

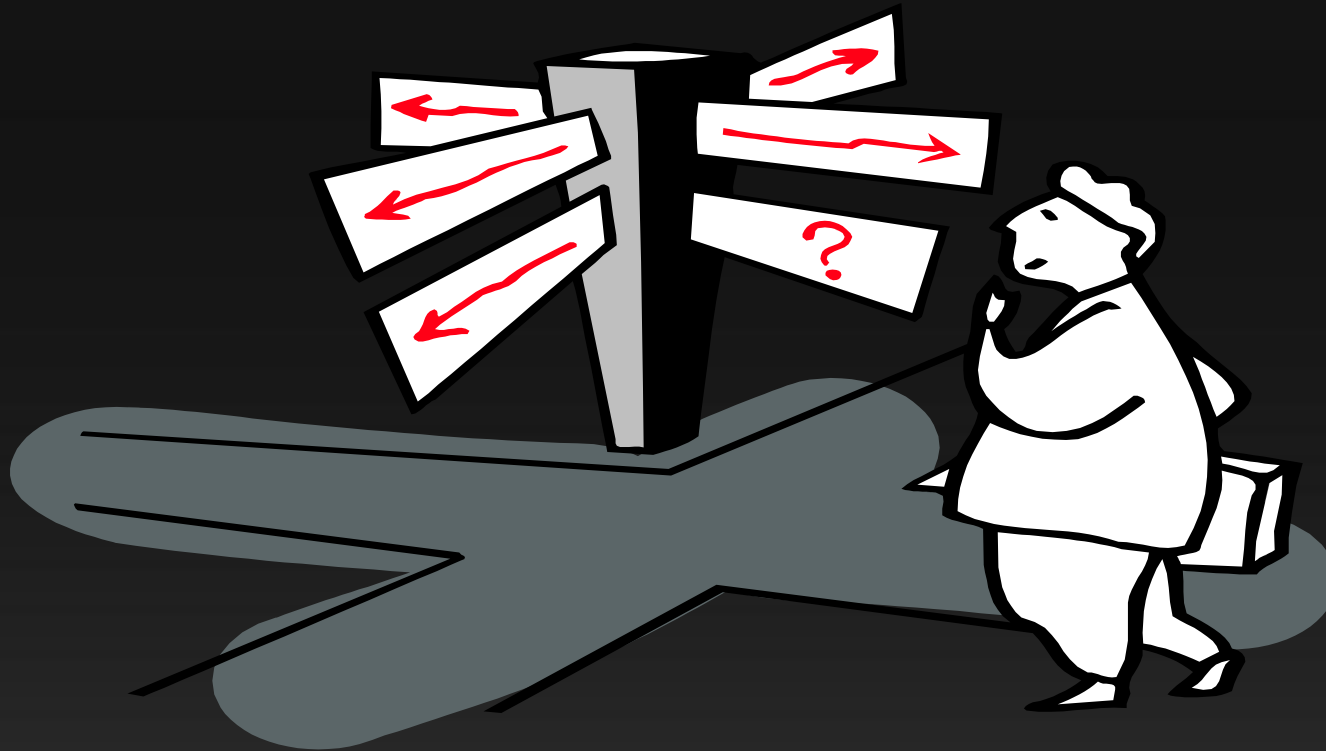
CUBRID를 활용한 SQL 교육

- 인덱스 집중 이해

작성자: CUBRID 운영파트

소속팀 /상위부서: 데이터플랫폼운영팀/NBP

1. 시작하기 전에



Query를 "잘" 작성하고 싶어서!

사전적 의미:

문의[질의]하다
의문을 제기하다.



Communication Basic

중요한 것은 "결과" 보다 "방법" 이다.

1. SQL은 그냥 결과만 나오면 되는 것 아닌가요? 개발하느라 바쁘는데...
2. DBMS 에 대해서 잘 몰라도 SQL을 작성할 수 있나요?
3. 그럼 DBMS별 특성을 전혀 몰라도 된다는 건가요?
4. CUBRID를 이용한 SQL ? CUBRID 하나도 모르는데요?

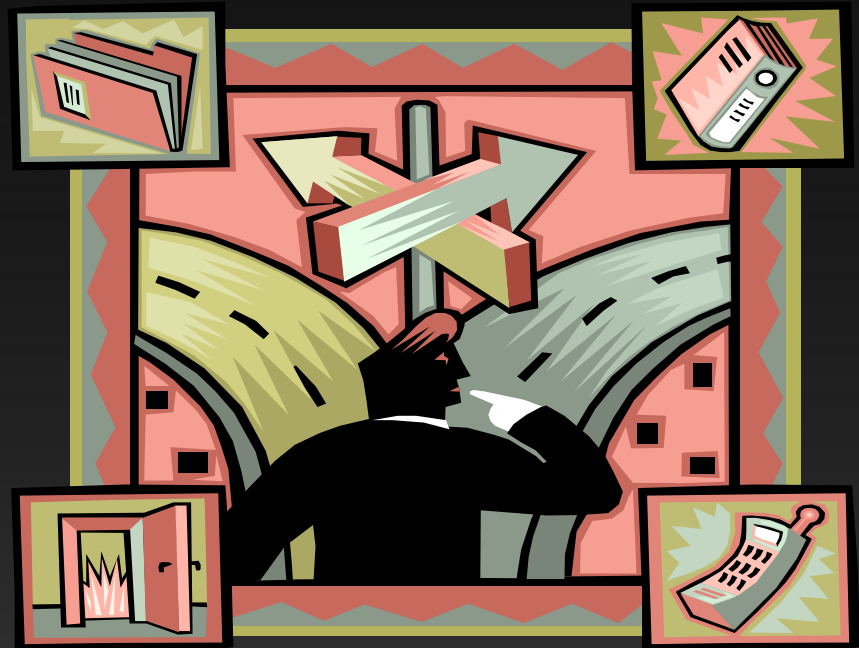
교육이 끝난 후에 난?

1. 인덱스의 구조를 이해할 수 있다.
2. 유사한 쿼리의 기능/성능 상의 차이점을 설명할 수 있다.
3. 실행계획을 읽고 일정 부분 개선할 수 있다.
4. 성능을 고려한 인덱스 설계를 어느 정도는 할 수 있다.
5. 성능을 고려한 쿼리 작성을 할 수 있다.
6. 실습이 하고 싶어진다.

2. 옵티마이저의 이해

✓ 옵티마이저(Optimizer)란?

