CUBRID 활용
(질의 튜닝 사례 중심)

작성자: DBMS개발랩

목차

CUBRID SQL

타입, 연산자, 함수

CUBRID 식별자 비교

항목	CUBRID	MySQL
데이터베이스 이름	도메인 내에서 고유. 대소문자 구분 X	도메인 내에서 고유. 대소문자 구분 O.
테이블 이름	데이터베이스 내에서 고유. 대소문자 구분 X	데이터베이스 내에서 고유. 대소문자 구분 O.
컬럼 이름	테이블 내에서 고유 대소문자 구분 X	테이블 내에서 고유 대소문자 구분 X.
인덱스 이름	테이블 내에서 고유 대소문자 구분 X	테이블 내에서 고유 대소문자 구분 X.
식별자 작성 규칙	반드시 문자로 시작. 공백 포함 X 연산자로 사용되지 않는 특수 문자, 숫자를 포함할 수 있음.	공백 포함 X "₩", "/", "." 문자를 포함할 수 없음
예약어를 식별자로 사용	"" , backtick("`"), [ ]로 처리	"", backtick("`")로 처리
문자열 처리	싱글 따옴표(')로 처리 이중 따옴표(")로 처리(ansi_quotes=no)	이중 따옴표(")로 처리
주석 처리	--. //, /* */	
BLOB/CLOB 저장소	CUBRID2008 R3.1부터 지원 (11월 출시 예정)	다른 데이터 페이지에 저장된 데이터

CUBRID 타입 비교

MySQL Type Name	CUBRID Type	Bytes	설명
BIT(n)	BIT(n)		
BINARY(M)	BIT VARYING(n)		
VARBINARY(M)	BIT VARYING(n)		
BOOL, BOOLEAN	Not supported		BIT 또는 SMALLINT로 대체
숫자 타입[UNSIGNED]	Not Supported		
TINYINT	SMALLINT로 대체	2	TINYINT➔ 1byte, SMALLINT➔ 2byte
SMALLINT[(M)]	SMALLINT	2	
MEDIUMINT[(M)]	INT로 대체	4	MIDIUMINT➔ 3byte, INT➔ 4byte
INT, INTERGER INT(M)	INT, INTEGER INT로 대체	4	INT➔ 4byte
BIGINT[(M)]	BIGINT	8	BIGINT➔ 8byte
FLOAT(n)	FLOAT(n)	4	MySQL: 1<=n<=53 CUBRID: 1<=n<=38
DECIMAL(p,s) NUMERIC(p,s)	DECIMAL(p,s) NUMERIC(p,s)	16	MySQL: 1<=p<=65 , 0<=s<=30 CUBRID: 1<=p<=38 , 0<=s<=p
DOUBLE[(M,B)]	DOUBLE	8	

CUBRID 타입 비교

MySQL Type Name	CUBRID Type	Format	설명
DATE	DATE	'10/31/2008'	일 단위
DATETIME	DATETIME	'01:15:45.000 PM 10/31/2008'	밀리초 단위
TIMESTAMP[(M)]	TIMESTAMP	'01:15:45 PM 10/31/2008'	초 단위
TIME	TIME	'01:15:45 PM'	초 단위
YEAR[(2|4)]	Not Supported CHAR(4)로 대체		
CHAR(M)	CHAR(n)		
VARCHAR(M)	VARCHAR(n)		
TINYBLOB BLOB MEDIUMBLOB LONGBLOB	BIT VARYING(n)으로 대체		2008 R3.1에서 BLOB, CLOB 지원 예정
TINYTEXT TEXT MEDIUMTEXT LONGTEXT	VARCHAR(n)으로 대체		2008 R3.1에서 BLOB, CLOB 지원 예정
ENUM('value1','value2',...)	Not Support		
SET('value1','value2',...)	Not Support		

CUBRID Operators

MySQL Type	CUBRID Type	설명
AND, && OR, || XOR NOT, !	AND, && OR, || XOR NOT, !	논리 연산자 동일, R3.0 에서 지원 : &&, XOR, !
IS NOT NULL IS NOT IS NULL IS = <=> <>, != > >= < <=	IS NOT NULL IS NOT IS NULL IS = <=> <>, != > >= < <=	비교 연산자 동일
+ - * /, DIV %, MOD	+ - * /, DIV %, MOD	산술 연산자 동일
& | ^ ~ << >>	& | ^ ~ << >>	비트 연산자 동일, R3.0 에서 지원
REGEXP RLIKE	Not Supported	

CUBRID Functions(붉은색: 3.0에서 추가)

MySQL	CUBRID
ABS	ABS
ACOS	ACOS
ASCII	ASCII
ASIN	ASIN
ATAN	ATAN
ATN2	ATN2
AVG	AVG
BIT_LENGTH	BIT_LENGTH
CEILING	CEIL
CHAR	CHR
CHAR_LENGTH	CHAR_LENGTH
COALESCE	COALESCE
CONCAT	CONCAT
COS	COS
COT	COT
COUNT	COUNT
CURRENT_DATE	CURRENT_DATE
CURRENT_TIME	CURRENT_TIME
DATE_ADD	DATE_ADD
DATEDIFF	DATEDIFF
DECODE	DECODE
DEGREES	DEGREES
DIV	DIV
FIELD	FIELD

MySQL	CUBRID
FLOOR	FLOOR
FORMAT	FORMAT
GREATEST	GREATEST
GROUP_CONCAT	GROUP_CONCAT
IFNULL	IFNULL
INSTR	INSTR
LAST_DAY	LAST_DAY
LEAST	LEAST
LEFT	LEFT
LN	LN
LOAD_FILE	LOCATE
LOCATE	LOCATE
LOG	LOG
LOG10	LOG10
LOG2	LOG2
LOWER,LCASE	LOWER,LCASE
LPAD	LPAD
LTRIM	LTRIM
MAX	MAX
MID	MID
MIN	MIN
MOD	MOD
NOW	NOW
NULLIF	NULLIF
PI	PI
POWER	POWER
RADIANS	RADIANS

CUBRID Functions

MySQL	CUBRID
RAND	DRANDOM
REPLACE	REPLACE
REVERSE	REVERSE
RIGHT	RIGHT
ROUND	ROUND
RTRIM	RTRIM
SIGN	SIGN
SIN	SIN
SQRT	SQRT
STD	STDDEV
SUBSTRING	SUBSTRING
SUM	SUM
TAN	TAN
TRIM	TRIM
TRUNCATE	TRUNCATE
UCASE,UPPER	UCASE,UPPER
VARIANCE	VARIANCE

MySQL	CUBRID
CONV	Not Supported
CRC32	Not Supported
DATENAME	Not Supported
DAY	Not Supported
ELT	Not Supported
FIND_IN_SET	Not Supported
INSERT	Not Supported
MAKE_SET	Not Supported
MONTH	Not Supported
NCHAR	Not Supported
NOT REGEXP	Not Supported
REPEAT	Not Supported
REPLICATE	Not Supported
RLIKE	Not Supported
SPACE	Not SUpported
SQUARE	Not Supported
UNICODE	Not Supported
YEAR	Not Supported
Not Supported	INCR
Not Supported	MONTHS_BETWEEN
Not Supported	NVL2

다른 DBMS와의 차이

CLICK COUNTER

MySQL	해당 기능 없음
CUBRID	SELECT문에 대한 결과 레코드를 반환한 후 지정한 컬럼 값을 자동으로 증가시키는 기능. ✓SELECT_LIST에서 INCR(column) 함수를 지정하면, 결과 레코드 1개 조회한 후, 해당 컬럼 값을 1만큼 증가시킴. ✓WHERE 절에서 WITH INCREMENT FOR column 을 명시하면, 결과 레코드 1개 조회한 후, 해당 컬럼 값을 1만큼 증가시킴.

INCR( )

```
CREATE TABLE TABLE1(read_cnt INT, id INT, name VARCHAR);
INSERT INTO TABLE1 VALUES (1, 1, 'test');
SELECT INCR(read_cnt), id, name
FROM TABLE1 WHERE id=1; //read_cnt=1을 출력
SELECT INCR(read_cnt), id, name
FROM TABLE1 WHERE id=1; //read_cnt=2을 출력
```

WITH INCREMENT FOR

```
SELECT read_cnt, id, name
FROM TABLE1 WHERE id=1 WITH INCREMENT FOR read_cnt;  //read_cnt=3을 출력
```

AUTO_INCREMENT and SERIAL

MySQL	AUTO_INCREMENT를 사용하여 시퀀스값 생성 ✓트랜잭션이 롤백되면 이미 생성한 값이라도 재사용 가능. ✓테이블 내에서 1개의 AUTO_INCREMENT 컬럼만 사용 가능. ✓AUTO_INCREMENT 컬럼에 반드시 인덱스가 구성되어 있어야 함.
CUBRID	SERIAL 객체를 생성하거나, AUTO_INCREMENT를 사용하여 시퀀스값 생성 ✓트랜잭션이 롤백되더라도 이미 생성한 값은 재사용하지 않음(중복 값은 없으나, 불연속 가능) ✓SERIAL은 테이블과 별개의 객체이므로 여러 테이블에서 사용 가능. 메모리 캐시 기능 지원 ✓테이블 내에서 여러 개의 AUTO_INCREMENT 컬럼 사용 가능 ✓AUTO_INCREMENT 컬럼에 인덱스 구성할 필요 없음

SERIAL

```
CREATE SERIAL SERIAL_NAME START WITH 1 MAXVALUE 1000 NOCYCLE;
CREATE TABLE TABLE1(      seqnum INT,
                          name VARCHAR);
INSERT INTO TABLE1 VALUES (SERIAL_NAME.next_value, 'test'); //seqnum=1
```

AUTO_INCREMENT

```
CREATE TABLE TABLE1(      seqnum INT AUTO_INCREMENT(1,1000) NOT NULL,
                          name VARCHAR);
INSERT INTO TABLE1 (name) VALUES ('test'); //seqnum=1
```

자동형변환

CUBRID	데이터 타입에 대해 정확한 타입을 지정 해줘야 함. 컬럼의 데이터 타입 변환 연산자인 'CAST' 를 사용하여 변환이 필요함.

