 동작하기 위한 조건을 입력한다.
- **수행 시점:** 트리거가 동작하는 시점을 지정한다. **default**를 선택하면 트리거 조건 확인 시점을 기준으로 동작한다.
- **트리거 내용:** 트리거 동작 형태를 선택한다. 트리거 동작 형태가 **PRINT**, **OTHER**, **STATEMENT**인 경우, 하단의 [SQL구문 및 출력 메시지]에 부가 정보를 입력할 수 있다.
- **트리거 상태:** 추가할 트리거를 활성화할 것인지 아니면 비활성화할 것인지를 지정할 수 있다.
- **우선순위:** 트리거의 우선 순위를 지정할 수 있다. 우선 순위 값은 **FLOAT** 값으로 00.00에서 9999.99까지 지정할 수 있다.

트리거 삭제

탐색 트리에서 해당 트리거를 마우스 오른쪽 버튼 클릭해서 [트리거 삭제]를 선택한다.

트리거 편집

탐색 트리에서 해당 트리거를 마우스 오른쪽 버튼 클릭해서 [트리거 편집]을 선택한다. [우선순위]의 값만 편집할 수 있다.

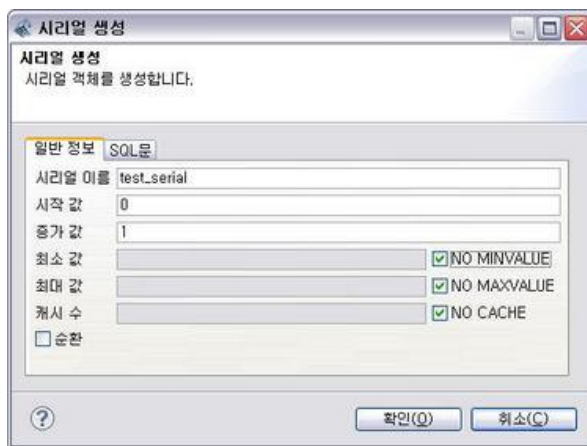
시리얼

데이터베이스에 로그인하면, 접근할 수 있는 시리얼을 탐색 트리에서 볼 수 있다.

시리얼 추가

탐색 트리의 [시리얼]을 마우스 오른쪽 버튼 클릭해서 [시리얼 추가]를 선택한다.

시리얼 이름은 데이터베이스 내에서 유일해야 한다.



시리얼 삭제

탐색 트리에서 해당 시리얼을 마우스 오른쪽 버튼 클릭해서 [시리얼 삭제]를 선택한다.

시리얼 편집

탐색 트리에서 해당 시리얼을 마우스 오른쪽 버튼 클릭해서 [시리얼 편집]을 선택한다.

저장 프로시저

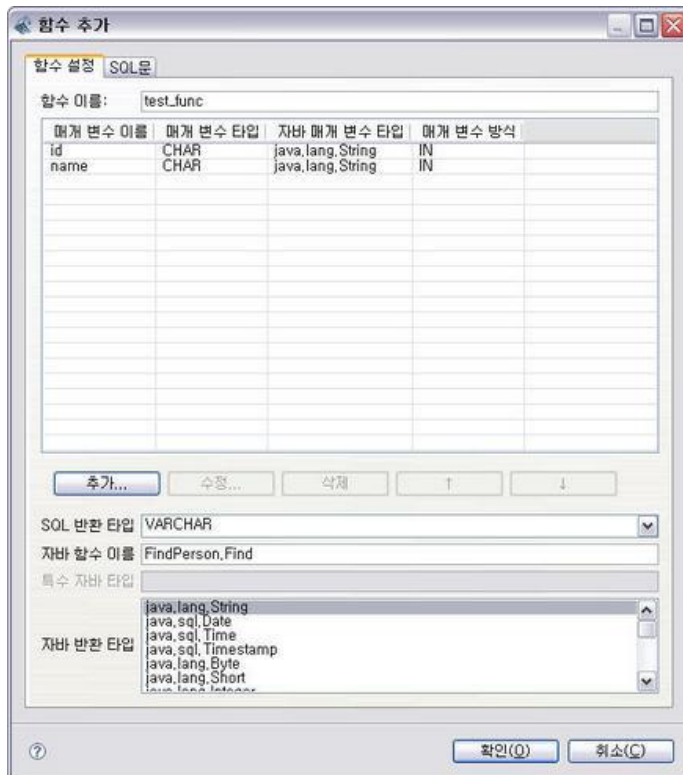
데이터베이스에 로그인하면, 접근할 수 있는 저장 프로시저를 볼 수 있다.



함수 추가

Java로 작성한 함수를 데이터베이스 서버에서 **loadjava** 명령을 통해 등록한 후, 해당 Java 함수를 사용하기 위한 데이터베이스 함수를 추가하는 기능이다.

탐색 트리의 [저장 프로시저]를 마우스 오른쪽 버튼 클릭해서 [함수 추가]를 선택한다.



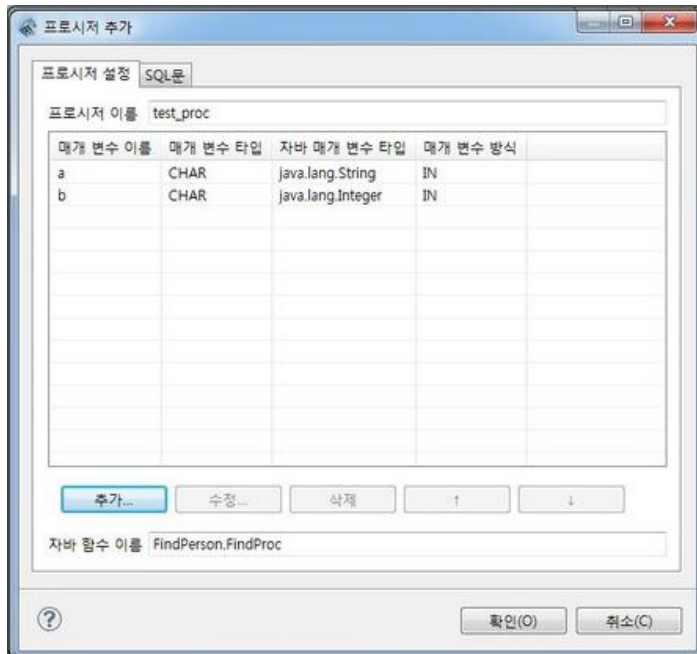
선택한 SQL 타입과 호환이 가능한 Java 타입만 보여 주어 사용자가 타입 매핑 시 실수하지 않도록 지원한다.

자세한 내용은 [Java 저장 함수/프로시저](#)를 참고한다.

프로시저 추가

Java로 작성한 프로시저를 데이터베이스 서버에서 **loadjava** 명령을 통해 등록한 후, 해당 Java 프로시저를 사용하기 위한 데이터베이스 프로시저를 추가하는 기능이다.

탐색 트리의 [저장 프로시저]를 마우스 오른쪽 버튼 클릭해서 [프로시저 추가]를 선택한다.



선택한 SQL 타입과 호환이 가능한 Java 타입만 보여 주어 사용자가 타입 맵핑 시 실수하지 않도록 지원한다.

자세한 내용은 [Java 저장 함수/프로시저](#)를 참고한다.

위 프로시저 추가 마법사를 통해 작성된 구문은 다음과 같다.

```
CREATE PROCEDURE "test_proc" ("a" CHAR, "b" CHAR)
AS LANGUAGE JAVA
NAME 'FindPerson.FindProc(java.lang.String,java.lang.Integer) '
```

함수/프로시저 삭제

탐색 트리에서 특정 함수 또는 프로시저를 마우스 오른쪽 버튼 클릭하여 [함수 삭제] 또는 [프로시저 삭제]를 선택하면 수행할 수 있다.

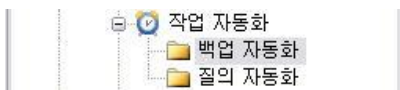
함수/프로시저 편집

탐색 트리에서 특정 함수 또는 프로시저를 마우스 오른쪽 버튼 클릭하여 [함수 편집] 또는 [프로시저 편집]을 선택하면 수행할 수 있다. 기능은 함수/프로시저 추가 시와 같다.

작업 자동화

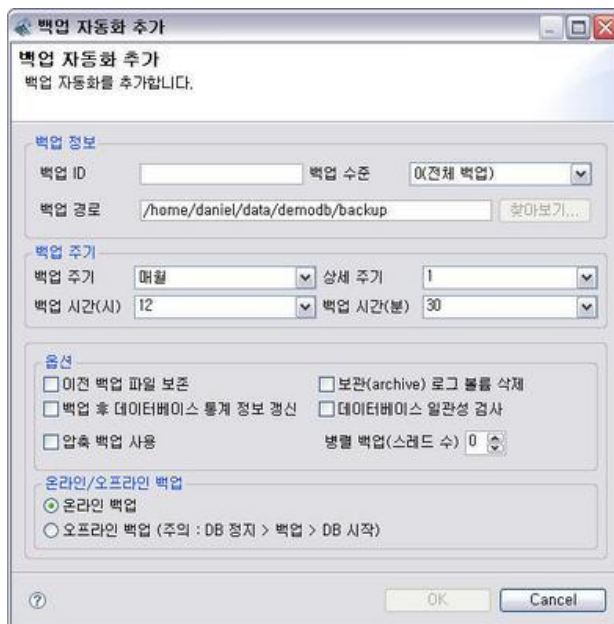
백업 자동화

데이터베이스에 로그인하면, 탐색 트리의 [작업 자동화] 아래의 [백업 자동화] 메뉴를 볼 수 있다. [백업 자동화]를 마우스 오른쪽 버튼 클릭하면 [백업 자동화 추가] 메뉴와 [백업 자동화 수행 로그] 메뉴가 나타난다.



백업 자동화 추가

CUBRID 매니저 서버를 통해서 주기적으로 백업을 수행하고자 할 경우 백업 자동화 기능을 사용하며, [백업 자동화 추가] 메뉴에서 관련 정보를 설정한다. 백업 자동화 기능은 매니저 서버가 구동 중인 상태에서 **dba** 사용자가 설정할 수 있으며, 해당 데이터베이스의 구동/중지 상태와는 무관하다. 매니저 서버가 구동 중인 상태에서만 설정된 정보에 따라 백업 작업이 자동으로 수행된다.



- **백업 ID:** 백업 작업의 이름을 설정한다. 백업 계획이 여러 개일 수 있으므로, 하나의 데이터베이스 내에서는 유일해야 한다.
- **백업 수준:** 0, 1, 2 중에서 선택할 수 있다. 0레벨은 전체 백업을 의미하고, 1레벨은 0레벨 백업 이후의 변경 내역만 백업하는 1차 증분 백업(incremental backup)을 의미하며, 2레벨은 1레벨 백업 이후의 변경

내역만 백업하는 2차 증분 백업을 의미한다. 즉, 0레벨 전체 백업본이 존재할 때에만 1 또는 2레벨 백업 작업이 자동 수행된다.