- 사용자가 질의한 SQL을 가장 빠르고 효율적으로 수행 할 최적의 실행방법을 결정하는 DBMS의 내부 **핵심엔진**



✓ 옵티마이저의 종류

▪ 규칙기반(RBO) 옵티마이저

- 미리 정해놓은 규칙(우선순위)에 따라 실행계획을 생성
- 이용 가능한 인덱스 유무, SQL에서 사용하는 연산자의 종류, 참조하는 테이블에 따라 우선순위가 정해져 있음.
- 항상 우선순위가 높은 규칙으로 실행계획을 생성

▪ 비용기반(CBO) 옵티마이저

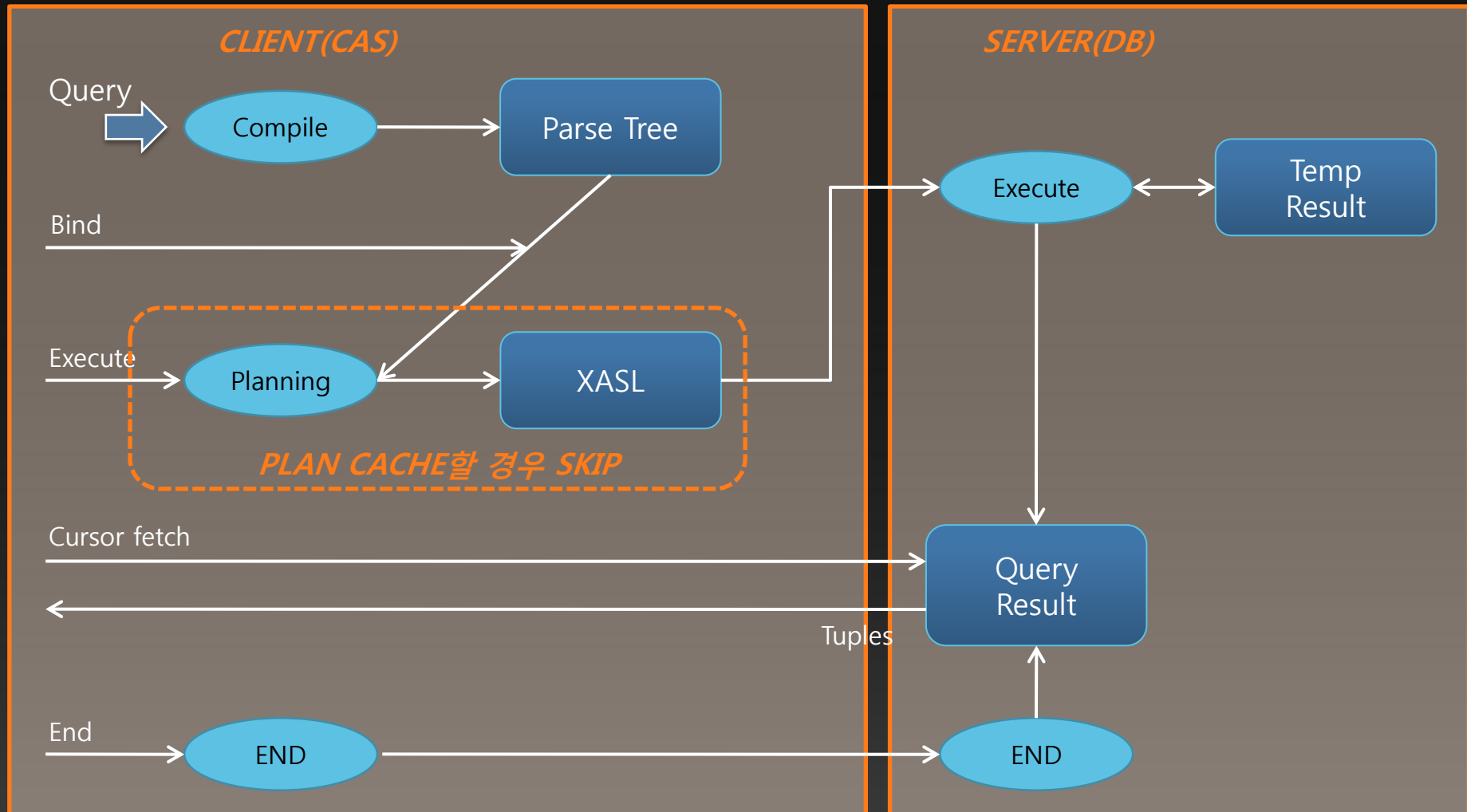
- SQL문을 처리하는데 필요한 비용이 가장 적은 실행계획을 선택
- 통계정보를 이용해 다양한 실행계획을 생성하고 비용을 예측
- 총 비용이 가장 낮은 실행계획 선택

CUBRID		MySQL (INNODB)
옵티마이저 타입	CBO	CBO
비용 산정 기준	<ul style="list-style-type: none"> CPU & DISK I/O 	<ul style="list-style-type: none"> DISK I/O (block random seek 수)
Block Size	<ul style="list-style-type: none"> 1K ~ 16K (NHN표준 : 16K) 	<ul style="list-style-type: none"> 16K 고정
데이터정렬	<ul style="list-style-type: none"> 데이터의 대한 정렬은 보장하지 않음 인덱스만 정렬되어 있음 인덱스는 레코드의 OID를 가지고 있음 	<ul style="list-style-type: none"> Clustered Index 저장 구조를 이용하여 데이터에 대한 순차적인 정렬을 보장함 인덱스는 Clustered Index의 Key값을 가지고 있음
통계정보	<ul style="list-style-type: none"> 테이블, 인덱스 컬럼에 대한 Cardinality/분포도 전체 페이지 샘플링 	<ul style="list-style-type: none"> 인덱스 컬럼에 대한 Cardinality/분포도 8개의 페이지 샘플링
NULL 처리	<ul style="list-style-type: none"> NULL 값은 인덱싱하지 않음 (통계정보에 NULL의 개수만 저장함) 	<ul style="list-style-type: none"> NULL을 값을 인식하여 인덱싱 지원
조인방식	<ul style="list-style-type: none"> Nested Loop JOIN Merge JOIN 	<ul style="list-style-type: none"> Nested Loop
인덱스 활용	<ul style="list-style-type: none"> FULL SCAN과 INDEX SCAN중 비용이 적은 것을 사용함. 	<ul style="list-style-type: none"> 인덱스를 적극 활용함
PLAN 생성	<ul style="list-style-type: none"> 통계정보 기반 (CBO) 	<ul style="list-style-type: none"> 통계정보 기반 (CBO)