예)

TB3_C 테이블의 mm 컬럼이 integer 타입인 경우

```
SELECT aaa.b_id, aaa.c_id, bbb.d_id,
nvl(ccc.mm,'0') as mm
FROM TB1_A aaa, TB2_B bbb, TB3_C ccc          ( X )

SELECT aaa.b_id, aaa.c_id, bbb.d_id,
nvl(cast(ccc.mm as char(1)),'0') as mm        ( O )
FROM TB1_A aaa, TB2_B bbb, TB3_C ccc

SELECT aaa.b_id, aaa.c_id, bbb.d_id,
nvl(ccc.mm, 0) as mm                          ( O )
FROM TB1_A aaa, TB2_B bbb, TB3_C ccc
```

Distinct 사용

CUBRID	Distinct 사용시 order by 문과 같이 사용할 경우, 에러가 발생함.

예)

Select 절에 order by 컬럼이 없기 때문에 CUBRID에서는 order by column 을 select 절에 포함함.

아래의 예에서 원하는 결과는 distinct col_2 지만 실제로는 distinct col_2, col_ymd 가 되어 의도하지 않은 결과가 출력됨.

이는 SQL 표준에서는 의도하지 않음. 실제로 MySQL 에서만 실행이 되고, CUBRID, MS-SQL 은 오류 처리

```
SELECT DISTINCT col_2
FROM tbl_1
WHERE app_del_yn='N' AND col_1 = 'AAA'
ORDER BY col_ymd DESC ;

SELECT DISTINCT A.col_2
FROM(
SELECT col_ymd, col_2
FROM tbl_1
WHERE app_del_yn='N' AND col_1 = 'AAA'
ORDER BY col_ymd DESC ;
) A;
```

컬럼의 default 값으로 systimestamp를 쓸 경우

CUBRID	컬럼의 default 로 systimestamp . Sysdate, sysdatetime을 사용할 경우, Insert 가 수행될 때의 timestamp 가 아닌 테이블 생성 시간이 입력됨.

예)

☞ 테이블 생성

```
create table xtbl
( A int,
  B timestamp default systimestamp not null,
  C datetime default sysdatetime not null
)
```

☞ 테이블 정보

```
 〈Class Name〉
    xtbl

 〈Attributes〉

    a               INTEGER
    b               TIMESTAMP DEFAULT timestamp '11:48:50 AM 12/15/2009' NOT NULL
    c               DATETIME DEFAULT datetime '11:48:50.722 AM 12/15/2009 ' NOT NULL
```

CUBRID SQL 사용 시 주의 사항 (1)

✓ 자동 형 변환을 지원하지 않는다.
➔ 숫자형 데이터에 대해 따옴표를 처리할 수 없다.

✓ 문자셋(character set)을 지원하지 않는다.
➔ 응용 프로그램에서 설정한 문자 셋을 그대로 저장하고 출력한다.

✓ 멀티바이트 문자를 지원하지 않는다.
➔멀티바이트 문자인 경우, 이를 고려하여 컬럼 사이즈를 충분히 정의하여야 한다.
➔문자열 함수에서 길이(length) 또는 위치 값(position)은 바이트 단위로 처리된다.

✓ 데이터베이스 간 조인은 지원하지 않는다.

✓ ALTER TABLE문을 사용하여 컬럼 사이즈를 변경할 수 없다.
➔추후 버전에서 보완될 예정이다.

CUBRID SQL 사용 시 주의 사항 (2)

✓ 타입 바인딩 문제

?를 이용한 expression사용시 오류

```
select a,b from xoo where (? + ? * ? - ? + 1 / ? ) < 5
insert into foo values(? * ?)
```

타입을 잘못 인식해서 발생하는 오류

```
select sysdate + ? from db_root
```

→ 응용에서 expression 처리 후 단일값 바인딩

→ 명시적인 타입 캐스팅

```
select sysdate + cast(? as int) from db_root
```

✓ '' 과 NULL 은 다르다.

select nvl ('', 'null') from db_root;	select nvl (null,'null') from db_root;
nvl('', 'null') ========== ''	nvl(null, 'null') ========== 'null'

CUBRID SQL 사용 시 주의 사항 (3)

✓ HA 사용 시 주의사항

Primary Key 필수

→Primary Key가 있는 경우에만 데이터 동기화됨

Trigger 사용 제한

→Trigger 사용의 경우 슬레이브에서 중복 수행 가능

Method 사용 제한

→복제 로그를 생성하지 않는 mothod 사용 시 동기화 안됨

예) add_user()

JSP 사용시 주의사항

→마스터 및 슬레이브 모든 노드에 load 필요

CUBRID 쿼리 작성

Join Query

CUBRID	Inner Join, Outer Join(Full outer Join은 미지원), Cross Join, Self Join ✓ON 절에 조인 조건을 명시하는 ANSI-92 표준을 따라 질의 문을 작성하도록 권장함. ✓조인 대상 테이블에 동일한 이름의 컬럼이 존재하는 경우, select_list에 테이블 명을 반드시 명시하여야 한다. 명시하지 않으면 ERROR: Reference to id is ambiguous. 에러가 발생한다.

[Inner] Join

```
SELECT select_list
FROM TABLE1 T1
        INNER JOIN TABLE2 T2 ON T1.COL1 = T2.COL2
WHERE T1.A = 'test' AND T2.B = 1;
```

Left [Outer] Join

```
SELECT select_list
FROM TABLE1 T1
        LEFT OUTER JOIN TABLE2 T2 ON T1.COL1 = T2.COL2 AND T2.B=1
WHERE T1.A = 'test';
```

Pagination(LIMIT RESULT SET)

CUBRID	ROWNUM, ORDERBY_NUM(), LIMIT를 사용하여 페이지 처리 ✓ROWNUM<=n: 조건을 만족하는 결과 n개를 먼저 가져온 후, ORDER BY 또는 GROUP BY 적용 ✓ORDERBY_NUM() <=n : ORDER BY 한 결과에서 n개만 출력함 ✓LIMIT 1, n: 최종 결과에서 n개만 출력함

ROWNUM

```
SELECT select_list FROM TABLE1 T1
WHERE T1.A = 'test' AND ROWNUM <= 100
ORDER BY ORDER_COLUMN;
```

ORDERBY_NUM()

```
SELECT select_list FROM TABLE1 T1
WHERE T1.A = 'test'
ORDER BY ORDER_COLUMN
FOR ORDERBY_NUM() <= 100;
```

LIMIT (from R3.0)

```
SELECT select_list FROM TABLE1 T1
WHERE T1.A = 'test'
ORDER BY ORDER_COLUMN
LIMIT 1,100;
```

INDEX

MySQL	Key Filter 기능을 제공하지 않으므로 인덱스 성능이 낮음(InnoDB에 해당)
CUBRID	Key Filter 기능을 제공하므로 인덱스 성능이 높음, 불필요한 스캔이 발생하지 않음. -> WHERE 절 처리 시 인덱스 레벨에서 스캔 하지 않을 레코드를 미리 필터링하여 제외시킴 다중 컬럼 정렬 인덱스를 지원함(col1 ASC, col2 DESC)

```
CREATE INDEX on TABLE1(zipcode,name,address);
SELECT * FROM TABLE1
WHERE zipcode=1000 AND name LIKE '%test%' AND address LIKE '%seoul‘;
```

CUBRID 내부 동작 방식:
✓1단계: zipcode=1000인 대상을 인덱스 레벨에서 찾음
✓2단계: name, address 조건까지 만족하는 대상에 대해서 데이터 레벨로 접근하여 추출
(반면, MySQL은 zipcode=1000인 모든 대상에 대해서 데이터 레벨로 접근한 다음 나머지 조건을 만족하는 것들만 추출)

INDEX 사용 팁:
✓인덱스 키의 사이즈는 되도록 작게 설계하여야 성능에 유리하다.
✓분포도가 좋은 컬럼(좁은 범위), 기본 키, 조인의 연결 고리가 되는 컬럼에 대해 인덱스를 구성한다.
✓업데이트가 빈번하지 않은 컬럼으로 인덱스를 구성한다.