- **백업 경로:** 백업 볼륨이 저장된 디렉토리를 설정한다.
- **백업 주기:** 백업을 수행할 주기를 매월, 매주, 매일, 특정일 중에서 선택할 수 있다.
- **상세 주기:** 선택한 백업 주기를 상세하게 설정할 수 있다.
- **백업 시간:** 자동 백업을 수행할 시간을 입력한다. 시간과 분을 각각 입력한다.
- **옵션:** 자동으로 수행할 백업 작업의 옵션을 지정할 수 있다. 자세한 정보는 [데이터베이스 백업](#)을 참고한다.
- **이전 백업 파일 보존:** 현재 데이터베이스의 기존 백업 볼륨 파일을 `database_directory/backupold` 디렉토리에 저장하는 옵션이다.
- **보관 로그 볼륨 삭제:** 백업 시, 보관 로그 볼륨들을 삭제한다. 해당 데이터베이스 서버가 복제 마스터 서버로 설정되어 있는 상태에서 보관 로그 볼륨 삭제를 수행하면, 자동으로 복제에 영향이 없는 부분까지만 삭제한다.
- **백업 후 데이터베이스 통계 정보 갱신:** 백업 수행 후, 통계 정보를 갱신한다.
- **데이터베이스 일관성 검사:** 백업 시, 데이터베이스의 일관성 검사를 수행한다.
- **압축 백업 사용:** 백업 시, 압축을 사용한다.
- **병렬 백업(스레드 수):** 백업 시, 몇 개의 스레드를 동시에 이용할 것인지를 설정할 수 있다. 최대값은 CPU 개수만큼 설정하는 것이 적절하며, 디폴트 값은 0이다. 디폴트 값으로 설정된 경우 시스템에 의해 자동으로 스레드 개수가 결정된다.
- **온라인 백업:** 데이터베이스가 구동 중인 상태에서만 자동 백업을 실행한다. 중지 상태에서는 데이터베이스를 백업하지 않고 에러 로그만 기록한다.
- **오프라인 백업:** 데이터베이스가 중단 상태일 때만 자동 백업을 실행한다. 데이터베이스가 구동 중이면 데이터베이스를 강제 중단하고 자동 백업을 수행한 후, 다시 데이터베이스를 구동한다.

백업 자동화 수행 로그

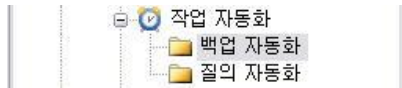
[백업 자동화 수행 로그] 메뉴에서 자동으로 수행한 백업 작업에 대한 로그를 확인할 수 있다. 백업 대상 데이터베이스, 백업 ID, 백업 수행 시간 등의 정보가 기록된다.

백업 자동화 편집 / 삭제

[백업 자동화 추가] 메뉴로 백업 ID가 every_mon_lv1이라는 백업 작업을 등록하면, 탐색 트리의 [백업 자동화] 항목 아래에 <every_mon_lv1> 항목이 생긴다. 해당 항목을 마우스 오른쪽 버튼 클릭하면 [백업 자동화 편집] 메뉴와 [백업 자동화 삭제] 메뉴가 나타나며, 등록된 백업 자동화 작업을 편집 또는 삭제할 수 있다.

질의 자동화

데이터베이스에 로그인하면, 탐색 트리의 [작업 자동화] 아래의 [질의 자동화] 메뉴를 볼 수 있다. [질의 자동화]를 마우스 오른쪽 버튼 클릭하면 [질의 자동화 추가] 메뉴와 [질의 자동화 수행 로그] 메뉴가 나타난다.



질의 자동화 추가

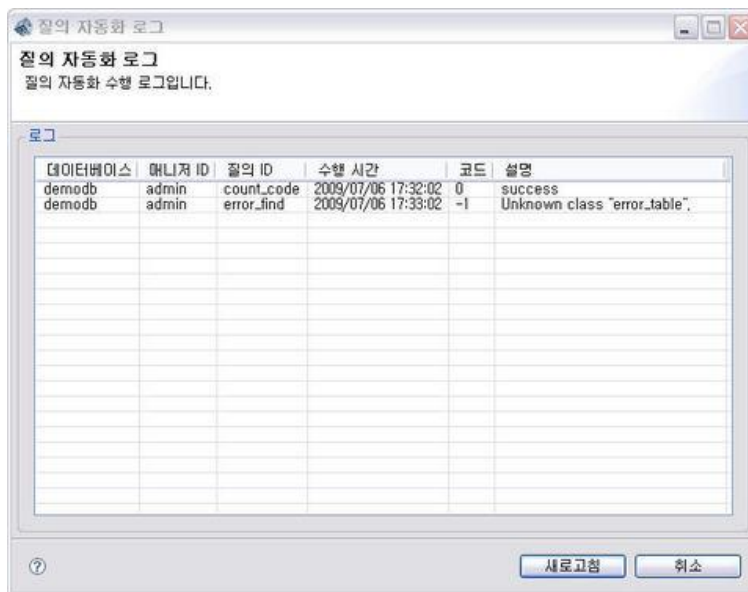
CUBRID 매니저 서버를 통해서 주기적으로 질의를 수행하고자 할 경우 질의 자동화 기능을 사용한다. [질의 자동화 추가] 메뉴에서 관련 정보를 설정할 수 있다. 질의 자동화 기능은 매니저 서버가 구동 중인 상태에서 **dba** 사용자가 설정할 수 있으며, 해당 데이터베이스의 구동/중지 상태와는 무관하다. 매니저 서버가 구동 중인 상태에서만 설정된 정보에 따라 질의 작업이 자동으로 수행된다.

- **질의 ID:** 질의 작업의 이름을 설정한다. 질의 작업이 여러 개일 수 있으므로, 하나의 데이터베이스 내에서는 유일해야 한다.
- **사용자 이름/비밀번호:** 등록된 질의를 자동으로 수행할 데이터베이스 사용자 정보를 비밀번호와 함께 입력한다. 사용자 정보가 틀리면 해당 질의를 자동으로 수행할 수 없으므로, 사용자 비밀번호가 변경되는 경우 [질의 자동화 편집] 메뉴에서 기존의 사용자 정보를 변경해야 한다.
- **실행 주기:** 질의를 자동 실행할 주기를 매월, 매주, 매일, 특정일 중에서 선택할 수 있다.
- **상세 주기:** 선택한 실행 주기를 상세하게 설정할 수 있다.
- **실행 시간:** 질의를 자동 실행할 시간을 입력한다. 시간과 분을 각각 입력한다.
- **질의 구문:** 자동 실행할 질의문을 입력한다. 등록된 질의문은 설정된 시간마다 주기적으로 자동 실행되지만, 실행 결과는 기록되지 않음을 유의한다. 자동 실행되는 질의의 결과는 기록되지 않으므로, 위 그림처럼 **SELECT** 문을 자동 실행하는 것은 아무 의미가 없다. 그러나, 통계 정보 갱신(**UPDATE STATISTICS**), 데이터 갱신 또는 삭제를 주기적으로 수행하고자 하면 질의 자동화 기능이 유용하다. 자동 실행한 질의에 대한 로그는 **\$CUBRID/log/manager/auto_execquery.log** 파일에 기록된다.

- 질의 검사: 자동 실행할 질의문을 등록하기 전에 해당 질의의 오류를 미리 검사할 수 있는 기능이다. 해당 질의를 실제 수행하지는 않고, 쿼리 플랜을 생성하여 오류를 판단한다. csq! 인터프리터에서 **optimization level 514** 모드로 수행하는 것과 동일하다.

질의 자동화 로그

[질의 자동화 수행 로그] 메뉴에서 자동으로 수행된 질의에 관한 로그를 확인할 수 있다. 대상 데이터베이스, 매니저 사용자 ID, 질의 ID, 질의 수행 시간, 에러 코드(성공:0, 실패:-1) 등의 정보가 기록된다.

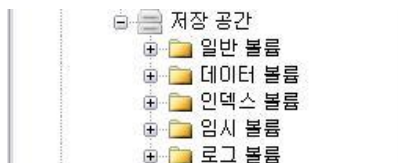


질의 자동화 편집 / 삭제

[질의 자동화 추가] 메뉴로 질의 ID가 every_mon_delete인 질의 작업을 등록하면, 탐색 트리의 [질의 자동화] 항목 아래에 <every_mon_delete> 항목이 생긴다. 해당 항목을 마우스 오른쪽 버튼 클릭하면 [질의 자동화 편집] 메뉴와 [질의 자동화 삭제] 메뉴가 나타나며, 등록된 질의 자동화 작업을 편집 또는 삭제할 수 있다.

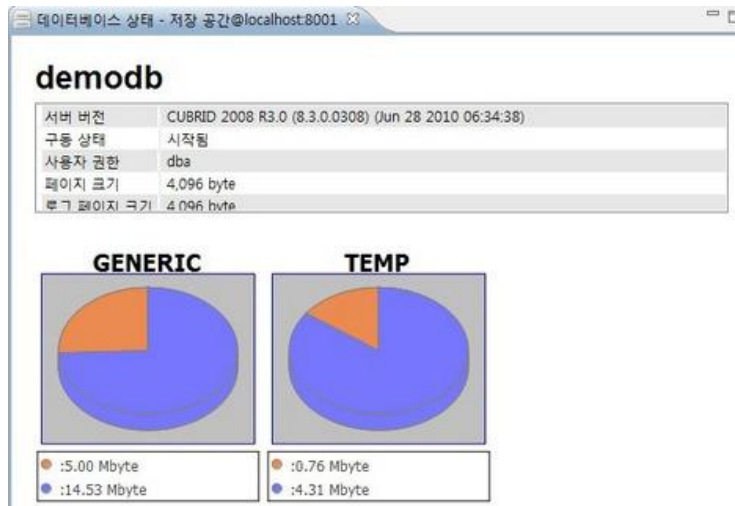
저장 공간

데이터베이스에 로그인하면, 접근할 수 있는 저장 공간을 탐색 트리에서 볼 수 있다. 데이터베이스 볼륨 상태를 분석하거나, 자동 볼륨 추가 설정 및 볼륨 추가를 수행할 수 있다.



데이터베이스 공간 정보

데이터베이스 공간 정보를 확인하려면, 탐색 트리에서 해당 데이터베이스 이름을 더블 클릭하거나, [저장 공간]을 마우스 오른쪽 버튼 클릭해서 [데이터베이스 공간 정보]를 선택한다.