2.3 질의 처리 과정

대외비

✓ CUBRID Query Processing



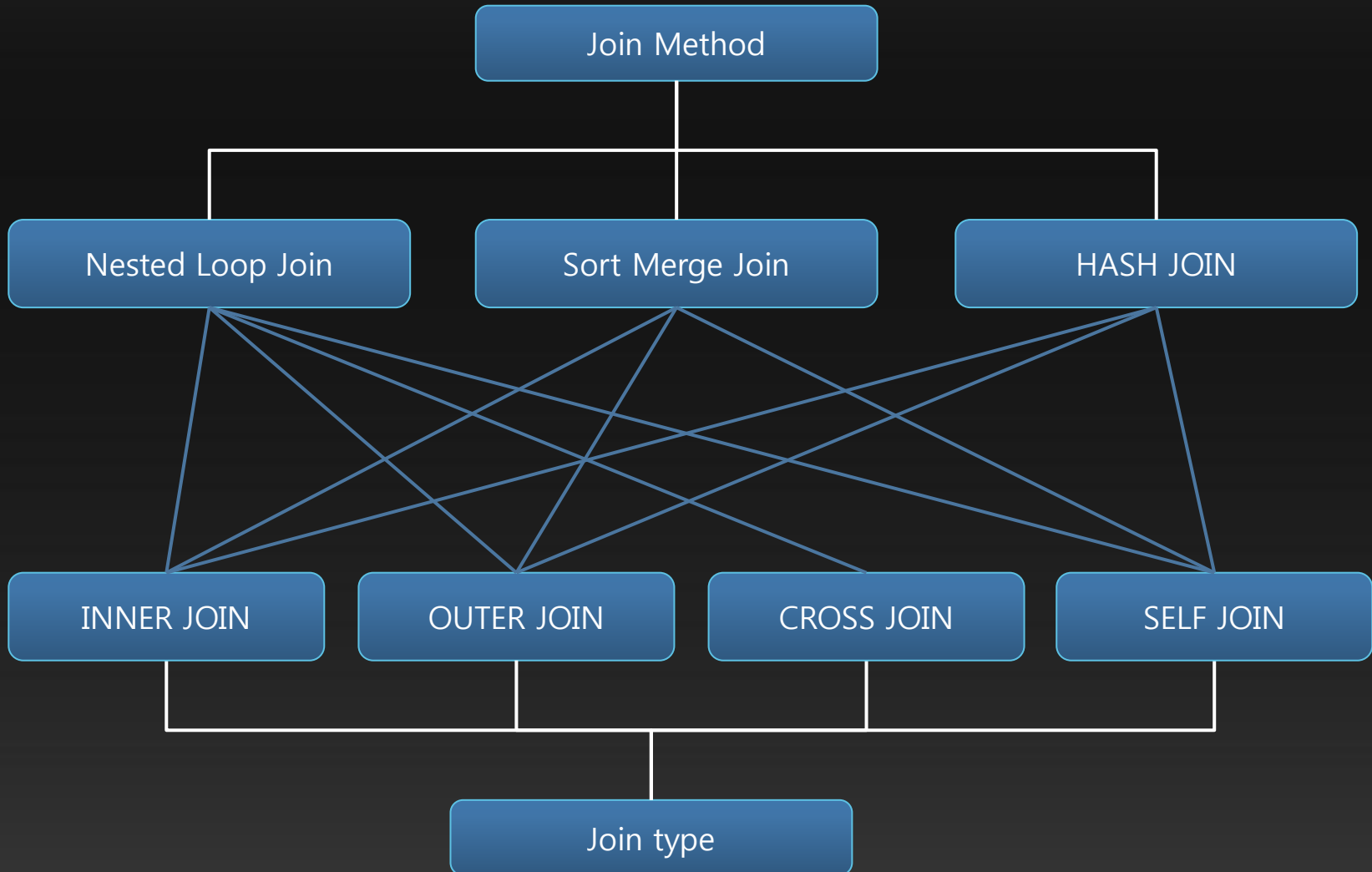
✓ 통계정보?

- Cost Base Optimizer 가 실행계획을 세울 당시에 참조하는 스키마 관련 정보
- 통계유형 : 테이블의 건수, heap page 수, 컬럼 개수
인덱스의 Total page수, Leaf 페이지 수, Tree의 depth
인덱스 컬럼의 distinct value , 데이터 분포도
숫자형 데이터 타입일 경우 min value / max value

✓ 통계정보 갱신 방법

- 쿼리 및 명령어를 통한 수집
 - cubrid optimizedb [OPTION] database-name
 - UPDATE STATISTICS ON { table_name | ALL CLASSES } ;
- 주의사항 : 통계정보 갱신 시 전체 테이블 데이터를 scan해서 통계정보를 갱신.
따라서 용량이 큰 테이블을 갱신하거나 DB를 여러 명이 사용하는 중이라면 조심!!