인덱스 정의와 USING INDEX 사용

```
CREATE [ UNIQUE ] INDEX [ index_name ]

ON table_name ( column_name[(prefix_length)] [ASC | DESC] [ {,
column_name[(prefix_length)] [ASC | DESC]} ...] ) [ ; ]
```

✓UNIQUE 인덱스는 유일성 제약 조건을 위한 인덱스를 생성한다.
✓인덱스 이름을 명시하지 않으면 자동으로 생성한다.
✓문자열의 앞부분에 대해서만 인덱스 정의할 수도 있다(Prefix Index)

```
SELECT/UPDATE/DELETE...USING INDEX {NONE | index_name[(+)],…};
```

✓인덱스 이름은 테이블 단위로 구분하며, table_name.index_name으로 사용한다.
✓USING INDEX절에서 지정한 인덱스 스캔 비용이 순차 스캔보다 작은 경우만 인덱스 스캔을 수행한다.
✓USING INDEX index_name(+)인 경우, 무조건 해당 인덱스 스캔을 수행한다.
✓USING INDEX NONE인 경우, 무조건 순차 스캔을 수행한다.
✓USING INDEX절 뒤에 두 개 이상의 인덱스 이름을 명시하면, 옵티마이저에 의해 적절한 인덱스가 선택된다.
✓두 개 이상의 테이블이 조인된 경우, 모든 테이블에 대해 사용해야 할 인덱스 이름을 명시하여야 한다.

CUBRID JDBC

SQL 타입과 Java 타입

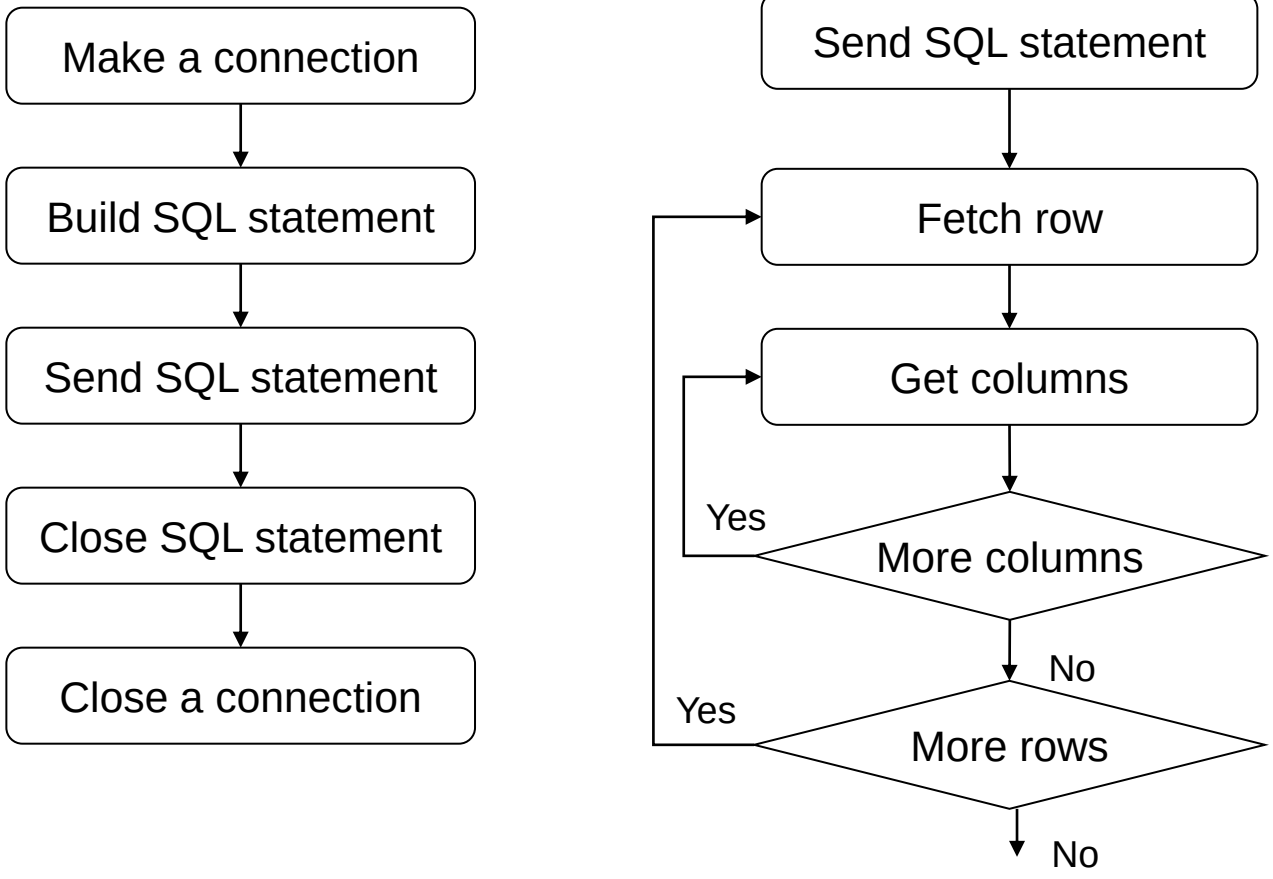
SQL Type	Java Type
CHAR, VARCHAR	java.lang.String, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.lang.Byte, java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float, java.lang.Double, java.math.BigDecimal, byte, short, int, long, float, double
NUMERIC, SHORT, INT, FLOAT, DOUBLE	java.lang.Byte, 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

JDBC 주요 인터페이스

✓**JDBC 2.0 표준 스펙을 지원한다.**

표준 인터페이스	설명
CallableStatement	SQL stored procedure를 지원하기 위한 preparedStatement의 확장된 인터페이스.
PreparedStatement	Precompiled SQL 구문을 위한 인터페이스
Connection	JDBC의 database Session. 세션을 위한 statement 객체 제공, statement의 transaction관리 디폴트로 auto-commit
DatabaseMetaData	connection 객체가 접속하고 있는 DB에 대한 정보 제공 JDBC ResultSet객체로 return
Driver	Driver 로드 시 자신의 instance 생성 후 DriverManager에 등록 DB에 접속하기 위해 DriverManager에서 URL을 받아 등록된 Driver중에서 해당 Driver로 연결
DriverManager	등록 JDBC드라이버 목록을 저장 초기화 시 driver속성에 지정된 모든 클래스 로드 getConnection() 에 의해 접속요청 시 원하는 Driver를 찾음
ResultSet	Database결과 집합. Database 질의에 대해 한 행씩 처리 현재 처리중의 행에 대한 포인터 소유
ResultSetMetaData	ResultSet의 컬럼 type과 속성에 관한 metadata 제공
Statement	Application이 database연동을 수행하기 위해 사용하는 embedded SQL문 표현 Statement 종료 시 ResultSet도 닫힘