탐색 트리에서 [저장 공간]의 하위 노드를 더블 클릭하면 해당 노드를 구성하는 볼륨 정보를 출력한다.

demodb-- Database space

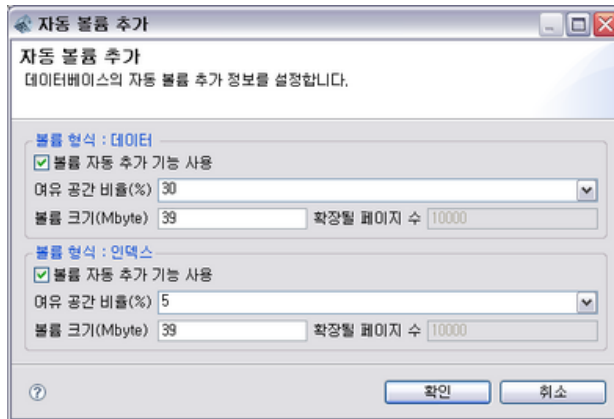
데이터 볼륨

볼륨 형식	DATA
볼륨 수	1
남은 용량	35,64M (9,124 pages)
총 용량	40M (10,240 pages)
페이지 크기	4,096 byte

볼륨 정보

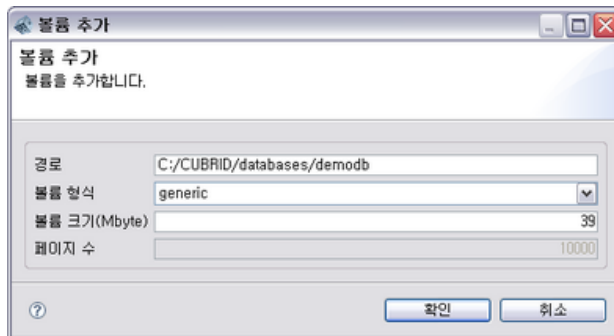
볼륨 이름	사용 용량/남은 용량(Mbyte)	총 용량	총 페이지 수
demodb_data_x001	4.36/35.64	40 M	10,240 pages

자동 볼륨 추가 설정



- **볼륨 형식:** 데이터 볼륨 또는 인덱스 볼륨에 대해서만 자동 추가 기능을 사용할 수 있다.
- **볼륨 자동 추가 기능 사용:** 해당 형식의 볼륨에 대해 자동 추가 기능을 사용한다.
- **여유 공간 비율:** [여유 공간 비율]에 지정한 값과 남은 볼륨이 같을 때, 자동으로 볼륨을 추가한다는 의미이다. 예를 들어, 여유 공간 비율이 5%이면, 기존 볼륨의 여유 공간이 5% 남았을 때, 자동으로 데이터 볼륨을 추가한다. 최소 값은 5%이고 최대 값은 30%이다.
- **볼륨 크기:** 자동으로 추가할 볼륨의 크기를 설정한다.

볼륨 추가



- **경로:** 추가할 볼륨이 저장되는 디렉터리를 입력한다. 디폴트 값은 데이터베이스 볼륨이 생성되는 디렉터리이다.
- **볼륨 형식:** 추가할 볼륨의 타입을 정한다. data, generic, index, temp 중 하나를 선택할 수 있다.
- **볼륨 크기:** 추가할 볼륨의 크기를 입력한다. 입력 단위는 Mbyte이다.
- **페이지 수:** 볼륨 크기에서 입력한 Mbyte를 해당 데이터베이스 페이지 크기로 자동 환산하여 보여 준다.

데이터베이스 로드

언로드한 데이터를 선택한 데이터베이스에 로드하려면, 해당 데이터베이스에 로그인한 후 다음 중 하나를 실행한다.

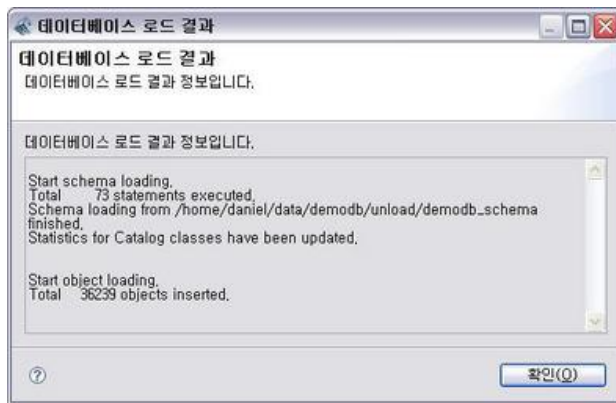
- 툴바에서 [데이터베이스 로드]를 클릭한다.
- 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하고 [데이터베이스 로드]를 선택한다.
- 메뉴에서 [동작] > [데이터베이스 로드]를 선택한다.

데이터베이스 로드 작업은 데이터베이스 서버가 중단된 상태에서만 수행할 수 있으므로 구동 중에는 [데이터베이스 복구] 메뉴가 비활성화된다.

- **데이터베이스 정보:** 언로드한 데이터를 로드할 대상 데이터베이스의 이름과 데이터베이스에 로그인한 사용자 권한이 제시된다. [사용자 이름]에는 현재 로그인한 사용자가 아닌 다른 사용자를 입력할 수 없다.

- **언로드 파일:** 언로드된 데이터의 정보를 입력한다. 리스트에서 선택하거나 언로드된 파일이 위치하는 경로를 직접 입력할 수 있다.
CUBRID 매니저 클라이언트와 CUBRID 매니저 서버가 다른 기기에서 구동된다면, 언로드된 스키마, 데이터, 인덱스, 트리거 파일은 CUBRID 매니저 서버 기기에 존재해야 한다.
기존에 CUBRID 매니저를 통해서 언로드한 정보가 있으면, 언로드 정보를 이용하여 로드할 수 있고, 없다면 기존에 언로드한 정보를 직접 입력하여 로드한다.
- **로드 옵션:** 문법 검사 후 데이터를 로드할지, 문법 검사 없이 데이터를 로드할지, 문법 검사만 수행할지 등을 선택할 수 있다. 자세한 내용은 [데이터베이스 가져오기](#)를 참조한다.

[데이터베이스 로드] 대화 상자에서 옵션을 선택한 후, [확인] 버튼을 클릭하면 데이터베이스 로드 작업의 수행 결과를 확인할 수 있다.

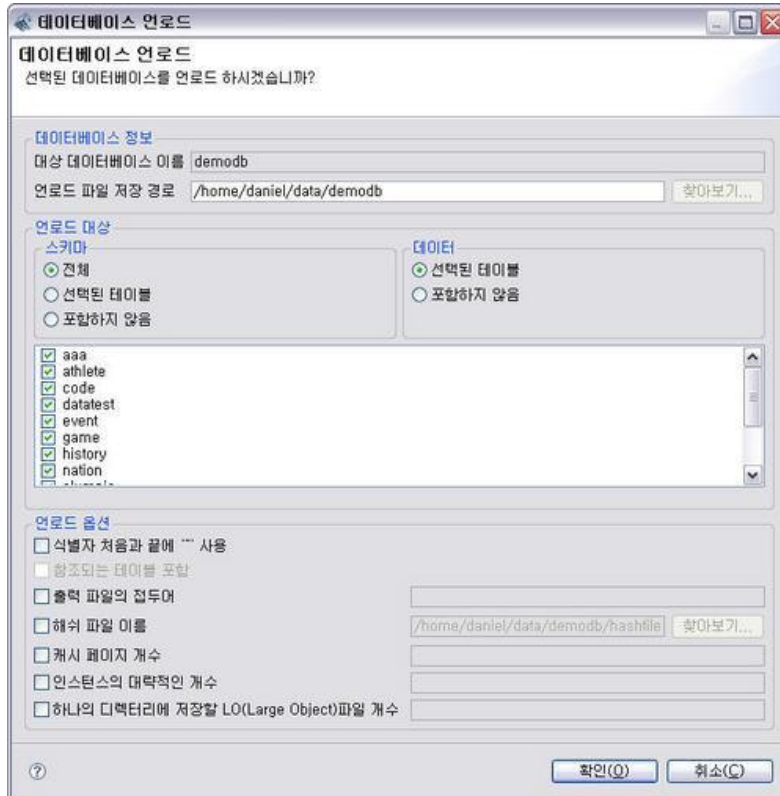


데이터베이스 언로드

선택한 데이터베이스의 데이터를 언로드하려면, 해당 데이터베이스에 로그인한 후 다음 중 하나를 실행한다.

- 툴바에서 [데이터베이스 언로드]를 클릭한다.
- 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하고 [데이터베이스 언로드]를 선택한다.
- 메뉴에서 [동작] > [데이터베이스 언로드]를 선택한다.

데이터베이스 언로드 작업은 데이터베이스 서버가 구동 중일 때에도 수행할 수 있다.




- **데이터베이스 정보:** 언로드할 데이터베이스 이름이 출력되며, 언로드 작업 후 생성될 파일이 저장될 디렉터리를 지정한다.
- **언로드 대상:** 언로드할 데이터베이스의 스키마와 데이터를 선택한다.
- **언로드 옵션:** 식별자를 문자열 처리(")할지, 한 디렉터리에 LO 파일을 몇 개씩 저장할지 등을 선택할 수 있다. 자세한 내용은 [데이터베이스 내보내기](#)를 참조한다.

[데이터베이스 언로드] 대화 상자에서 옵션을 선택한 후, [확인] 버튼을 클릭하면 데이터베이스 언로드 작업의 수행 결과를 확인할 수 있다.



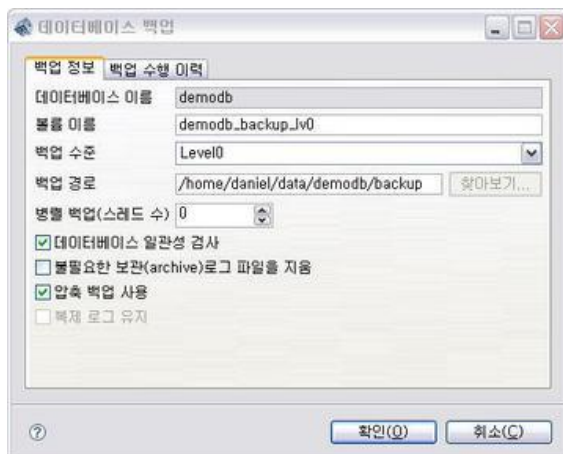
데이터베이스 백업

데이터베이스의 데이터를 백업하려면, 해당 데이터베이스에 로그인한 후 다음 중 하나를 실행한다.

- 툴바에서 [데이터베이스 백업] 을 클릭한다.
- 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하고 [데이터베이스 백업]을 선택한다.
- 메뉴에서 [동작] > [데이터베이스 백업]을 선택한다.

백업 작업은 데이터베이스 서버가 구동 중일 때에도 수행할 수 있다

데이터베이스 백업에 대한 자세한 사항은 [데이터베이스 백업](#)을 참조한다.



- **데이터베이스 이름:** 백업할 데이터베이스의 이름이다.

데이터베이스 로드 작업은 데이터베이스 서버가 중단된 상태에서만 수행할 수 있으므로 구동 중에는 [데이터베이스 복구] 메뉴가 비활성화된다.

데이터베이스 복구에 대한 자세한 사항은 [데이터베이스 복구](#)를 참조한다.