3. 조인의 이해

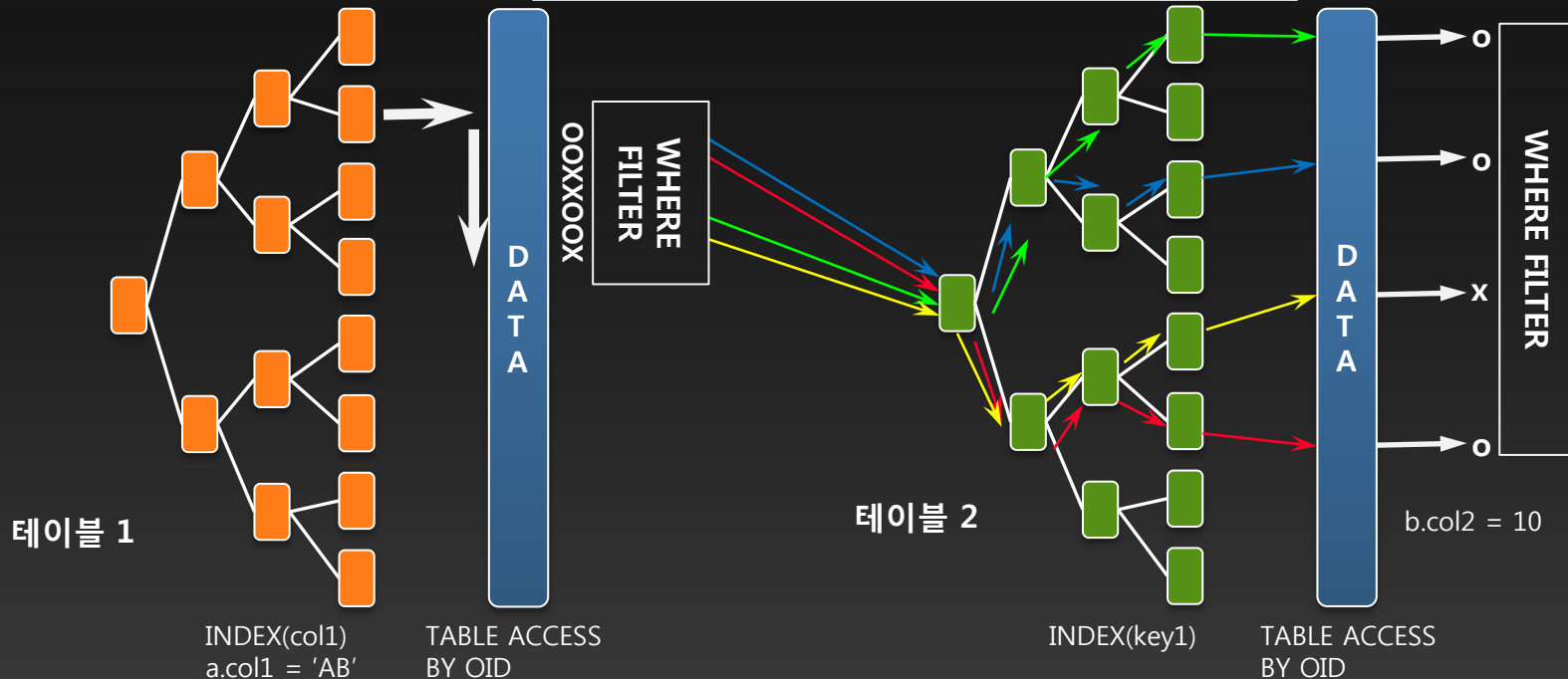


Nested Loop Join

✓ Nested Loop Join

- 순차적인 처리를 하며, 먼저 처리되는 테이블의 처리 범위에 따라 전체 쿼리의 비용이 결정됨
- 조인에 참여하는 레코드 건 수가 많아질수록 전체적인 응답 속도 저하가 발생
- 조인 연결고리 칼럼에 인덱스 구성이 되어 있지 않은 경우 Lookup 회수만큼 Full Table Scan을 해야 함
- 다량의 Random I/O가 발생
- 선행 테이블의 결과 건수가 작을 수록 성능에 유리, 후행 테이블의 JOIN KEY에는 인덱스가 있어야 성능에 유리.
- 부분 범위 처리 가능(Join 에 성공하는 데이터를 먼저 사용자에게 return)
- 주로 좁은 범위 처리에 유리

```
SELECT /*+ USE_NL */ a.col1, ... b.col1
FROM tab1 a, tab2 b
WHERE a.key1 = b.key2 AND a.col1 = 'AB' AND b.col2 = 10
```

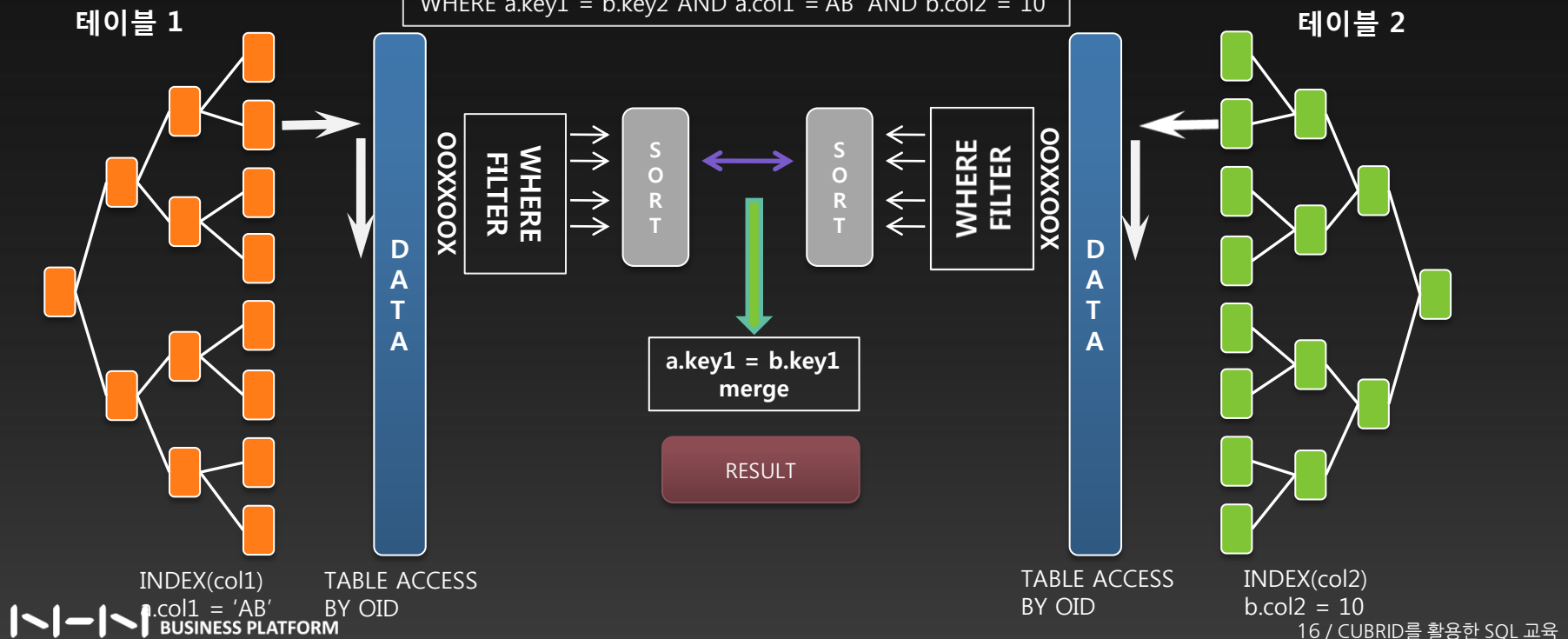


Sort Merge Join

✓ Sort Merge Join

- 양쪽 테이블을 정렬한 다음 조인 수행
- 연결고리 보다 각 테이블이 가지고 있는 검색 조건에 따른 Data량에 따라 성능 차이가 큼
- 통계성 작업이나 배치 작업에 주로 사용
- 대량 정렬 작업 시 Temp 를 사용하기 때문에 OLTP 성 업무에 적합하지 않음
- 양쪽 테이블에 JOIN KEY 로 인덱스가 걸려있을 경우 별도의 정렬작업 없이 바로 조인 가능
- 부분 범위 처리 불가능 (결과 set을 모두 만든 후 사용자에게 값을 return)
- 메모리 크기에 비해 sort할 양이 많아질 경우 Disk IO를 발생시키므로 정렬 효율 저하