JDBC 프로그래밍 흐름



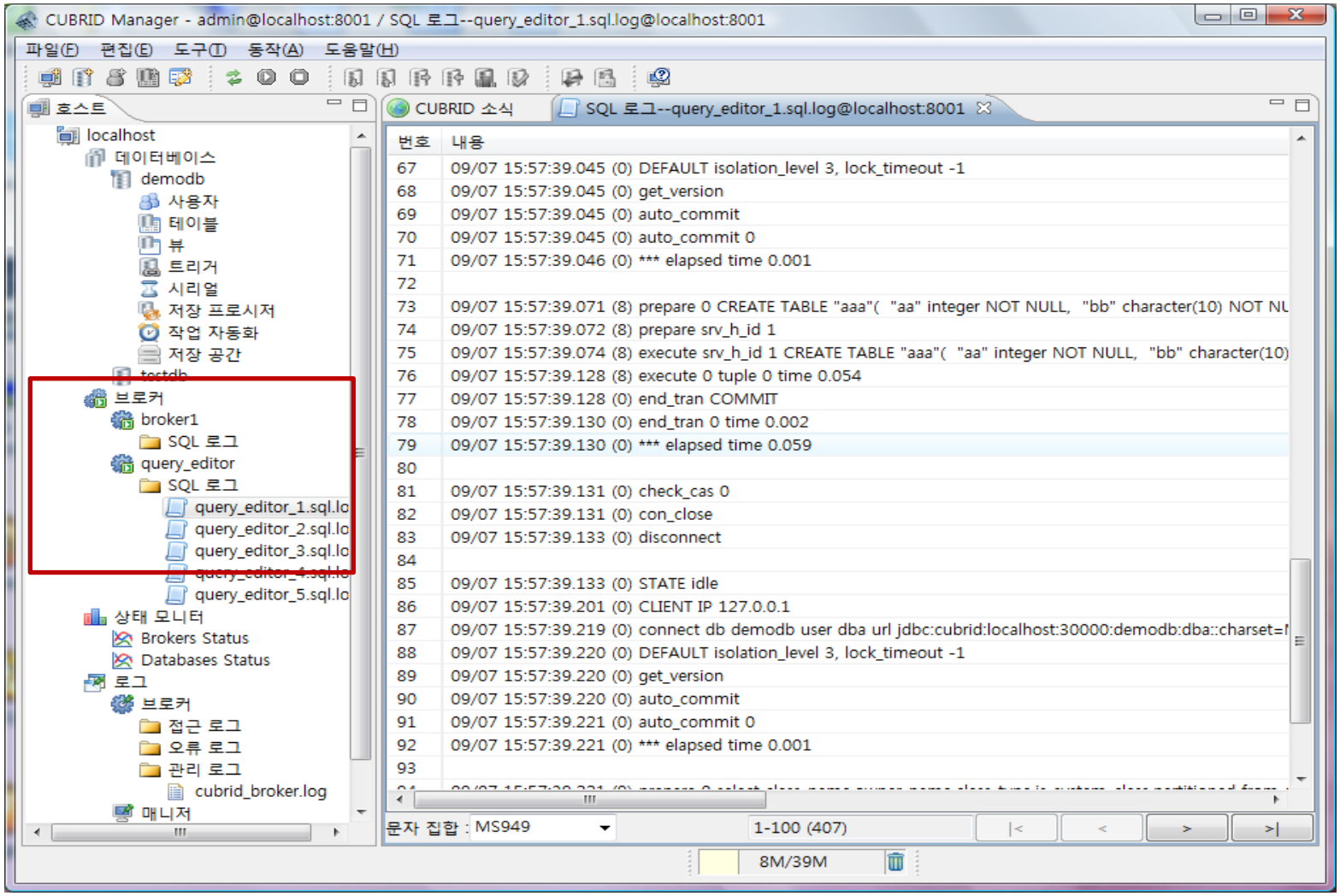
CUBRID JDBC 사용 시 주의 사항

- ✓ 자원 반환
  - ✓ ResultSet, Statement, Connection과 같은 데이터베이스 객체는 사용 후 반드시 반환
  - ✓ 반환은 해당 객체에 대해 close() 메소드를 호출함으로 수행
  - ✓ AutoCommit False로 사용할 경우 Connection에 대해 트랜잭션 종료 (Commit/Rollback)를 명시적으로 한 후 반환

- ✓ 중첩 질의문을 수행하는 경우, 각각 다른 Connection 객체를 할당하여 사용해야 함.
  - ✓ 조회한 데이터를 이용한 순환 문에서 다른 트랜잭션을 발생시킬 경우

- ✓ 사용 중인 Connection 객체에 대해 트랜잭션(Commit/Rollback)을 발생시킬 경우, 사용 중인 ResultSet이 종료됨

SQL Log 정보 (CUBRID 매니저)



SQL Log 정보

SQL 로그

- SQL 로그 파일은 응용 클라이언트가 요청하는 SQL을 기록하며, broker_name_app_server_num.sql.log라는 이름으로 저장된다.

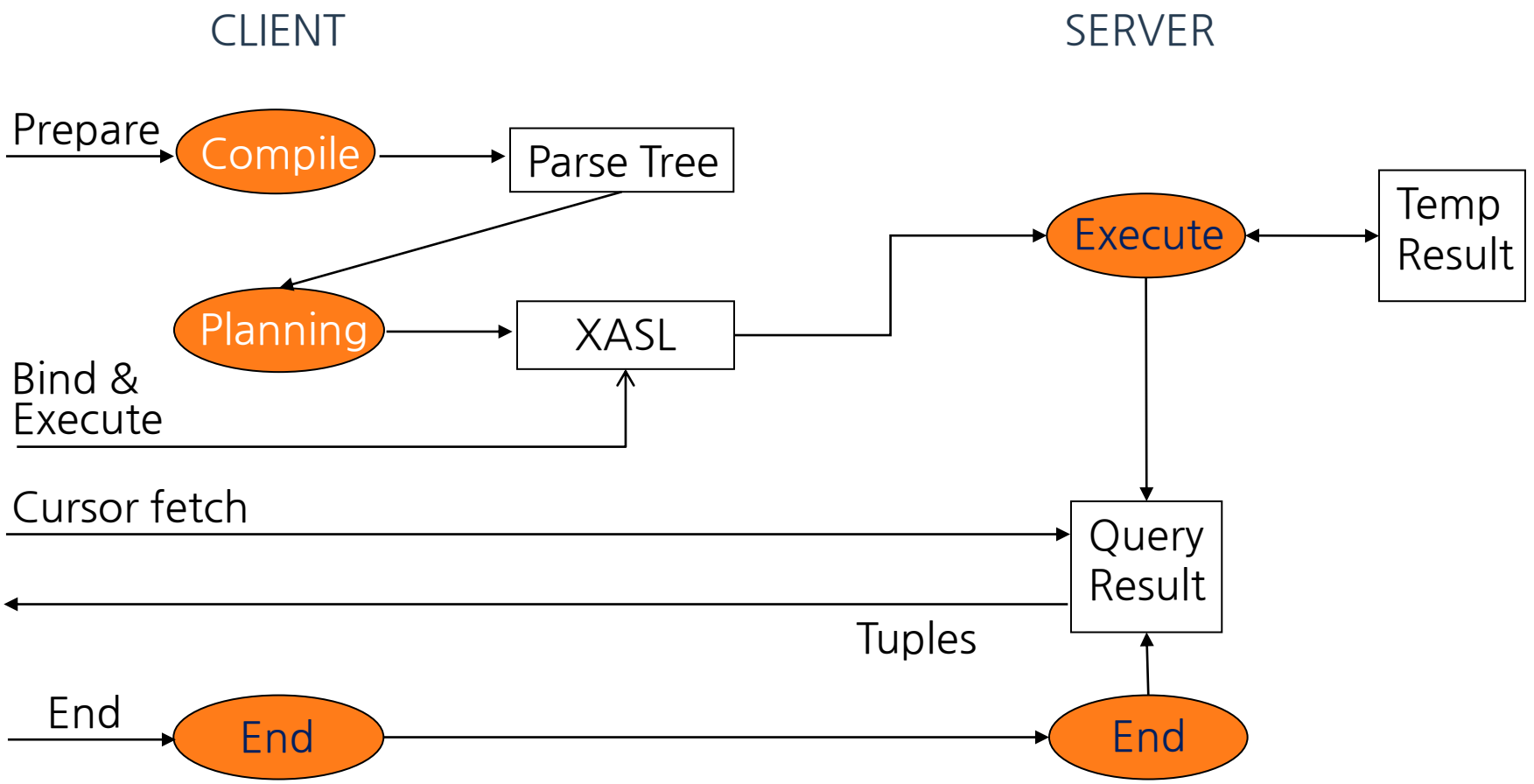
```
02/04 13:45:17.687 (38) prepare 0 insert into unique_tbl values (1)
02/04 13:45:17.687 (38) prepare srv_h_id 1
02/04 13:45:17.687 (38) execute srv_h_id 1 insert into unique_tbl values (1)
02/04 13:45:17.687 (38) execute error:-670 tuple 0 time 0.000, EID = 39
02/04 13:45:17.687 (0) auto_rollback
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
```

- 응용 클라이언트의 요청 시각
- (39) : SQL 문의 시퀀스 번호
- (PC) : 캐시된 플랜을 사용함
- SELECT... : 실행 SQL 문.
  WHERE 절의 binding 변수가 ?로 표시된다.
- execute 0 tuple 1 time 0.000
  - 1개의 row가 생성되고, 소요 시간은 0.000초
- auto_commit/auto_rollback
  - 자동으로 커밋 되거나, 롤백 되는 것을 의미
  - 두 번째 auto_commit/auto_rollback은
    에러 코드이며, 0은 에러가 없이 트랜잭션이 완료