- **데이터베이스 이름:** 복구할 대상이 되는 데이터베이스 이름이다.
- **복구 일자 및 시간:** 어느 시점으로 데이터베이스를 복구할지 지정한다. [백업 시점으로 복구]를 선택하면 복구 명령어에 **backuptime** 키워드를 사용하여 복구하며, 이는 백업이 완료된 시점에서의 복구를 의미한다. [특정 시점으로 복구]를 선택하면 사용자가 원하는 날짜와 시간을 입력할 수 있다.
- **가용한 백업 정보:** 복구할 대상이 되는 데이터베이스가 어떤 백업 레벨로 백업되었는지 확인하여, 원하는 복구 레벨을 선택할 수 있다. 표시되는 파일 경로는 해당 레벨로 백업된 파일이 위치하는 디렉터리 경로이다.
- **보관 로그가 존재하지 않으면 강제로 부분 복구 수행:** 완전한 로그 정보가 없으면 부분 복구를 수행한다. 즉, 데이터베이스를 복구할 때, 백업 시점 이후에 생성된 보관 로그나 활동 로그가 없어도 복구를 수행할 수 있다.
- **사용자가 지정한 경로에 복구:** 데이터베이스 위치 정보 파일(**databases.txt**)에 지정된 경로로 대상 데이터베이스를 복구한다.
- **백업 정보 보기:** 선택한 백업 수준으로 백업된 파일의 정보를 보여준다.



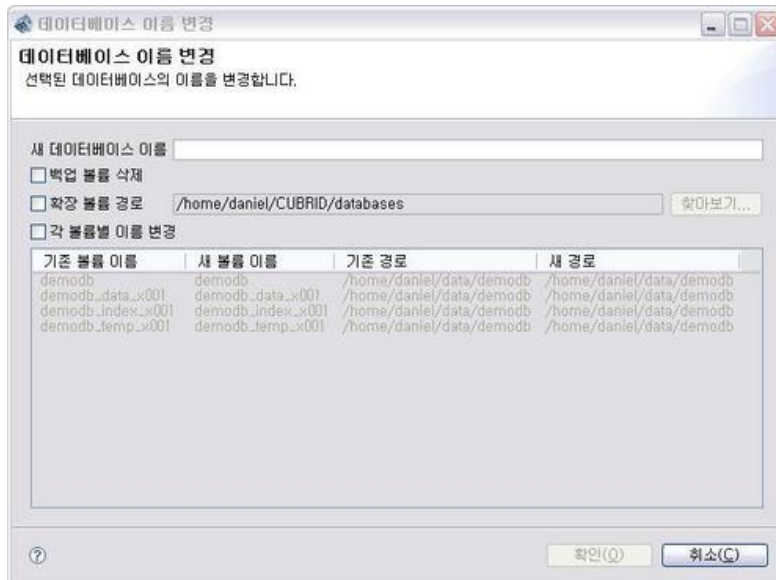
데이터베이스 이름 변경

데이터베이스의 이름을 변경하려면, 해당 데이터베이스에 로그인한 후 다음 중 하나를 실행한다.

- 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하고 [데이터베이스 이름 변경]을 선택한다.
- 메뉴에서 [동작] > [데이터베이스 이름 변경]을 선택한다.

데이터베이스 이름 변경 작업은 데이터베이스 서버가 중단된 상태에서만 수행할 수 있으므로 구동 중에는 [데이터베이스 이름 변경] 메뉴가 비활성화된다.

데이터베이스 이름 변경에 대한 자세한 사항은 [데이터베이스 변경](#)을 참조한다.



- **새 데이터베이스 이름:** 이름을 변경하고자 하는 데이터베이스의 새 이름을 입력한다.
- **백업 볼륨 삭제:** 이름 변경 전의 데이터베이스의 백업 볼륨을 삭제한다.
- **확장 볼륨 경로:** 새 이름의 데이터베이스에서 추가될 볼륨들이 저장되는 경로를 지정하는 옵션이다.
- **각 볼륨별 이름 변경:** 변경 전의 데이터베이스 볼륨이 여러 개인 경우, 각각의 데이터베이스 볼륨의 이름을 변경하고, 각 데이터베이스 볼륨이 저장되는 새로운 디렉터리 경로를 지정할 수 있다.

데이터베이스 복사

데이터베이스를 복사하려면, 해당 데이터베이스에 로그인한 후 다음 중 하나를 실행한다.

- 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하고 [데이터베이스 복사]를 선택한다.
- 메뉴에서 [동작] > [데이터베이스 복사]를 선택한다.


데이터베이스 복사 작업은 데이터베이스 서버가 중단된 상태에서만 수행할 수 있으므로 구동 중에는 [데이터베이스 복사] 메뉴가 비활성화된다.



- **원본 데이터베이스:** 복사할 데이터베이스의 이름, 데이터베이스 볼륨이 저장된 경로 및 로그 파일이 저장된 경로가 표시된다.
- **대상 데이터베이스:** 복사하면 생성될 데이터베이스의 이름, 데이터베이스 볼륨이 저장될 경로, 확장 볼륨이 저장될 경로 및 로그 파일이 저장될 경로를 입력한다.
- **개별 볼륨 복사 설정:** 원본 데이터베이스의 볼륨이 여러 개인 경우, 각 데이터베이스 볼륨의 이름을 변경하고, 각 데이터베이스 볼륨이 복사될 새로운 디렉터리 경로를 지정할 수 있다.
- **동일한 파일 덮어쓰기:** 대상 데이터베이스와 이름이 동일한 데이터베이스가 이미 존재하면 덮어쓴다.
- **원본 데이터베이스 삭제:** 데이터베이스 복사 후, 원본 데이터베이스를 삭제한다.

데이터베이스 최적화

데이터베이스를 최적화하려면, 해당 데이터베이스에 로그인한 후 다음 중 하나를 실행한다.

- 툴바에서 [데이터베이스 최적화 - 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하고 [데이터베이스 최적화]를 선택한다.

- 메뉴에서 [동작] > [데이터베이스 최적화]를 선택한다.

[데이터베이스 최적화] 대화 상자에서 모든 테이블 또는 일부 테이블에 대해 최적화를 수행할 수 있다.

최적화할 대상을 선택하고 [확인]을 클릭하면 데이터베이스 최적화가 진행된다.

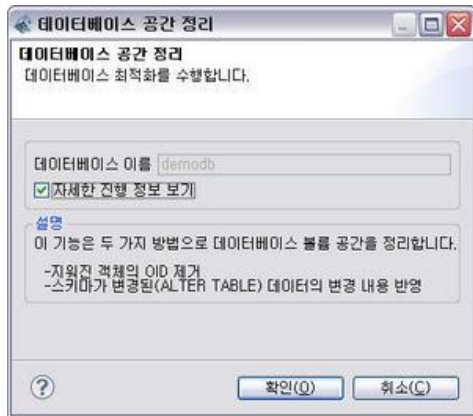


데이터베이스 공간 정리

데이터베이스의 공간을 정리하려면, 해당 데이터베이스에 로그인한 후 다음 중 하나를 실행한다.


- 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하고 [데이터베이스 공간 정리]를 선택한다.
- 메뉴에서 [동작] > [데이터베이스 공간 정리]를 선택한다.

데이터베이스 공간 정리 작업은 데이터베이스 서버가 중단된 상태에서만 수행할 수 있으므로 구동 중에는 [데이터베이스 공간 정리] 메뉴가 비활성화된다. [자세한 진행 정보 보기]를 선택하면, 공간 정리 작업 완료 후 작업에 대한 자세한 정보가 출력된다.



데이터베이스 검사


데이터베이스를 검사하려면, 해당 데이터베이스에 로그인한 후 다음 중 하나를 실행한다.

- 툴바에서 [데이터베이스 검사] 를 클릭한다.
- 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하고 [데이터베이스 검사]를 선택한다.
- 메뉴에서 [동작] > [데이터베이스 검사]를 선택한다.



데이터베이스 잠금 정보

데이터베이스의 잠금 정보를 보려면, 해당 데이터베이스에 로그인한 후 다음 중 하나를 실행한다.

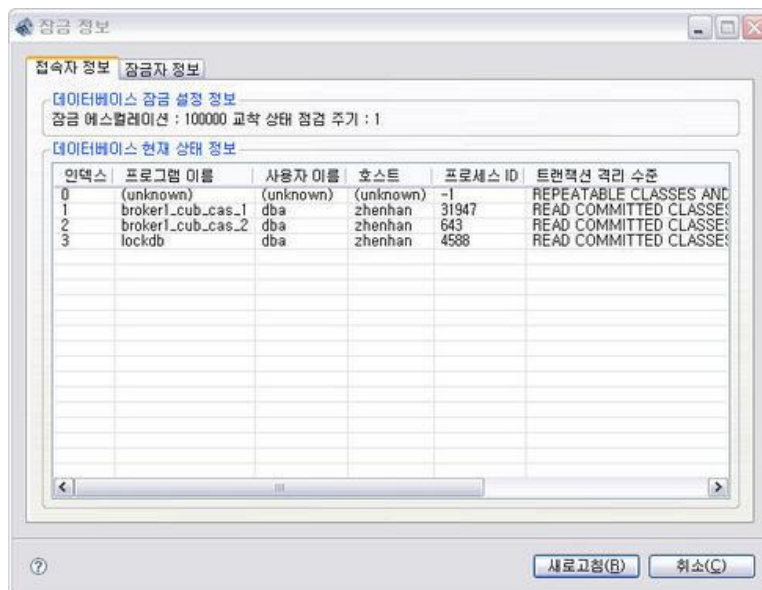
- 툴바에서 [데이터베이스 잠금 정보] 를 클릭한다.
- 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하고 [데이터베이스 잠금 정보]를 선택한다.

- 메뉴에서 [동작] > [데이터베이스 잠금 정보]를 선택한다.

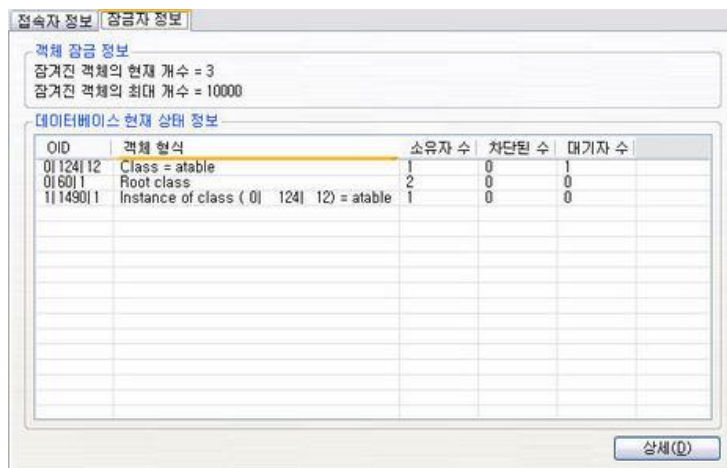
이 기능은 데이터베이스 서버가 구동 중일 때에만 수행할 수 있으며, 서버가 구동하지 않을 때에는 해당 메뉴가 비활성화된다.

[잠금 정보] 대화 상자는 두 개의 탭으로 이루어져 있다.