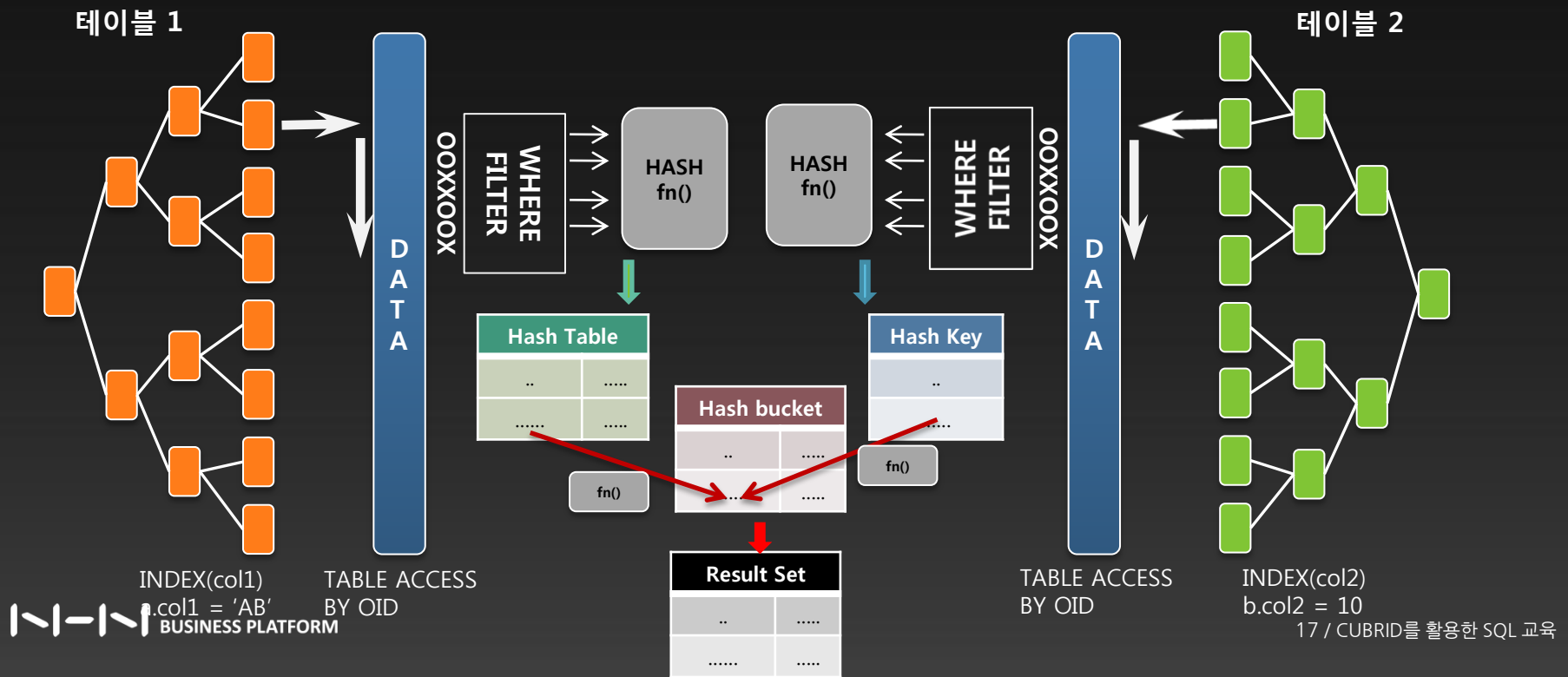
```
SELECT /*+ use_merge */ a.col1, ... b.col1
FROM tab1 a, tab2 b
WHERE a.key1 = b.key2 AND a.col1 = 'AB' AND b.col2 = 10
```



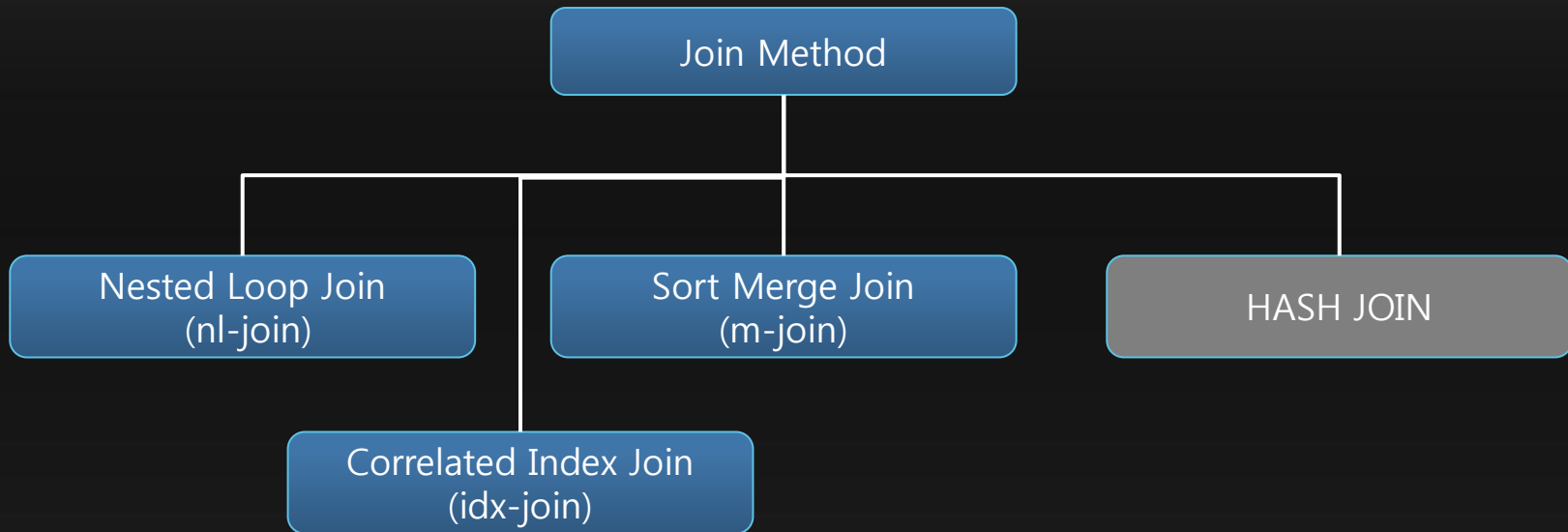
✓ Hash Join

- Nested Loop의 단점인 random IO와 Merge Join의 단점인 Disk IO 문제를 해소하기 위해 생긴 Join방식.
- 연결 고리로 Hash Key를 사용하기 때문에 Equi-Join에만 사용 가능
- CPU power에 의존적
- 대량의 데이터 JOIN이 필요하고 마땅히 사용할 인덱스가 없을 경우 선택