CUBRID 질의 튜닝

질의 처리 과정

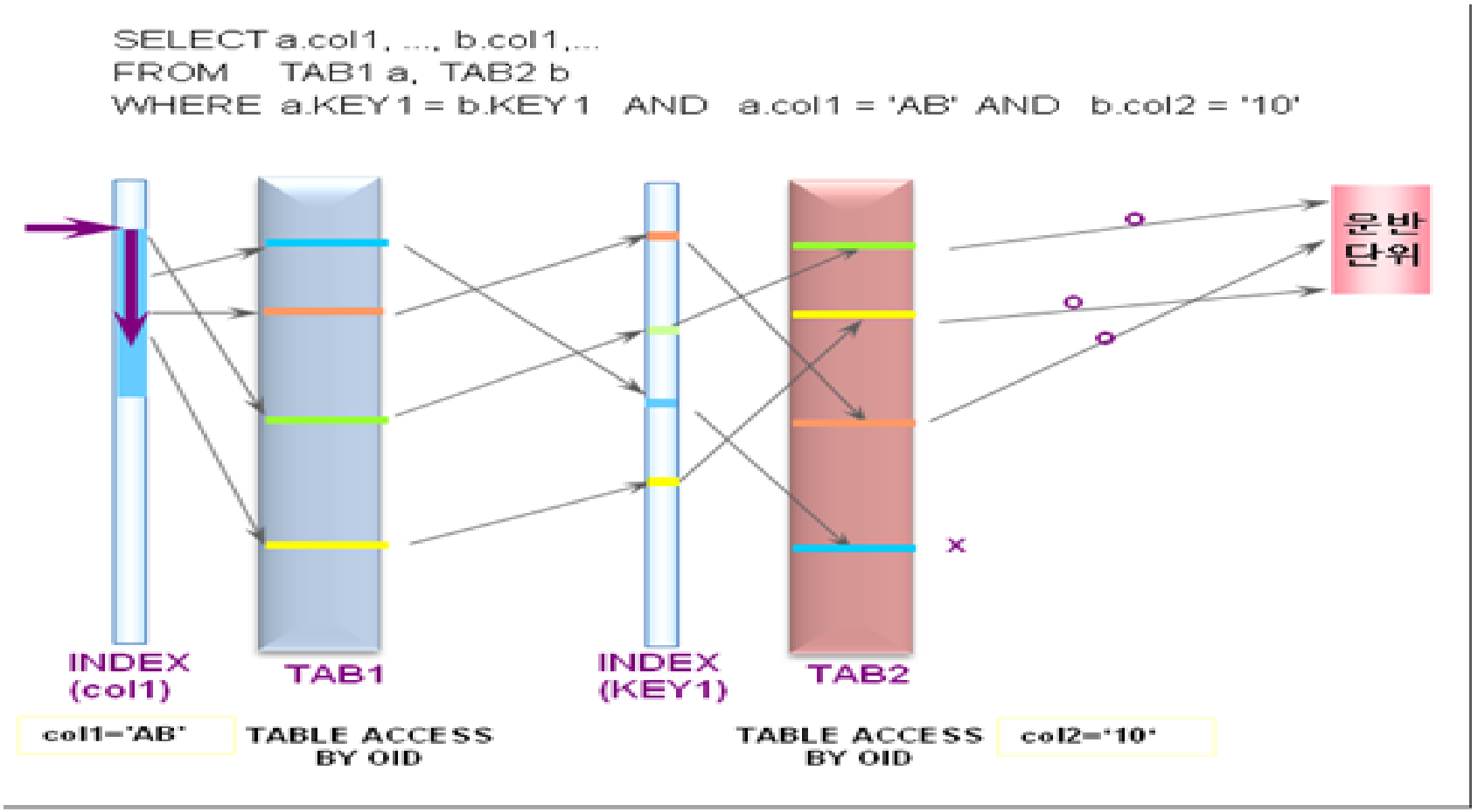

CLIENT
SERVER
Prepare
Compile
Parse Tree
Execute
Temp Result
Planning
XASL
Bind & Execute
Cursor fetch
Query Result
Tuples
End
End
End

스캔 방식

스캔방식	
Sequential scan	테이블의 모든 페이지를 읽는 방식
Temp(List File) scan	Derived Table (inline view) Temporary Results Sorting: distinct, union, order by, group by …
Index scan	Index page read + heap file access Indexable term Key ranges and filters

조인 방식

✓ Nested Loop Join (중첩 반복 조인)



조인 방식

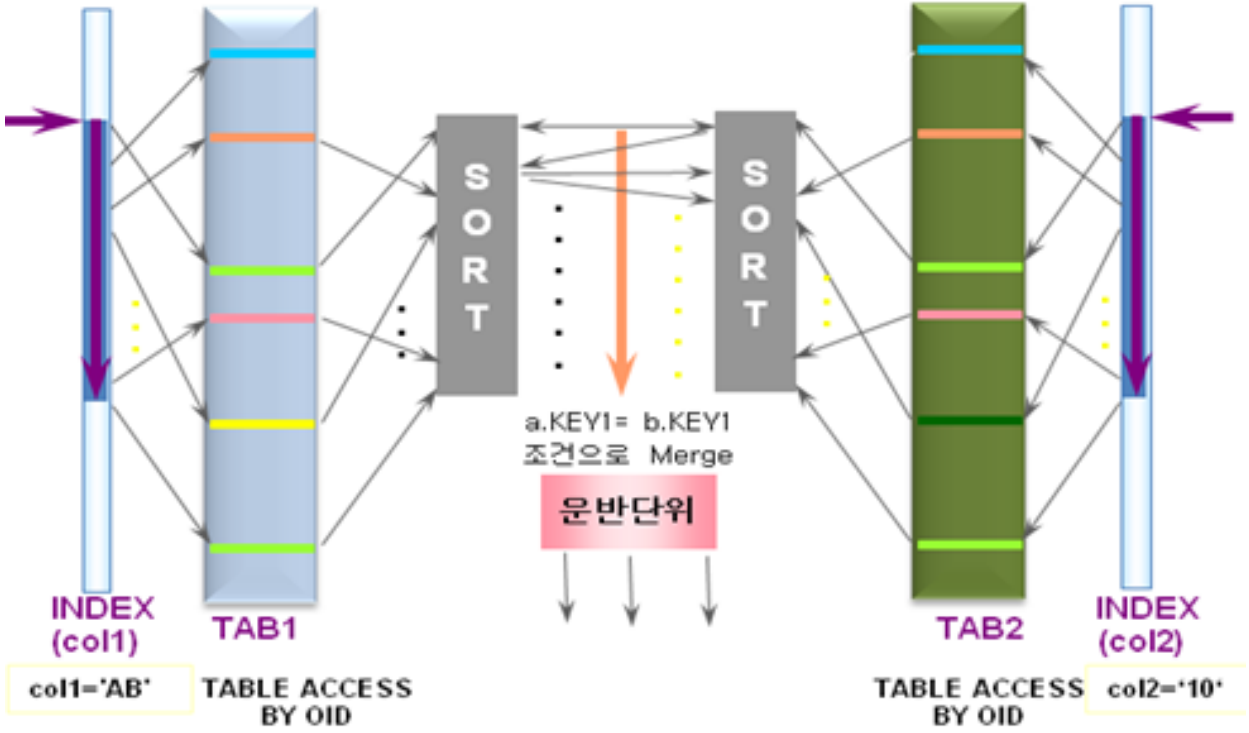
✓ Nested Loop Join의 특징

- 순차적 처리
- 종속적 (먼저 처리되는 테이블의 처리범위에 따라 처리량 결정)
- 랜덤(Random) I/O가 발생
- 연결고리 상태에 따라 성능 차이가 큼
- 주로 좁은 범위 처리에 유리

조인 방식

✓ Sort Merge Join (정렬 병합 조인)

```
SELECT /*+ use_merge */ a.col1, ..., b.col1,...
FROM   TAB1 a, TAB2 b
WHERE  a.KEY1 = b.KEY1  AND  a.col1 = 'AB'  AND  b.col2 = '10'
```



조인 방식

✓ Sort Merge Join 의 특징

- 동시적 처리

  양쪽 테이블을 동시에 읽고 양쪽테이블이 조인할 준비가 되었을 때 조인 시작

- 독립적

  선행테이블의 처리범위와 상관없이 각자 테이블의 필터링 조건이 성능에 영향을 미침

- 연결고리 보다 각자가 가지고 있는 필터링 조건에 따라 성능 차이가 큼

- 통계성 작업이나 배치 작업에 주로 사용됨.

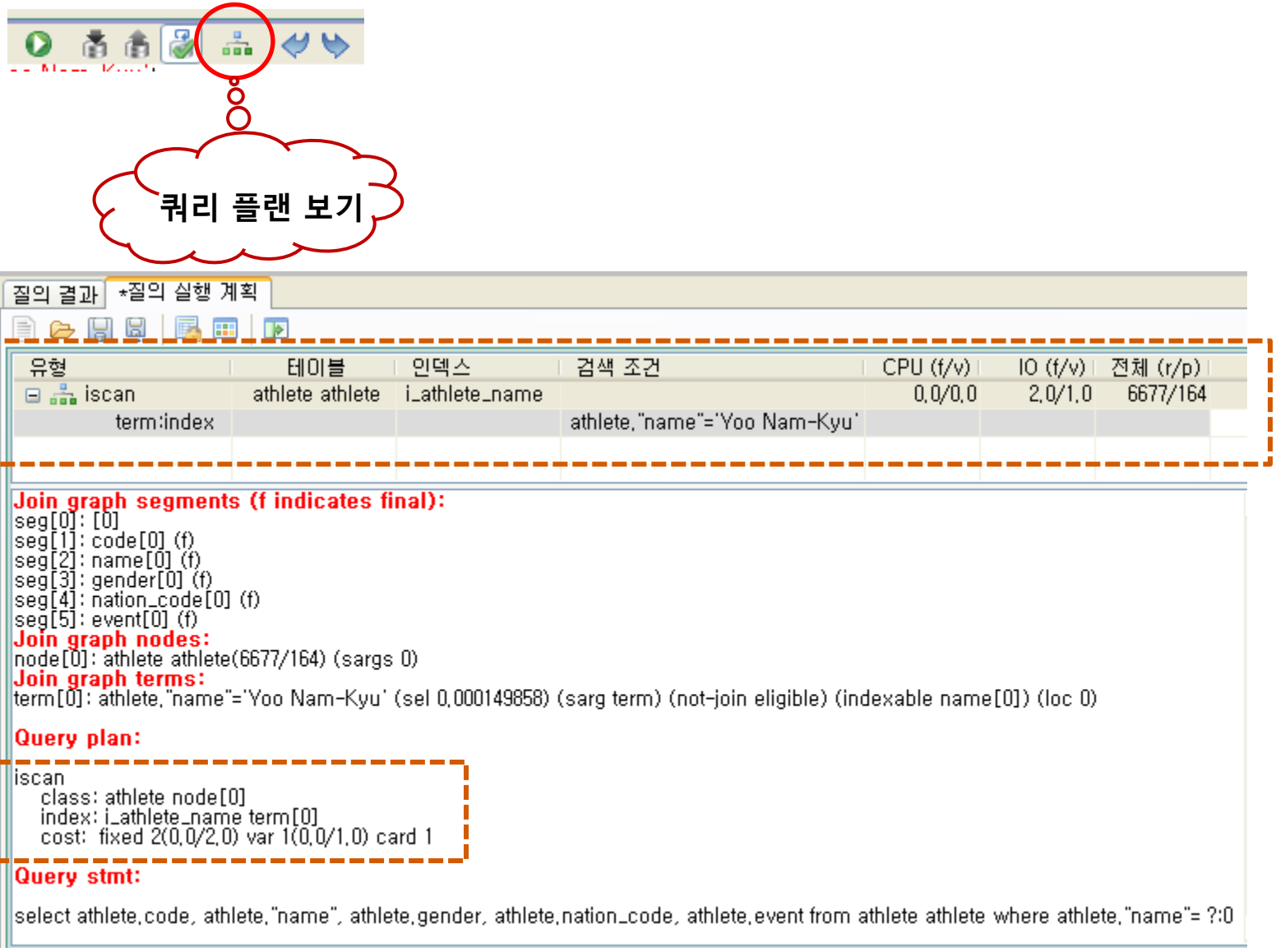
- 대량 Sort 작업시 Temp 를 사용하기 때문에 OLTP 성 업무에 적합하지 않음.