[접속자 정보] 탭은 현재 데이터베이스에 접속한 클라이언트의 정보를 제공한다.



[잠금자 정보] 탭에서는 데이터베이스 객체에 대한 잠금 정보를 확인할 수 있다.



[잠금자 정보] 탭에서 [상세]를 클릭하면 잠금에 대한 상세 정보를 볼 수 있다.

잠금 상세 정보
잠금 상세 정보입니다.

객체 ID: 124112
객체 형식: Class = atable
잠금 소유자

트랜잭션 인덱스	허가된 잠금 모드	개수	잠긴 하위 행 수
12	IX-LOCK	1	1

차단된 잠금 소유자

트랜잭션 인덱스	허가된 잠금 모드	개수	차단된 잠금 모드	대기 시작 시간

잠금 대기자

트랜잭션 인덱스	차단된 잠금 모드	대기 시작 시간	대기 설정 시간
1	X-LOCK	Mon Jul 6 22:08:06 2009	-1

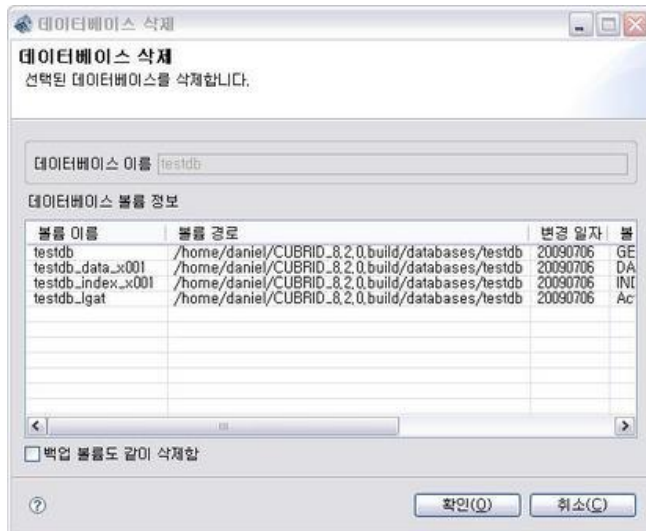
확인(O)

데이터베이스 트랜잭션 정보

데이터베이스의 트랜잭션 정보를 보려면, 해당 데이터베이스에 로그인한 후 다음 중 하나를 실행한다.

- 톨바에서 [데이터베이스 트랜잭션 정보]를 클릭한다.
- 해당 데이터베이스를 마우스 오른쪽 버튼 클릭하고 [데이터베이스 트랜잭션 정보]를 선택한다.
- 메뉴에서 [동작] > [데이터베이스 트랜잭션 정보]를 선택한다.

이 기능은 데이터베이스 서버가 구동 중일 때에만 수행할 수 있으며, 서버가 구동하지 않을 때에는 해당 메뉴가 비활성화된다.



[확인]을 클릭하면 **dba** 비밀번호를 확인한 후 삭제한다. **dba** 이외의 사용자는 삭제할 수 없다.

OID 탐색기

OID로 데이터를 탐색할 때 사용한다.

해당 데이터베이스 또는 질의 편집기의 결과 창에서 마우스 오른쪽 버튼 클릭하고 [OID 탐색기]를 선택하여 실행할 수 있다. 이 기능은 데이터베이스 서버가 구동 중일 때에만 수행할 수 있으며, 서버가 구동하지 않을 때에는 해당 메뉴가 비활성화된다.

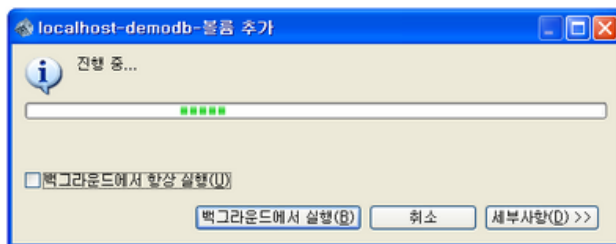


다중 호스트 환경에서의 백그라운드 운영

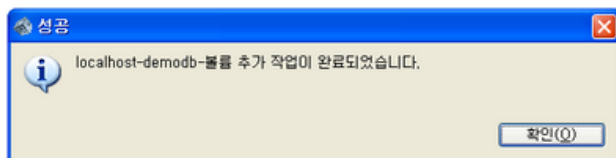
[호스트 추가]를 이용하여 여러 호스트의 데이터베이스를 관리하는 경우, 데이터베이스 운영 작업이 백그라운드에서 실행되도록 설정할 수 있다. 예를 들어, A 호스트의 데이터베이스를 백업하는 도중에 B 호스트의 데이터베이스에 질의를 실행하고자 한다면, 시간이 오래 걸리는 데이터베이스 백업 작업이 백그라운드에서 실행되도록 설정할 수 있다. 다음은 백그라운드에서 실행할 수 있는 데이터베이스 운영 작업이다.

- **데이터베이스 관련:** 데이터베이스 생성, 데이터베이스 언로드/로드, 데이터베이스 백업/복구, 데이터베이스 이름 변경, 데이터베이스 복사, 데이터베이스 최적화, 데이터베이스 공간 정리, 데이터베이스 검사
- **테이블 관련:** 데이터 올리기/내려 받기, 모든 데이터 삭제, 제약 조건을 **NULL**에서 **NOT NULL**로 변경하면서 데이터 업데이트
- **볼륨 관련:** 볼륨 추가

볼륨을 추가하면 다음과 같은 창이 나타난다. 이 창에서 [백그라운드에서 실행]을 선택할 수 있다.



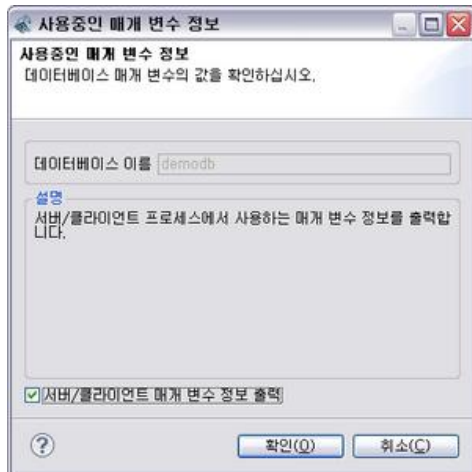
해당 운영 작업이 실행 완료되면, 다음과 같은 작업 완료 창이 나타난다.



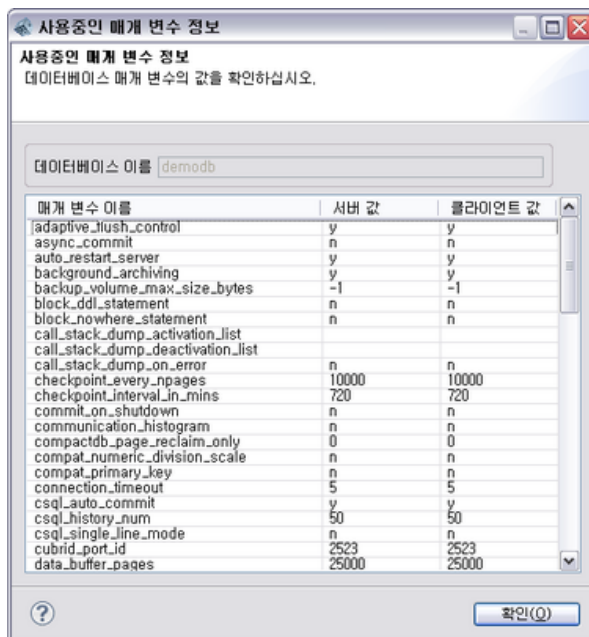
운영 작업이 백그라운드로 실행 중인 상태에서 관리자가 해당 데이터베이스를 종료하거나 해당 데이터베이스나 호스트로부터 로그아웃하면, 백그라운드로 수행 중인 작업이 있음을 알리는 창이 나타난다.

사용 중인 매개 변수 정보

해당 데이터베이스에서 사용하는 매개 변수(parameter) 정보를 출력한다. **cubrid paramdump** 유틸리티 실행 시 출력되는 항목을 매니저에서 확인할 수 있다. 유틸리티에 관한 자세한 내용은 [서버/클라이언트에서 사용하는 파라미터 출력](#)을 참고한다.



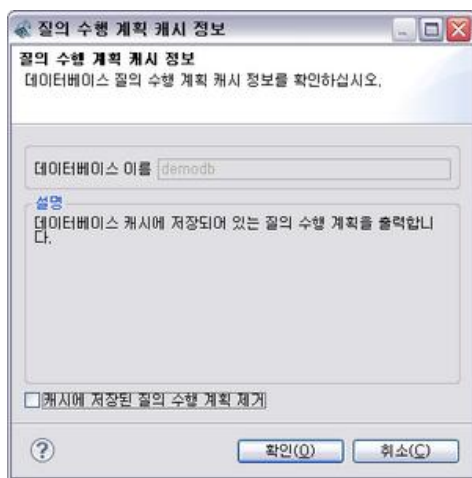
[서버/클라이언트 매개 변수 정보 출력]을 선택하면 서버에 적용된 파라미터 값과 클라이언트에 적용된 파라미터 값을 모두 확인할 수 있고, 선택을 해제하면 서버에 적용된 파라미터 값만 확인할 수 있다. [확인]을 클릭하면 아래와 같이 서버/클라이언트 파라미터 값을 확인할 수 있다.



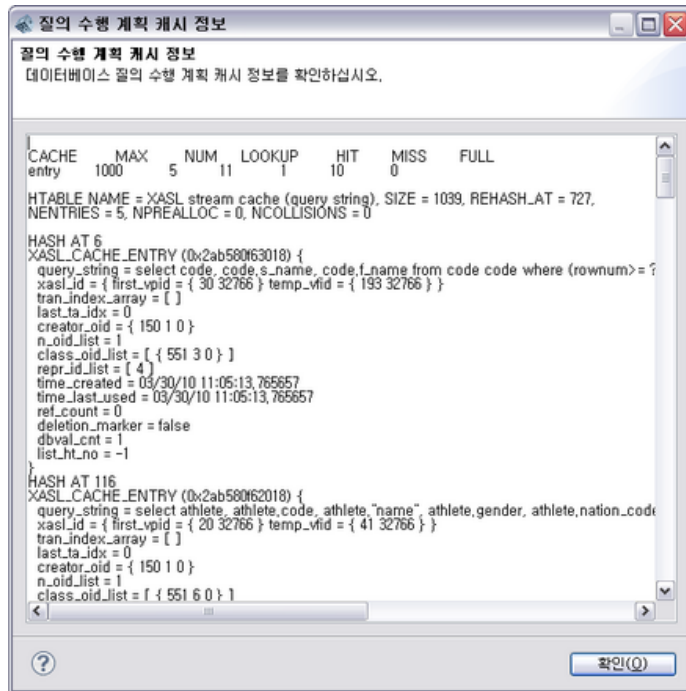
질의 수행 계획 캐쉬 정보

해당 데이터베이스 서버에 캐시 저장되어 있는 질의 수행 계획에 관한 정보를 출력한다. **cubrid plandump** 유틸리티 실행 시 출력되는 항목을 매니저에서 확인할 수 있다. 유틸리티에 관한 상세한 설명은 [질의 수행 계획 캐시 확인](#)을 참고한다.