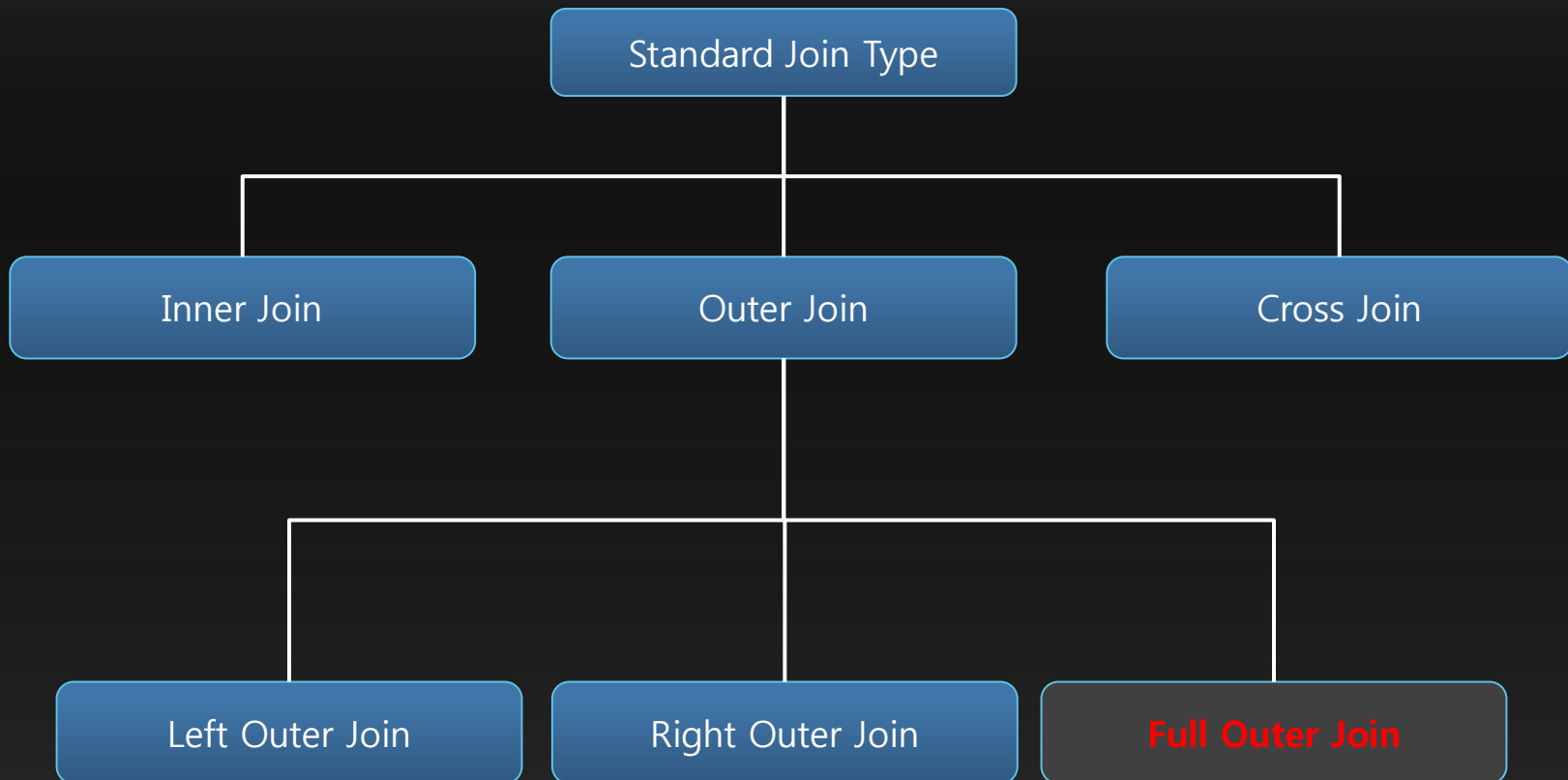
```
SELECT a.col1, ... b.col1
FROM tab1 a, tab2 b
WHERE a.key1 = b.key2 AND a.col1 = 'AB' AND b.col2 = 10
```



	Nested Loop Join	Sort Merge Join	Hash Join
MySQL	O	X	X
CUBRID	O	O	X
ORACLE	O	O	O
MSSQL	O	O	O



- 후행 테이블의 join key에 인덱스가 존재하는 경우

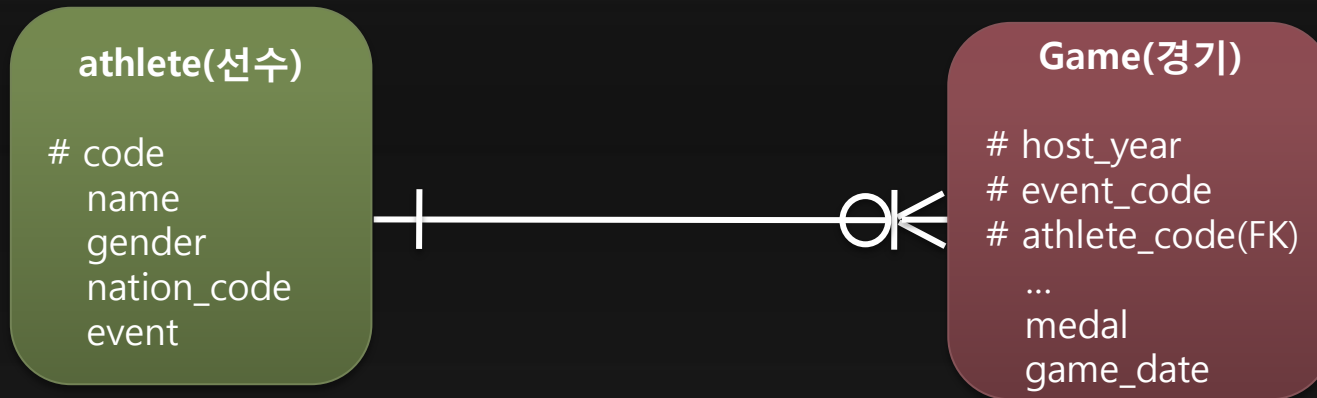


4. 다양한 데이터 연결 방식

4.1 데이터를 추출하는 여러 가지 방법

대외비





A 집합 : 국적이 한국인 양궁 선수

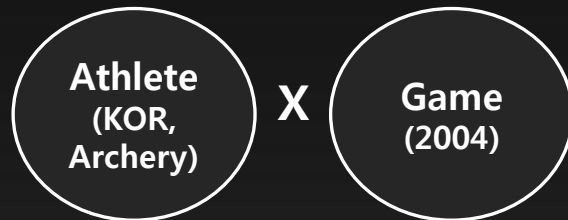
B 집합 : 2004년에 개최된 게임

```
SELECT a....  
FROM athlete a  
WHERE a.nation_code = 'KOR' AND a.event = 'Archery'
```