질의 계획과 힌트

✓스캔 방식(sscan, iscan)과 조인 방식(nl-join, idx-join, m-join)을 기반으로 질의 계획 생성

질의 계획	관련 힌트	설명
sscan	USING INDEX NONE	순차 스캔을 수행하여 실행 계획 생성
iscan	USING INDEX idx_name(+)	인덱스 스캔을 수행하여 실행 계획 생성
nl-join	/*+ USE_NL */	중첩 루프 조인 실행 계획 생성
idx-join	/*+ USE_IDX */	조인 키에 인덱스가 존재하는 경우 실행 계획 생성
m-join	/*+ USE_MERGE */	정렬 병합 조인 실행 계획 생성
	/*+ ORDERED */	FROM절에 명시된 테이블의 순서대로 조인하는 실행 계획 생성

질의 계획 보기 설정과 확인(CUBRID 매니저)



질의 계획 보기 예제(sscan)

```
SELECT * FROM athlete WHERE name='Yoo Nam-Kyu';
```

(card, page#)

유형	테이블	인덱스	검색 조건	CPU (f/v)	IO (f/v)	전체 (r/p)
sscan	athlete athlete			0.0/16.7	0.0/164.0	6677/164
term:select			athlete."name"='Yoo Nam-Kyu'			

```
Join graph segments (f indicates final):
seg[0]: [0]
seg[1]: code[0] (f)
seg[2]: name[0] (f)
seg[3]: gender[0] (f)
seg[4]: nation_code[0] (f)
seg[5]: event[0] (f)
Join graph nodes:
node[0]: athlete athlete(6677/164) (sargs 0)
Join graph terms:
term[0]: athlete."name"='Yoo Nam-Kyu' (sel 0.001) (sarg term) (not-join eligible) (loc 0)

Query plan:

sscan
    class: athlete node[0]
    sargs: term[0]
    cost:  fixed 0(0.0/0.0) var 181(16.7/164.0) card 7
Query stmt:

select athlete.code, athlete."name", athlete.gender, athlete.nation_code, athlete.event from athlete athlete where athlete."name"= ?:0
```

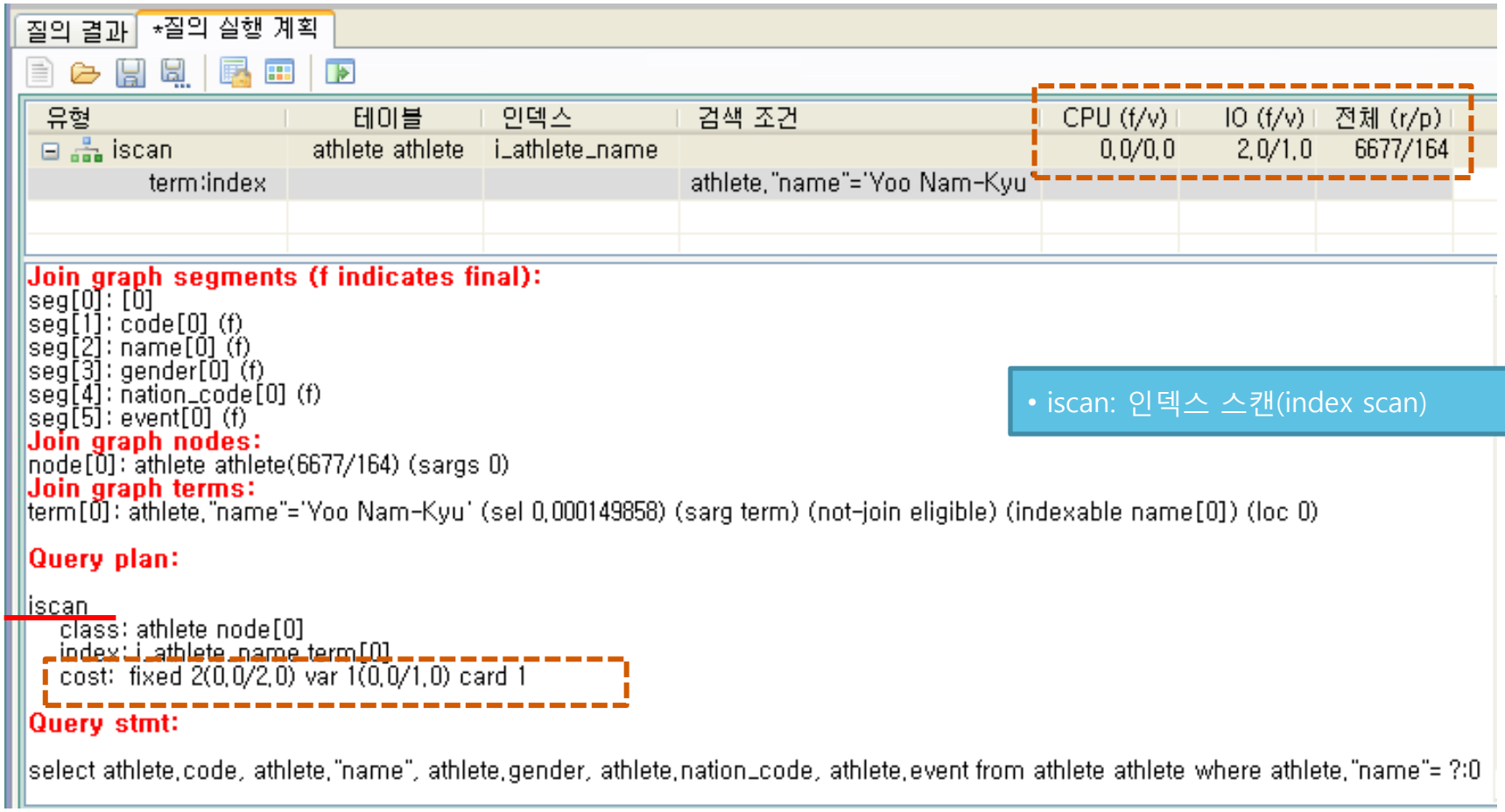
(card, page#)

sel

- sscan: 순차 스캔(sequential scan)
- card: 예상 결과 집합의 레코드 수
- page#: 예상 페이지 접근 수
- sel(selectivity): 검색 조건을 만족하는 예상 선택도

질의 계획 보기 예제(iscan)

```
CREATE INDEX ON athlete(name);
SELECT * FROM athlete WHERE name='Yoo Nam-Kyu';
```



질의 계획 보기 예제(nl-join)

```
SELECT * FROM olympic, nation WHERE olympic.host_nation=nation.name;
```

유형	테이블	인덱스	검색 조건	CPU (f/v)	IO (f/v)	전체 (r/p)
nl-join (inner join)				0.0/13.6	0.0/16.0	
term:join			olympic.host_nation=nation."name"			
sscan	olympic olympic			0.0/0.1	0.0/4.0	25/4
sscan	nation nation			0.0/0.5	0.0/6.0	215/6
term:select			olympic.host_nation=nation."name"			

Query plan:

```
nl-join (inner join)
   edge:  term[0]
   outer: sscan
            class: olympic node[0]
            cost:  fixed 0(0.0/0.0) var 4(0.1/4.0) card 25
   inner: sscan
            class: nation node[1]
            sargs: term[0]
            cost:  fixed 0(0.0/0.0) var 7(0.5/6.0) card 215
   cost:  fixed 0(0.0/0.0) var 30(13.6/16.0) card 5
```

Query stmt:

```
select olympic.host_year, olympic.host_nation, olympic.host_city, olympic.opening_date, olympic.closing_date,
olympic.mascot, olympic.slogan, olympic.introduction, nation.code, nation."name", nation.continent, nation.capital from
olympic olympic, nation nation where olympic.host_nation=nation."name"
```

✓**outer table: 레코드 수가 적은 것**
✓**inner table: 레코드 수가 많고, 인덱스가 있는 것**

질의 계획 보기 예제(idx-join)

```
SELECT * FROM game, athlete WHERE game.athlete_code=athlete.code;
```

유형	테이블	인덱스	검색 조건	CPU (f/v)	IO (f/v)	전체 (r/p)
idx-join (inner join)				0.0/43.3	3.0/294.0	
sscan	game game			0.0/21.6	0.0/147.0	8653/147
iscan	athlete athlete	pk_athlete_code		0.0/0.0	3.0/1.0	6677/164
term:index			game.athlete_code=athlete.code			

term[0]: game.athlete_code=athlete.code (sel 0.000149768) (join term) (mergeable) (inner-join) (indexable athlete_code[0] code[1]) (loc 0)