이 기능은 데이터베이스 서버 구동 중에만 수행할 수 있으므로, 데이터베이스 서버가 구동 중일 때만 [질의 수행 계획 캐시 정보] 메뉴가 활성화된다.



[캐시에 저장된 질의 수행 계획 제거]를 선택하면, 수집한 데이터베이스 질의 수행 계획 캐시 정보를 출력한 후 이를 초기화한다. [확인] 버튼을 누르면 아래와 같이 캐시에 저장된 질의 수행 계획 정보를 확인할 수 있다.

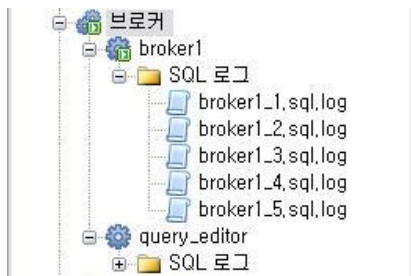


브로커

브로커 구조

브로커는 다양한 인터페이스(ODBC, OLEDB, JDBC, PHP 등)가 데이터베이스에 접속할 수 있도록 연결해주는 연결자로서 다양한 기능을 가진다. 자세한 내용은 [관리자 안내서](#)를 참고한다.

브로커는 설정된 각 브로커 이름과 해당 브로커의 SQL로그로 구성된다.



브로커 메뉴

탐색 트리에서 [브로커]를 마우스 오른쪽 버튼 클릭하면, 브로커 전체에 대한 기능을 수행할 수 있다. [상태 보기], [브로커 시작], [브로커 중지], [속성], [새로고침]으로 구성되어 있다.

브로커가 시작되어 있으면 탐색 트리에서 아이콘으로 표시되고 중지 상태이면 아이콘으로 표시된다.

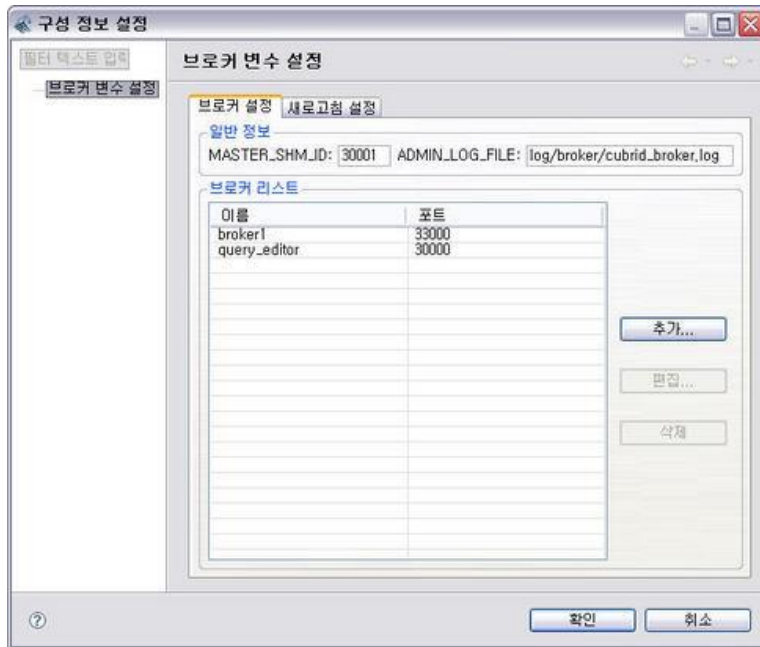


상태 보기

[상태 보기]를 선택하면, 브로커 전체에 대한 상태 정보를 확인할 수 있다. 이는 아래의 명령어를 통해서도 확인할 수 있다.

```
cubrid broker status
```

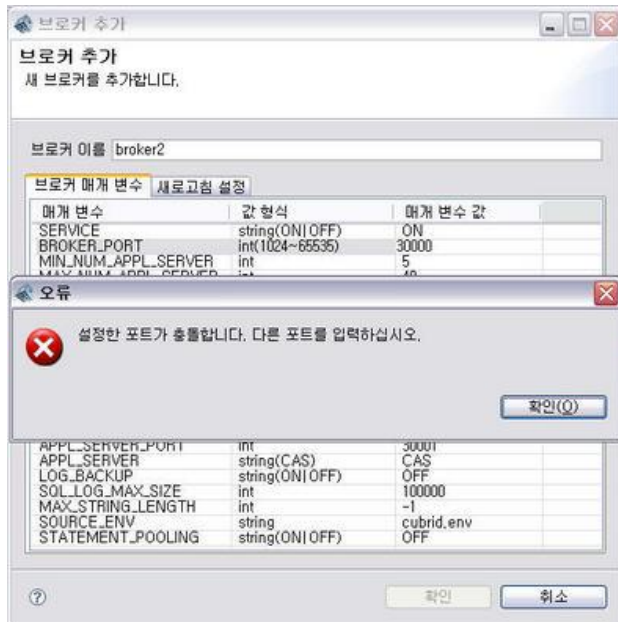
주기적으로 브로커 상태 정보를 모니터링하려면, [속성]의 [새로고침 설정]에서 [켜기]를 선택하고 새로 고침 간격을 설정한다.



\$CUBRID/conf/cubrid_broker.conf 파일내의 모든 구성 정보를 GUI로 추가, 편집, 삭제할 수 있다. 각 매개 변수에 대한 자세한 설명은 [브로커 설정](#)을 참조한다.

매개 변수	값 형식	매개 변수 값
SERVICE	string(ON OFF)	ON
BROKER_PORT	int(1024~65535)	33000
MIN_NUM_APPL_SERVER	int	5
MAX_NUM_APPL_SERVER	int	40
APPL_SERVER_SHM_ID	int(1024~65535)	33000
APPL_SERVER_MAX_SIZE	int	20
LOG_DIR	string	log/broker/sql_log
ERROR_LOG_DIR	string	log/broker/error_log
SOL_LOG	string(ON OFF)	ON
TIME_TO_KILL	int	120
SESSION_TIMEOUT	int	300
KEEP_CONNECTION	string(ON OFF AUTO)	AUTO
ACCESS_LIST	string(ON OFF)	
ACCESS_LOG	string	ON
APPL_SERVER_PORT	int	30001
APPL_SERVER	string(CAS)	CAS
LOG_BACKUP	string(ON OFF)	OFF
SOL_LOG_MAX_SIZE	int	100000
MAX_STRING_LENGTH	int	-1
SOURCE_ENV	string	cubrid.env
STATEMENT_POOLING	string(ON OFF)	OFF

브로커에서 설정하는 모든 값은 입력 또는 편집 시 바로 유효성 검사를 수행하여 사용자의 실수를 최소화한다.



개별 브로커 기능

브로커는 다수의 개별 브로커로 구성될 수 있으며, 개별 브로커는 유일한 브로커 이름, 포트, 공유 메모리 ID로 설정해야 한다.



각 브로커의 상태 보기와 속성 편집이 가능하며, 해당 브로커를 시작/정지할 수 있다.

시작된 브로커는 탐색 트리에서 아이콘으로 표시되고, 정지 상태의 브로커는 아이콘으로 표시된다.

broker1

ID	PID	REQ	SIZE	STATUS	LAST ACCESS	JOB INFO
1	17310	0	10156	IDLE	2009/07/05 16:11:21	
2	17311	0	9580	IDLE	2009/07/05 16:11:21	
3	17312	0	10552	IDLE	2009/07/05 16:11:21	
4	17313	0	8876	IDLE	2009/07/05 16:11:21	
5	17314	0	10032	IDLE	2009/07/05 16:11:21	

작업 큐 상태

ID	PRIORITY	ADDRESS	TIME	REQ

SQL 로그

해당 브로커의 SQL_LOG 매개 변수 값이 ON이면 해당 브로커를 통해서 수행된 모든 질의는 로그 파일에 기록된다. 이 로그 파일을 CUBRID 매니저에서 보고 분석 및 재실행해 볼 수 있다.



로그 보기

선택한 SQL 로그 파일에 기록된 SQL 로그를 100라인씩 읽어 와서 볼 수 있는 기능으로, 로그 정보 중 특정 범위를 선택하여 복사할 수 있는 기능을 제공한다.

SQL 로그--broker1-1.sql.log

번호	내용
70	07/05 17:17:50.115 (0) auto_commit 0
71	*** 0.019
72	
73	07/05 17:17:50.115 (0) close_req_handle srv_h_id 2
74	07/05 17:17:50.116 (6) prepare 0 SELECT * FROM athlete WHERE ROWNUM BETWEEN 1 AND (RANDOMC
75	07/05 17:17:50.117 (6) prepare srv_h_id 2 (PC)
76	07/05 17:17:50.117 (6) execute srv_h_id 2 SELECT * FROM athlete WHERE ROWNUM BETWEEN 1 AND (R
77	07/05 17:17:50.117 (6) execute 0 tuple 1 time 0.000
78	07/05 17:17:50.117 (0) auto_commit
79	07/05 17:17:50.127 (0) auto_commit 0
80	*** 0.012
81	
82	07/05 17:17:50.128 (1) execute srv_h_id 1 SELECT 1 FROM db_root
83	07/05 17:17:50.128 (1) execute 0 tuple 1 time 0.000
84	07/05 17:17:50.128 (0) auto_commit
85	07/05 17:17:50.128 (0) auto_commit 0

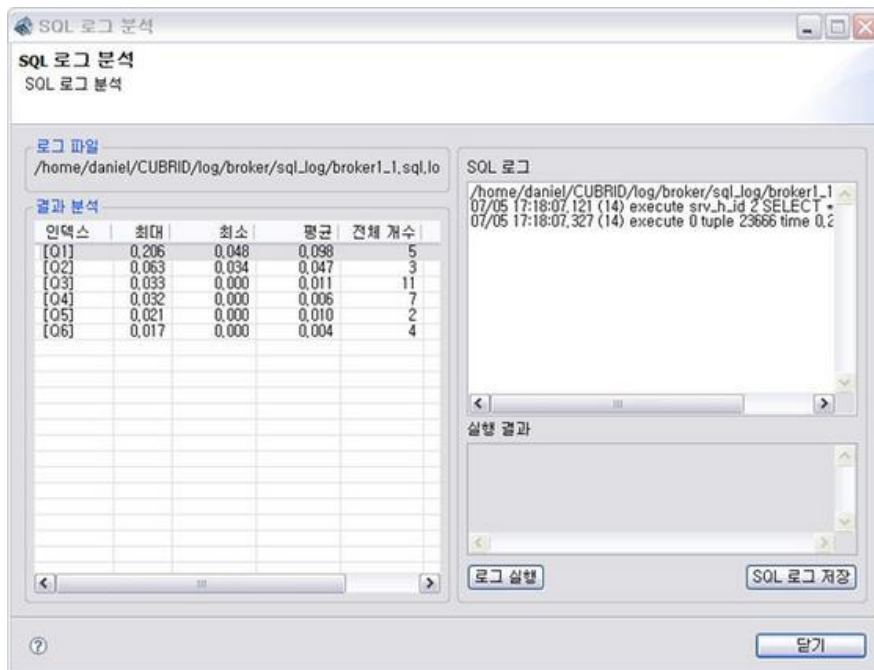
1-100 (354)

로그 분석

[로그 분석]을 선택하면, 어떤 브로커 SQL 로그를 분석할 것인지 선택할 수 있는 [분석 파일 선택] 대화 상자가 나타난다. [트랜잭션 기반 분석]을 선택하면 트랜잭션 단위로 로그 분석을 수행하고, 체크하지 않으면 각 질의마다 로그 분석을 수행한다.