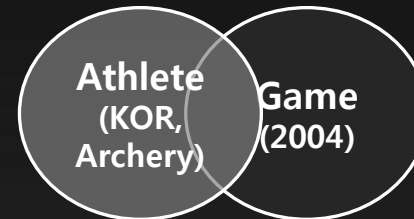
```
SELECT g....  
FROM game g  
WHERE g.host_year = 2004
```

```
SELECT a.code, a.name, g.athlete_code, g.game_date, g.medal
FROM athlete a
[CROSS | INNER | LEFT OUTER ] JOIN game g ON a.code = g.athlete_code AND g.host_year = 2004
WHERE a.nation_code = 'KOR' AND a.event = 'Archery' AND g.host_year = 2004
```

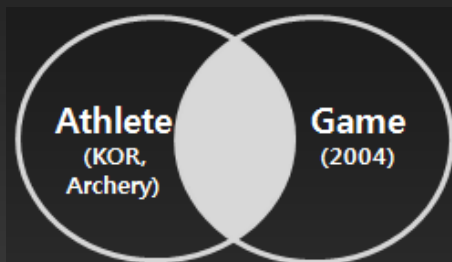
✓ Cross Join



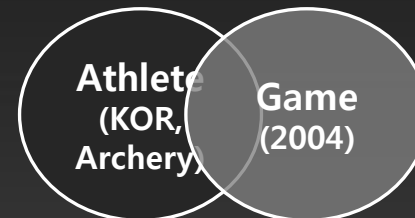
✓ Left Outer Join



✓ Inner Join



✓ Right Outer Join



✓ Join 의 특징

- 1:M 관계의 테이블을 JOIN할 경우 1인 테이블의 행이 M인 테이블의 행의 갯수만큼 **복제**된 것처럼 보임.
- A JOIN B 의 결과는 컬럼 목록(가로)으로 나열할 수 있음
- Left / Right outer JOIN일 경우 조건에 매치하지 않는 컬럼은 NULL로 표현됨

a.code	a.name	nation	event
1	Lee Sung Jin	KOR	Archery
2	Park Sung Hyun	KOR	Archery
3	Kim Jo-Sun	KOR	Archery

g.athlete_code	g.game_date
1	2004-03-28
1	2004-04-20
2	2004-08-28
4	2004-08-22

4.2 Standard Join

✓ Inner Join

a.code	a.name	nation	event		g.athlete_code	g.game_date		a.code	a.name	g.game_date
1	Lee Sung Jin	KOR	Archery	→	1	2004-03-28	=	1	Lee Sung Jin	2004-03-28
2	Park Sung Hyun	KOR	Archery	→	1	2004-04-20		1	Lee Sung Jin	2004-04-20
3	Kim Jo-Sun	KOR	Archery	→	2	2004-08-28		2	Park Sung Hyun	2004-08-28
				✗	4	2004-08-22				

✓ (Left) Outer Join

a.code	a.name	nation	event		g.athlete_code	g.game_date		a.code	a.name	g.game_date
1	Lee Sung Jin	KOR	Archery	→	1	2004-03-28	=	1	Lee Sung Jin	2004-03-28
2	Park Sung Hyun	KOR	Archery	→	1	2004-04-20		1	Lee Sung Jin	2004-04-20
3	Kim Jo-Sun	KOR	Archery	→	2	2004-08-28		2	Park Sung Hyun	2004-08-28
				✗	4	2004-08-22		3	Kim Jo-Sun	NULL

- ✓ Standard JOIN 사용 시 이것만은 지켜주세요.!!

반드시 ANSI 표준으로 쿼리를 작성해주세요.

```
SELECT a.code, a.name, g.athlete_code, g.game_date, g.medal
FROM athlete a , game g
WHERE a.nation_code = 'KOR' AND a.event = 'Archery' AND g.host_year = 2004
```

WHY?

REAL DB에서 발생하는 인재 중
JOIN 키 실종으로 인한 장애 빈도가 높음

- ✓ Left / Right OUTER JOIN 시 주의사항

기준이 되는 테이블에 filter로 걸리는 조건은 반드시 **ON** 절에 명시해야 함. (oracle style join의 경우 (+))

ex)

```
SELECT a.code, a.name, g.athlete_code, g.game_date, g.medal
FROM athlete a
```

```
LEFT OUTER JOIN game g ON a.code = g.code AND g.host_year = 2004
WHERE a.nation_code = 'KOR' AND a.event = 'Archery'
```

WHY?

WHERE절에 조건이 올 경우 INNER JOIN
으로 풀려 원하는 결과가 나오지 않을 수 있음

member(사용자)

```
# mbr_id  
naverid  
nickname  
gender  
.....
```

A 집합 : naverid = 'nhn'

```
SELECT a....  
FROM member a  
WHERE a.naverid = 'nhn'
```

B 집합 : nickname = 'nhn'

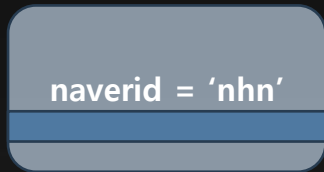
```
SELECT a....  
FROM member a  
WHERE a.nickname = 'nhn'
```

SELECT ... FROM member as a WHERE a.naver_id = 'nhn'

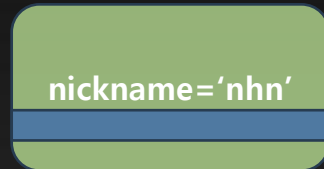
[UNION | UNION ALL | DIFFERENCE | INTERSECT]

SELECT ... FROM member as a WHERE a.mbr_nickname = 'nhn'