Query plan:

```
idx-join (inner join)
   outer: sscan
            class: game node[0]
            cost:  fixed 0(0.0/0.0) var 169(21.6/147.0) card 8653
   inner: iscan
            class: athlete node[1]
            index: pk_athlete_code term[0]
            cost:  fixed 3(0.0/3.0) var 1(0.0/1.0) card 6677
   cost:  fixed 3(0.0/3.0) var 337(43.3/294.0) card 8653
```

Query stmt:

```
select game.host_year, game.event_code, game.athlete_code, game.stadium_code, game.nation_code, game.medal,
game.game_date, athlete.code, athlete."name", athlete.gender, athlete.nation_code, athlete.event from game game, athlete athlete
where game.athlete_code=athlete.code
```

질의 계획 보기 예제(m-join)

```
SELECT /*+ USE_MERGE */
        * FROM game, athlete WHERE game.athlete_code=athlete.code;
```

유형	테이블	인덱스	검색 조건	CPU (f/v)	IO (f/v)	전체 (r/p)
temp (athlete_code)				36201.7/21.6	589.0/219.7	
m-join (inner join)				48.3/36148.4	375.5/213.4	
term:join			game.athlete_code=athlete.code			
temp (athlete_code)				26.6/21.6	190.3/135.2	
sscan	game game			0.0/21.6	0.0/147.0	8653/147
temp (code)				21.7/16.7	185.2/78.2	
sscan	athlete athlete			0.0/16.7	0.0/164.0	6677/164

Query plan:

```
temp
    order: athlete_code[0]
    subplan: m-join (inner join)
                edge:  term[0]
                outer: temp
                        order: athlete_code[0]
                        subplan: sscan
                                class: game node[0]
                                cost:  fixed 0(0.0/0.0) var 169(21.6/147.0) card 8653
                        cost:  fixed 217(26.6/190.3) var 157(21.6/135.2) card 8653
                inner: temp
                        order: code[1]
                        subplan: sscan
                                class: athlete node[1]
                                cost:  fixed 0(0.0/0.0) var 181(16.7/164.0) card 6677
                        cost:  fixed 207(21.7/185.2) var 95(16.7/78.2) card 6677
                cost:  fixed 424(48.3/375.5) var 36362(36148.4/213.4) card 8653
    cost:  fixed 36791(36201.7/589.0) var 241(21.6/219.7) card 8653
```

인덱스 정의와 USING INDEX 사용 - 튜닝

✓**인덱스 정의 시 질의 계획을 확인하면서 Index를 구성**

튜닝 전	튜닝 후
CREATE INDEX idx1 ON userinfo(phone); SELECT COUNT(*) FROM userinfo WHERE allow_search=1 AND phone='02';	CREATE INDEX idx1 ON userinfo(phone, allow_search); SELECT COUNT(*) FROM userinfo WHERE allow_search=1 AND phone='02';

✓**인덱스 컬럼 값과 NULL을 비교하는 경우 인덱스 스캔 X ➔ 질의 수정**

튜닝 전	튜닝 후
SELECT name,email_addr FROM student WHERE email_addr IS NOT NULL;	SELECT name,email_addr FROM student WHERE email_addr >= '';

✓**검색 조건을 Covering 하도록 인덱스 생성할 것**
✓**ORDER BY 정렬 조건을 Covering 하도록 인덱스 생성할 것**
✓**동적 파라미터를 바인딩하여 LIKE 검색하는 경우, 인덱스 스캔을 못하므로 주의**

```
SELECT * FROM tbl WERE col1 LIKE ? || '%' //순차 스캔 발생
➔SELECT * FROM tbl WHERE col1 LIKE 'AAA' || '%' //static 값 입력
```

인덱스 정의와 USING INDEX 사용 - 튜닝

✓**WHERE절에서 인덱스 컬럼(yymm)을 함수로 가공하면 인덱스 스캔 X**

튜닝 전	튜닝 후

`SELECT student_id`
`FROM record`
`WHERE substring(yymm, 1, 4) = '1997';`
	`SELECT student_id`
	`FROM record`
	`WHERE yymm`
	`BETWEEN '199701' AND '199712';`

질의 튜닝 참고

No Silver Bullet!

✓JOIN시 자주 사용되는 컬럼은 인덱스로 등록한다.

✓되도록 동등(=) 비교를 사용한다.

✓단일 인덱스 여러 개 보다는 multi-column index를 생성을 고려한다.

✓인덱스를 많이 생성하는 것은 INSERT/UPDATE/DELETE 성능 저하의 원인이 될 수 있다.

✓WHERE 절에서 자주 사용되는 컬럼에는 인덱스 추가를 고려한다.

✓동일한 값을 가지는 레코드가 적은 컬럼에 인덱스를 설정한다.

✓인덱스 SCAN이 FULL TABLE SCAN보다 항상 빠르지는 않다. (보통 selectivity가 5~10% 이내인 경우, 인덱스 SCAN이 우수함)

✓JOIN의 대상이 되는 테이블의 개수를 줄인다.

질의 튜닝 참고

✓ **데이터가 적은 테이블이 outer 로 join 할 수 있도록 한다.**
예) 10건이 있는 small 테이블과, 1010건이 있는 large 테이블과의 조인 순서 비교

	small 테이블이 outer 일 경우	Large 테이블이 outer 일 경우
Plan	Join graph nodes: node[0]: small small(10/1) node[1]: large large(1010/2) Join graph equivalence classes: eqclass[0]: i[0] i[1] Join graph edges: term[0]: small.i=large.i (sel 0.000990099) (join term) (mergeable) (inner-join) (indexable i[0] i[1]) (loc 0) Query plan: idx-join (inner join) outer: sscan class: small node[0] cost: fixed 0(0.0/0.0) var 1(0.0/1.0) card 10 inner: iscan class: large node[1] index: i_large_i term[0] cost: fixed 2(0.0/2.0) var 1(0.0/1.0) card 1010 cost: fixed 2(0.0/2.0) var 2(0.1/2.0) card 10 SQL statement execution time: 0.007128 sec	Join graph nodes: node[0]: large large(1010/2) node[1]: small small(10/1) Join graph equivalence classes: eqclass[0]: i[0] i[1] Join graph edges: term[0]: small.i=large.i (sel 0.000990099) (join term) (mergeable) (inner-join) (indexable i[1] i[0]) (loc 0) Query plan: idx-join (inner join) outer: sscan class: large node[0] cost: fixed 0(0.0/0.0) var 5(2.5/2.0) card 1010 inner: iscan class: small node[1] index: i_small_i term[0] cost: fixed 0(0.0/0.0) var 1(0.0/1.0) card 10 cost: fixed 0(0.0/0.0) var 9(5.1/4.0) card 10 SQL statement execution time: 0.009670 sec
Fetch	Num_data_page_fetches = 55	Num_data_page_fetches = 3056

질의 튜닝 예

질의 튜닝 예 (1)

✓ **Pagenation 질의 (테이블의 데이터가 커질 경우, ASC , DESC 인덱스를 혼용하여 처리)**

원인 : 페이지 처리시 정렬 비용을 줄이기 위해 인덱스를 사용하는데,
뒷 페이지를 조회할 수록 성능이 나빠 지는 현상 (예) 러시아 페인트 공 문제)

정 방향 인덱스 – 원래 정렬하고자 하는 정렬 순서로 생성한 인덱스
역 방향 인덱스 – 원래 정렬하고자 하는 정렬 순서의 역순으로 생성한 인덱스

예) 페이지당 10건의 데이터를 출력할 경우, 전체 데이터가 100 개이면, 총 10페이지가 생성됨.
1~5 페이지는 정 방향 인덱스를 사용, 6~10 페이지는 역 방향 인덱스를 사용하여 나온 결과를 정렬하고자 하는 컬럼으로 재 정렬 수행..

아래는 기본적으로 사용하는 페이지 처리 질의임. 아래의 질의를 위의 방식으로 변경.

1~5 페이지 검색	SELECT * FROM aaa using index idx_aaa_asc Order by a asc for orderby_num() BETWEEN ((찾는 페이지 수 - 1) * 1페이지당 개수) + 1) AND (찾는페이지 수 * 1 페이지당 개수) ;

질의 튜닝 예 (1)

✓ **Pagenation 질의 (계속)**

6~10 페이지	Select * from (SELECT * FROM aaa using index idx_aaa_desc Order by a desc for orderby_num() BETWEEN (전체 data 개수 - ((찾는 페이지 수 * 1페이지당 개수) -1) AND (전체 data 개수 - ((찾는 페이지 수 - 1) * 페이지당 개수)) t order by t.a asc

질의 튜닝 예 (2)

✓ **정렬이 필요한 질의의 경우 index 를 사용하여 정렬비용을 줄이자.**

정렬이 필요한 컬럼 이 index 로 생성되어있을 경우, 질의 최적화 단계에서 order by 를 Skip 하여 성능이 향상됨.