[분석 대상 파일 선택] 대화 상자에서 분석하고자 하는 SQL 로그 파일을 선택하고 [확인]을 클릭하면, 선택한 로그 파일을 분석한 결과를 보여주는 [SQL 로그 분석] 대화 상자가 나타난다.



- **로그 파일:** 분석 대상이 되는 브로커 SQL 로그 파일의 이름 및 디렉터리 경로를 표시한다.
- **결과 분석:** 로그 분석 결과를 보여준다. [트랜잭션 기반 분석]을 선택했다면 각 트랜잭션의 수행 시간이 표시되고, 선택하지 않았다면 각 질의별 분석 정보(전체 수행 횟수, 에러 횟수, 최고 수행 시간, 최저 수행 시간, 평균 수행시간)가 표시된다. 결과 분석 목록의 컬럼을 클릭하면, 선택된 컬럼 값을 기준으로 분석 결과가 정렬된다.
- **SQL 로그:** 분석 결과에 관한 로그 내용을 보여준다.
- **실행 결과:** 로그 실행 결과를 보여준다.
- **로그 실행:** 로그 내용에 있는 SQL 로그를 재실행한다. 로그 질의를 변경하고 재실행하면서 질의를 튜닝하고 오류를 수정할 수 있다.
- **SQL 로그 저장:** 로그 내용을 파일로 저장한다.

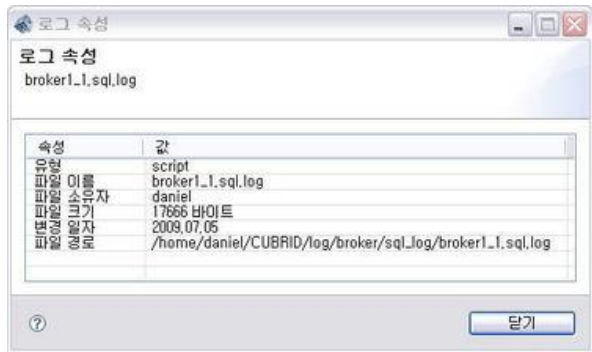
로그 재실행

[로그 재실행] 메뉴를 선택하면, 다음과 같이 로그 재실행을 위한 환경을 설정할 수 있는 [재실행 정보] 대화 상자가 나타난다.

- **데이터베이스:** 로그를 재실행할 데이터베이스를 선택한다.
- **브로커 이름:** 로그를 재실행할 브로커를 선택한다.
- **사용자 ID/비밀번호:** 로그를 재실행할 데이터베이스의 사용자 ID와 비밀번호를 입력한다.
- **동시 실행 개수:** 동시에 실행할 로그 질의를 개수를 지정한다. 로그가 재실행될 때, 이 개수만큼의 스레드가 생성되어 같은 질의가 동시에 실행된다. 다중 사용자 환경에서 질의가 어떻게 실행되는지 확인할 때 유용한 기능이다.
- **반복 횟수:** 질의의 반복 실행 횟수를 지정한다.
- **질의 실행 결과 보기:** 질의 실행 결과를 확인할 수 있다.
- **질의 실행 계획 보기:** 이 옵션은 [질의 실행 결과 보기]를 선택한 경우에만 유효하다.

로그 속성

해당 로그 파일의 정보를 제공한다.



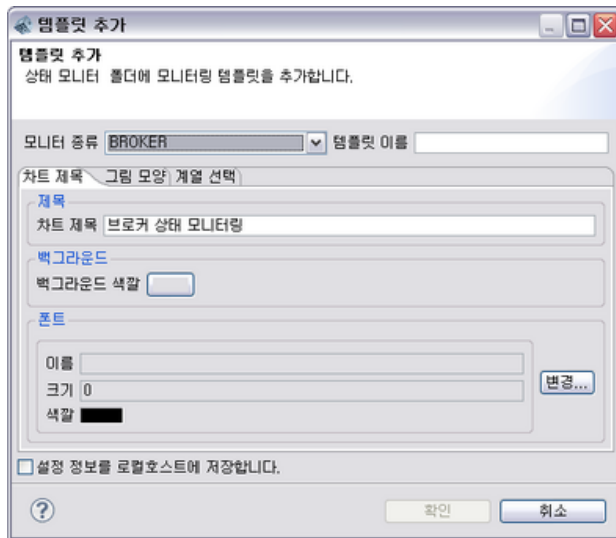
상태 모니터

상태 모니터는 다음과 같은 기능을 제공한다.

상태 모니터 추가

탐색 트리에서 [상태 모니터]를 마우스 오른쪽 버튼 클릭하고 [상태 모니터 추가]를 선택하면 아래와 같은 [템플릿 추가] 대화 상자가 나타나며, 모니터링 대상, 모니터링 항목 및 모니터링 차트 속성을 선택하여 모니터링 템플릿을 생성할 수 있다. 단, 3.0 미만 버전의 서버에 접속하는 경우 상태 모니터 추가창의 기능과 설정 방법이 다를 수 있다.

- **[모니터 종류]:** [DATABASE] 또는 [BROKER] 중 하나의 모니터링 대상을 선택할 수 있다.
- **[템플릿 이름]:** 생성할 상태 모니터링 템플릿의 이름을 입력한다.
- **[차트 제목] 탭:** 차트 상단에 출력될 제목, 배경라운드 색상, 폰트 등을 설정할 수 있다.
- **[그림 모양] 탭:** 차트 제목, 가로/세로 축 및 눈금선 등의 색상을 설정할 수 있다.
- **[계열 선택] 탭:** 모니터링 항목을 선택할 수 있다.



데이터베이스 모니터링 항목

[모니터 종류]로 [DATABASE]를 선택하면, [계열 선택] 탭에서 원하는 데이터베이스 모니터링 항목을 선택하여 템플릿에 추가할 수 있다.

데이터베이스 모니터링은 **cuprid statdump** 유틸리티를 이용해 CUBRID 데이터베이스 서버가 실행한 통계 정보를 수집하여 보여 주며, 데이터 페이지 버퍼 적중률을 비롯하여 로그, 페이지, 인덱스, 질의, 트랜잭션과

같은 항목을 모니터링한다. 데이터베이스 모니터링 항목에 관한 보다 자세한 내용은 [데이터베이스 서버 실행 통계 정보 출력](#)을 참고한다.

브로커 모니터링 항목

[모니터 종류]로 [BROKER]를 선택하면, [계열 선택] 탭에서 원하는 브로커 모니터링 항목을 선택하여 템플릿에 추가할 수 있다.

브로커 모니터링 항목은 ACTIVE_SESSION, ERR_Q, LONG_Q, LONG_T, QPS, RPS, TPS가 있다. 자세한 내용은 [브로커 상태 확인](#)을 참고한다. 브로커의 ACTIVE_SESSION을 제외한 나머지 모니터링 항목은 모두 마지막 샘플링 시간 간격(1초)마다 수집한다.

- **ACTIVE_SESSION**: 브로커에 현재 접속 중인 세션 수를 나타낸다. 즉, **BUSY** 수이다.
- **ERR_Q**: 에러가 발생한 질의 수이다.
- **LONG_Q**: 장기 수행된 질의 수이다.
- **LONG_T**: 장기 수행된 트랜잭션 수이다.
- **QPS**: 브로커에서 처리한 초당 질의 수이다.
- **RPS**: 브로커에 들어온 초당 요청 횟수이다.
- **TPS**: 브로커에서 처리한 초당 트랜잭션 수이다.

템플릿 정보

- **이름**: 새롭게 생성할 상태 모니터링 템플릿의 이름을 입력한다.
- **설명**: 생성할 템플릿에 관한 설명을 입력한다.
- **샘플링 간격(초)**: 대상 객체를 모니터링할 주기를 초 단위로 지정한다.
- **대상 데이터베이스**: 모니터링 대상 객체가 데이터베이스 서버와 관련된 경우 활성화되며, 대상 데이터베이스를 선택한다.

상태 모니터 편집

편집하려는 상태 모니터 템플릿을 마우스 오른쪽 버튼 클릭하고 [상태 모니터 편집]을 선택하면, [상태 모니터 추가]와 동일한 인터페이스에서 편집할 수 있다.

상태 모니터 삭제

등록된 상태 모니터 템플릿을 삭제하려면, 해당 상태 모니터 템플릿을 마우스 오른쪽 버튼 클릭하고 [상태 모니터 삭제]를 선택한다.

상태 모니터 수행

수행하려는 상태 모니터 템플릿을 마우스 오른쪽 버튼 클릭하고 [상태 모니터 수행]을 선택하면, 선택한 템플릿에 따라 데이터베이스 서버와 브로커의 상태를 모니터링할 수 있는 상태 모니터링 창이 나타난다.

상태 모니터는 각 수집 항목별로 별도의 차트를 보여 주며, 현재 값, 최소 값, 최대 값 그리고 평균 값으로 구성되어 있다.

상태 모니터는 별도의 뷰로 구성되어 있어 다른 뷰 인터페이스와 동시에 볼 수 있으며, CUBRID 매니저 외부로 끌어내어 볼 수도 있으므로, 다중 모니터 환경에서 작업과 상태 모니터링을 동시에 할 때 편리하다.

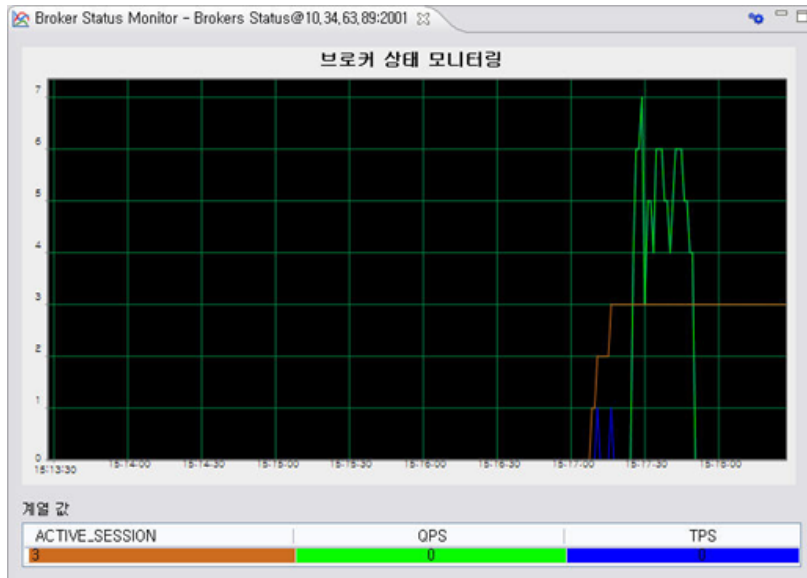


통합 상태 모니터

통합 상태 모니터링은 브로커 및 데이터베이스 모니터링 항목을 하나의 차트로 통합하여 확인할 수 있는 기능이다. CUBRID 2008 R2.2 이상 버전에서 지원한다.