예) 게시판에서 최근에 등록된 게시글 10개를 출력

튜닝 전	SELECT * FROM tbl_1 a INNER JOIN tbl_2 b ON a.num = b.num WHERE a.col_1 = 'AAA' AND a.col_2 = 'bbb' ORDER BY a.col_3 DESC FOR ORDERBY_NUM() BETWEEN 1 AND 10
튜닝 후	create index i_index_name on tbl_1 (col_1, col_2, col_3 desc); SELECT * FROM tbl_1 a INNER JOIN tbl_2 b ON a.num = b.num WHERE a.col_1 = 'AAA' AND a.col_2 = 'bbb' Using index i_index_name(+) ORDER BY a.col_1, a.col_2 , a.col_3 DESC FOR ORDERBY_NUM() BETWEEN 1 AND 10

질의 튜닝 예 (3)

✓ **where 조건이 없는 테이블의 min, max**

min, max 의 컬럼이 단일 Index 컬럼 으로 생성되어있을 경우에, Min, Max 를 바로 반환하여 성능을 높일 수 있음.

예) where 조건이 없는 테이블의 가장 최근 날짜를 가져오는 질의

튜닝 전

```
SELECT col_ymd
FROM tbl_5
ORDER BY col_ymd DESC
FOR ORDERBY_NUM() = 1

Query Plan
temp(order by)
    subplan: sscan
                class: tbl_5 node[0]
                cost:  fixed 0(0.0/0.0) var 192(9.9/182.0) card 3977
    sort:  1 desc
    cost:  fixed 197(14.9/182.0) var 29(9.9/19.4) card 3977
```

튜닝 후

```
create index on tbl_5 ( col_ymd desc );
select max(col_ymd) from tbl_5

sscan
    class: tbl_5 node[0]
    cost:  fixed 0(0.0/0.0) var 192(9.9/182.0) card 3977
```

질의 튜닝 예 (4)

✓ **where 조건이 없는 테이블의 count (*)**

보통은 해당 테이블을 Sscan 하여 튜플을 count 함.
CUBRID 에서는 해당 테이블에 Primary Key 가 존재하면, Primary Key Index Tree 의 Root 페이지에 전체 count 개수를 반환하여 성능을 높일 수 있음.

예) 게시판에 등록된 전체 글 개수를 반환

튜닝 전	`SELECT COUNT(*) FROM tbl_2`
튜닝 후	테이블에 Primary Key 생성. `SELECT COUNT(*) FROM tbl_2;`

질의 튜닝 예 (5)

✓ **Like 질의 사용시 index 스캔 유무 확인**

Like ? || '%' 패턴의 경우, 해당 컬럼에 인덱스가 생성되어 있어도 실제 바인딩 되는 ? 의 값을 모르기 때문에 Sscan 수행하게 됨.

튜닝 전	`SELECT * FROM tbl WHERE col1 LIKE ? || '%'`
튜닝 후	1) Like 질의에 대해 ? 대신 직접 static 값으로 수정 Select * from tbl where col1 Like 'AAA' || '%' 2) Query Optimizer 가 rewrite 하는 방법으로 수정 Select * from tbl where col1 >= 'AAA' and col < 'AAB' Select * from tbl where col1 >= '' and col1 like '%S%';

질의 튜닝 예 (6)

✓ **테이블이 커짐에 따라 Join 방식이 변경되어 성능 튜닝**

기존의 테이블이 작아서 MERGE Join 으로 도 성능에 영향을 끼치지 않았으나, 테이블의 데이터가 많아 짐에 따라 Merge Join 이 불리해짐.

튜닝 전	SELECT count(i.id) FROM tbl_7 i LEFT OUTER JOIN tbl_8 k ON k.id = i.id WHERE 1 = 1 AND i.col_ymd BETWEEN TO_TIMESTAMP('20101011000000', 'YYYYMMDDHH24MISS') AND TO_TIMESTAMP('20101011235959', 'YYYYMMDDHH24MISS') AND i.col_no NOT IN (729,730,731,732,733,734,617,547,548,703,704,705,700,701,702,388)
튜닝 후	SELECT /*+ USE_IDX */ count(i.id) FROM tbl_7 i LEFT OUTER JOIN tbl_8 k ON k.id = i.id WHERE 1 = 1 AND i.col_ymd BETWEEN TO_TIMESTAMP('20101011000000', 'YYYYMMDDHH24MISS') AND TO_TIMESTAMP('20101011235959', 'YYYYMMDDHH24MISS') AND i.col_no NOT IN (729,730,731,732,733,734,617,547,548,703,704,705,700,701,702,388)

질의 튜닝 예 (6)

- ✓ **Merge Join 으로 수행하지 않고, Index Join 으로 수행하도록 Hint 를 주어서 성능을 향상시킴.**

튜닝 전	Num_data_page_fetches = 282437 Num_data_page_dirties = 1151818 count(i.id) ========== 3088 1 rows selected. SQL statement execution time: 3.058376 sec
튜닝 후	Num_data_page_fetches = 18803 Num_data_page_dirties = 0 count(i.id) ========== 3088 1 rows selected. SQL statement execution time: 0.103381 sec

질의 튜닝 예 (7)

✓ **Select 절의 subquery 호출 횟수를 줄이도록**

튜닝 전

```
SELECT *
FROM
    ( SELECT  col_no     ,  col_avg,  col_cnt    ,
              (  (SELECT COUNT(*) FROM    tbl_1
                  WHERE   col_no = tbl_9.num
                  AND     col_ymd
                  BETWEEN '201009300000' AND '20100930235959'
                   AND     col_1 = 'BBB'
                  ) * 3
                  +
                 (SELECT COUNT(*) FROM tbl_1
                  WHERE   col_no = tbl_9.num
                  AND     col_3
                  BETWEEN '20100930000000' AND '20100930235959'
                  AND     col_1 = 'BBB'
                  ) * 7 )
             t0
       FROM     tbl_9
       ORDER BY t0 DESC FOR ORDERBY_NUM() <= 60 * 3
       )
       tbl_9
       JOIN tbl_2
       ON     tbl_9.num = tbl_2.num
WHERE  tbl_2.col_cd        = 'AAA'
AND    tbl_2.col_yn  = 'Y'
AND    ROWNUM <= 60
```

질의 튜닝 예 (7)

튜닝 후

```
SELECT *
FROM
    ( SELECT  col_no     ,  col_avg,  col_cnt    ,
              NVL(t1.cnt,0) + NVL(t2.cnt,0) AS t0
      FROM     tbl_9 t9
               LEFT OUTER JOIN
                 (SELECT col_no, COUNT(*) * 3 as cnt
                  FROM    tbl_1
                  WHERE col_ymd
                  BETWEEN '201009300000' AND '20100930235959'
                   AND     col_1 = 'BBB'
                   group by col_no
                  ) t1 on t9.num = t1.num
                LEFT OUTER JOIN
                 (SELECT col_no, COUNT(*) * 7 as cnt
                  FROM tbl_1
                  WHERE col_3
                  BETWEEN '20100930000000' AND '20100930235959'
                  AND     col_1 = 'BBB'
                  group by col_no
                ) t2 on tbl_2_no = t2.num
      ORDER BY t0 DESC FOR ORDERBY_NUM() <= 60 * 3
     )
       tbl_9
       JOIN tbl_2
       ON     tbl_9.num = tbl_2.num
WHERE  tbl_2.col_cd        = 'AAA'
AND    tbl_2.col_yn  = 'Y'
AND    ROWNUM <= 60
```

질의 튜닝 예 (7)

- ✓ **SELECT 절의 서브쿼리는, 반환되는 tuple 개수만큼 수행됨,**
  예로 튜닝 전의 질의가, 60건을 반환하는 질의인데, 60건의 반환 Tuple안의 select count(*) 문을 2번씩 수행하여 총 120번을 수행하게 됨. 그러나 튜닝 후 불필요한 count(*) 질의를 줄여서 질의 수행시간을 단축시킬 수 있음.

튜닝 전	Num_data_page_fetches = 8542636 Num_data_page_dirties = 0 SQL statement execution time: 10.522477 sec
튜닝 후	Num_data_page_fetches = 53789 Num_data_page_dirties = 20614 SQL statement execution time: 0.108345 sec

질의 튜닝 예 (8)

✓ **컬럼 개수가 많은 테이블에 대해 특정 컬럼 값만 갱신이 자주 일어나는 경우, 테이블 전체에 대한 갱신이 발생되지 않도록 테이블 분리를 한다.**

✓ **트랜잭션 로그에 의한 성능 저하.**

DBMS 는 트랜잭션 처리를 위하여 undo/redo 로그를 사용.
Undo/redo 로그는 insert/update/delete 수행 시 쓰여짐.
예) update 가 수행될 경우, 아래와 같은 이미지가 저장됨.

Undo 로그 (update 이전이미지)	Redo 로그 (update 이후 이미지)

그러나 변형되는 컬럼의 값만 저장되는 것이 아니라 해당 Tuple 전체의 이미지가 저장됨.
따라서 테이블의 컬럼이 많을 경우 update 가 자주 발생하는 컬럼이 적다면 테이블 분리를 고려해볼 필요가 있음.

테이블 분리 전 VS 테이블 분리 후 아래 질의 실행
update tbl_2 set exec_cnt = exec_cnt + 1 where col_no = 100 의 질의를 1000건 반복 수행.

	테이블 분리 전	테이블 분리 후
Log 파일 개수	125000 페이지 log 파일 2개 사용	125000 페이지 log 파일 1개 사용
수행시간	약 0.00087~90 sec	약 0.000358 sec

Thank you.