브로커 상태 모니터

하나의 차트에서 브로커 관련 항목을 통합 모니터링할 수 있다. 브로커 상태 모니터링 창의 상단 바에 있는 [설정]을 클릭해서 모니터링 항목과 차트 속성을 설정할 수 있다.



데이터베이스 상태 모니터

하나의 차트에서 데이터베이스 관련 항목을 통합 모니터링할 수 있다. 모니터링할 수 있는 항목은 **cubrid statdump** 유틸리티 실행 시 출력되는 항목과 같다. 데이터베이스 상태 모니터링 창의 상단 바에 있는 [설정]을 클릭해서 모니터링 항목과 차트 속성을 설정할 수 있다.

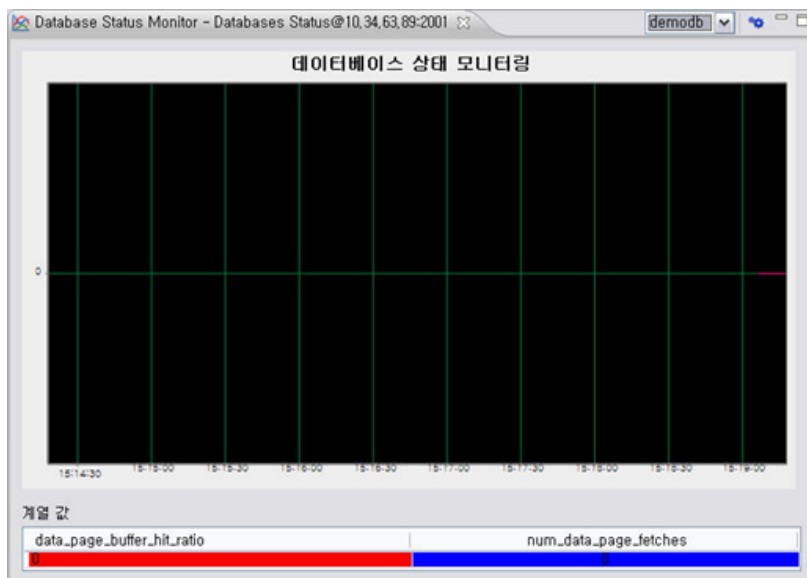
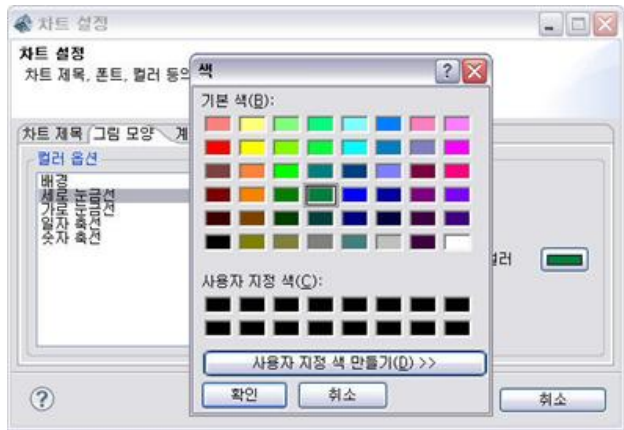


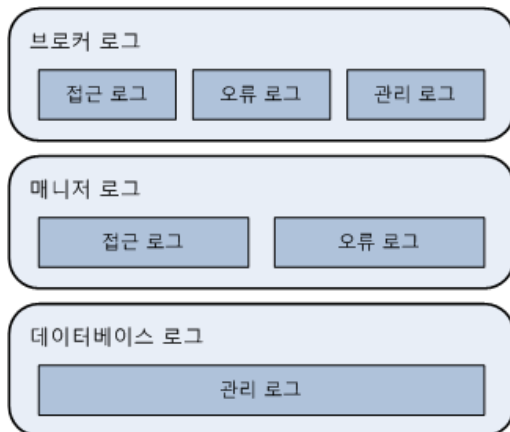
차트 속성 설정

차트 설정 마법사를 이용해서 차트 제목, 그림 모양, 모니터링 항목을 설정할 수 있다.



로그

로그는 브로커 로그, 매니저 로그, 데이터베이스 로그로 구성되어 있다. 로그 정보를 구성하는 것은 접근 로그, 오류 로그 그리고 관리 로그로 분류해 볼 수 있다. 각 로그는 다음과 같이 구성된다.



브로커 로그

접근 로그

접근 로그 파일은 응용 클라이언트 접속에 관한 정보를 기록하며, "broker_name.access"의 이름으로 저장된 것을 분석하여 출력한다. 또한, 브로커 환경 설정 파일에서 **LOG_BACKUP** 매개 변수가 "ON"으로 설정된 경우, 브로커의 구동이 정상적으로 종료되면 접속 로그 파일에 종료된 날짜와 시간 정보가 추가되어 로그 파일이 저장된다.

번호	인덱스	접속 IP	시작 시간	종료 시간	소요 시간	프로세스 ID
1	1	127.0.0.1	2009/06/26 09:32:32	2009/06/26 09:32:32	0:0:0	4436
2	1	127.0.0.1	2009/06/26 09:32:46	2009/06/26 09:32:46	0:0:0	4436
3	1	127.0.0.1	2009/06/26 09:35:46	2009/06/26 09:35:46	0:0:0	4436
4	1	127.0.0.1	2009/06/26 09:35:56	2009/06/26 09:35:56	0:0:0	4436
5	1	127.0.0.1	2009/06/26 10:25:05	2009/06/26 10:25:05	0:0:0	4436
6	1	127.0.0.1	2009/06/26 10:25:15	2009/06/26 10:25:15	0:0:0	4436
7	1	127.0.0.1	2009/06/26 10:25:25	2009/06/26 10:25:25	0:0:0	4436
8	2	127.0.0.1	2009/06/26 10:36:17	2009/06/26 10:36:17	0:0:0	7260
9	1	127.0.0.1	2009/06/26 10:36:27	2009/06/26 10:36:27	0:0:0	4436
10	2	127.0.0.1	2009/06/26 10:36:27	2009/06/26 10:36:27	0:0:0	7260
11	1	127.0.0.1	2009/06/26 13:48:55	2009/06/26 13:48:55	0:0:0	4436
12	1	127.0.0.1	2009/06/26 13:49:22	2009/06/26 13:49:22	0:0:0	4436
13	1	127.0.0.1	2009/06/26 13:49:36	2009/06/26 13:49:36	0:0:0	4436
14	1	127.0.0.1	2009/06/26 13:49:38	2009/06/26 13:49:38	0:0:0	4436
15	1	127.0.0.1	2009/06/26 13:49:44	2009/06/26 13:49:44	0:0:0	4436

오류 로그

오류 로그 파일은 응용 클라이언트의 요청을 처리하는 도중에 발생한 오류에 관한 정보를 기록하며, "broker_name_app_server_num.err"의 이름으로 저장된다.

번호	시간	오류 형식	오류 코드	트랜잭션 ID	오류 ID	오류 메시지
1	05/08/09 10:15:42.875	SYNTAX ERROR	-493	1	1	Syntax: Unknown class "participant". select *
2	05/14/09 10:21:00.468	ERROR	-191	-1	1	Cannot connect to server "demoth" on "C:\13080
3	05/14/09 10:21:00.468	ERROR	-677	-1	2	Failed to connect to database server, "demoth
4	05/14/09 10:21:00.484	ERROR	-191	-1	3	Cannot connect to server "demoth" on "C:\13080
5	05/14/09 10:21:00.484	ERROR	-677	-1	4	Failed to connect to database server, "demoth
6	05/14/09 10:21:05.859	ERROR	-191	-1	5	Cannot connect to server "demoth" on "C:\13080
7	05/14/09 10:21:05.859	ERROR	-677	-1	6	Failed to connect to database server, "demoth
8	05/14/09 10:21:05.875	ERROR	-191	-1	7	Cannot connect to server "demoth" on "C:\13080
9	05/14/09 10:21:05.875	ERROR	-677	-1	8	Failed to connect to database server, "demoth
10	05/14/09 10:21:11.453	ERROR	-191	-1	9	Cannot connect to server "demoth" on "C:\13080
11	05/14/09 10:21:11.453	ERROR	-677	-1	10	Failed to connect to database server, "demoth
12	05/14/09 10:21:11.468	ERROR	-191	-1	11	Cannot connect to server "demoth" on "C:\13080
13	05/14/09 10:21:11.468	ERROR	-677	-1	12	Failed to connect to database server, "demoth
14	05/14/09 13:35:52.531	ERROR	-191	-1	13	Cannot connect to server "demoth" on "C:\13080
15	05/14/09 13:35:52.546	ERROR	-677	-1	14	Failed to connect to database server, "demoth

다음은 오류 로그의 예와 설명이다.

Time: 02/04/09 13:45:17.687 - SYNTAX ERROR *** ERROR CODE = -493, Tran = 1, EID = 38
Syntax: Unknown class "unknown_tbl". select * from unknown_tbl

- **Time: 02/04/09 13:45:17.687** : 오류 발생 시각
- **SYNTAX ERROR** : 오류의 종류(SYNTAX ERROR, ERROR 등)
- ***** ERROR CODE = -493** : 에러 코드
- **Tran = 1** : 트랜잭션 ID. -1은 트랜잭션 ID를 할당받지 못한 경우.
- **EID = 38** : 오류 ID. SQL 문 처리 중 오류가 발생한 경우, 서버와 클라이언트 오류 로그가 관련이 있는 SQL 로그를 찾을 때 사용함.

관리 로그

서비스 구동 및 정지에 관한 이력을 관리한다.

번호	시간	상태
1	2009/06/26 09:07:22	start
2	2009/06/26 09:09:04	stop
3	2009/06/26 09:10:07	start

매니저 로그

접근 로그

접근 로그 파일은 매니저 접속에 관한 정보를 기록하며, 매니저 사용자 계정, 작업 수행 내역, 시간 정보를 확인할 수 있다.

오류 로그

오류 로그 파일은 매니저 접속에 대해 발생한 오류에 관한 정보를 기록한다.

데이터베이스 로그

관리 로그

데이터베이스 서버 구동에 관한 오류 정보를 기록하며, 이는 <database_name>_<date>_<time>.err 파일 내용을 출력한다.