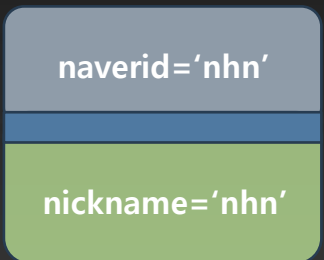
✓ UNION



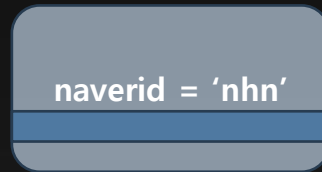
+



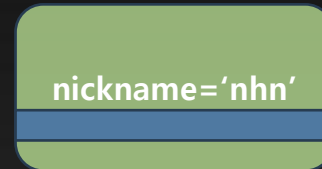
=



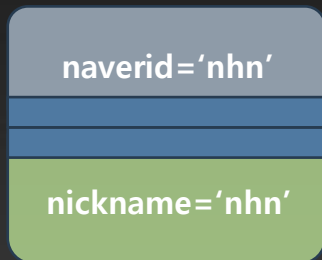
✓ UNION ALL



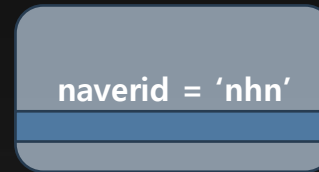
+



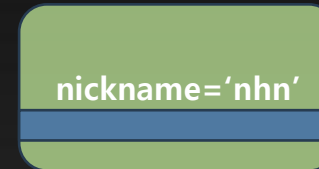
=



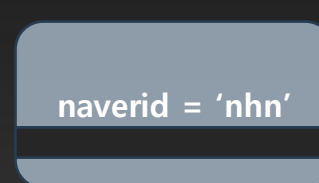
✓ DIFFERENCE



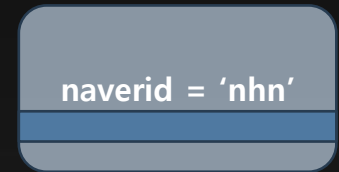
-



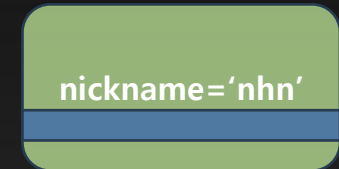
=



✓ INTERSECT



^



=



✓ SET 의 특징

- JOIN과 같은 데이터 복제는 발생하지 않음.

다만, UNION ALL 인 경우 중복 데이터를 제거하지 않으므로 동일한 데이터가 그대로 출력됨

- A 와 B 를 JOIN한 결과는 COLUMN 이 가로로 펼쳐지지만 SET OPERATION 한 결과는 세로로 나열됨.

- 따라서 A와 B의 결과를 SET OPERATION 하려면 컬럼 목록이 동일해야 함.

여기서 잠깐!!!



Q : UNION 과 UNION ALL 의 차이를 아시나요 ?

A : 중복 데이터만 제거해주는 거 아닌가요..??

맞습니다만, 또 다른 차이점은 “동작 방식”입니다.

UNION 의 경우, 모든 데이터를 정렬한 후 중복된 항목을 제거하므로

Sorting 연산으로 인해 성능을 저하시킬 수 있어요.

따라서, “기능”과 “동작 방식”을 함께 고려하여 “잘” 사용하여야 합니다.

위치에 따른 분류

Subquery 위치	비고
SELECT	<p>스칼라 서브쿼리라고도 부름</p> <p>다른 테이블의 값을 조회할 때 사용됨</p> <p>결과 행만큼 Subquery 내의 테이블에 접근하며 일치하는 행이 없을 경우 NULL 리턴</p> <p><u>메인 쿼리의 결과 집합의 개수에 영향을 주지 않음</u></p> <p>SUB QUERY 내의 결과 건수는 반드시 1건이어야 함</p>
FROM	<p>가상의 결과 집합을 만들어내기 때문에 INLINE VIEW라고도 부름</p> <p>INNER JOIN 이거나 LEFT OUTER JOIN이어도 1:N 관계일 경우 <u>메인 쿼리의 결과 집합의 개수에 영향을 줌</u></p> <p>SUBQUERY 내의 컬럼을 SELECT 절에 사용할 수 있음</p>
WHERE	<p>메인 쿼리의 데이터를 filtering 하는데 사용됨</p> <p>메인 쿼리의 결과 집합의 개수를 늘리지는 않고 <u>동일하게 유지하거나 감소시킴</u></p> <p>SUBQUERY 내의 컬럼을 SELECT절에 사용할 수 없음</p>
HAVING	<p>GROUP BY 의 HAVING 절의 조건 작성 시 다른 테이블과 연관된 조건을 작성할 때 사용</p>
INSERT문의 INTO절	<p>다른 테이블의 값을 추출해 INSERT 할 때 사용됨</p>
UPDATE문의 SET절	<p>다른 테이블의 값을 추출해 UPDATE 할 때 사용됨</p>

결과 건수에 따른 분류

서브쿼리 결과 건수	비고
단일행 서브쿼리	서브쿼리의 리턴 건수가 1건이며 WHERE 절에 위치하는 서브쿼리일 경우 =,<,> 등의 연산자를 사용할 수 있다. SELECT절에 오는 서브쿼리는 반드시 단일행 서브쿼리여야 한다.
복수행 서브쿼리	서브쿼리의 결과가 2건 이상인 경우 WHERE절에서 사용할 경우 반드시 복수행 연산자 (IN, ANY, SOME, EXISTS 등)를 사용해야 한다.
다중 컬럼 서브쿼리	서브쿼리의 실행 결과로 여러 컬럼을 반환한다. 메인 쿼리의 조건 절에서 여러 컬럼을 동시에 비교할 수 있으며 컬럼 순서는 동일해야한다.

메인쿼리 연관성에 따른 분류

연관성	비고
상관 서브쿼리	서브쿼리의 값이 메인쿼리의 값에 의존 서브쿼리 단독으로 실행되지 않음 (오류 발생!!) 메인 쿼리의 결과 건수만큼 서브쿼리의 테이블에 접근
비상관 서브쿼리	외부 쿼리를 참조하지 않고 단독으로 사용되는 서브 쿼리 서브쿼리만 단독으로 실행해도 오류가 발생하지 않음

결과의 차이

```
SELECT a.code, a.name, g.athlete_code, g.game_date, g.medal
FROM athlete a
WHERE a.nation_code = 'KOR' AND a.event = 'Archery'
AND a.code [NOT][ IN | SOME | ANY | ALL ]
      ( SELECT athlete_code from game g where g.host_year = 2004 )
AND Exists (SELECT * FROM game g where a.code = g.athlete_code and g.athlete_code = a.code)
```

✓ IN / Exists / SOME / ANY

Main query의 결과에서 Sub query 연결 조건에 만족하는 값이 sub query의 결과에 **하나 이상** 만족하는 경우 true를 반환

✓ ALL

Main query의 결과에서 Sub query 연결 조건에 만족하는 값이 sub query의 **결과에 모두** 만족하는 경우 true

✓ NOT IN / NOT Exists

Main query의 결과에서 Sub query 연결 조건에 만족하는 값이 sub query의 결과 **전체에 만족하지 않는** 경우 true를 반환

동작 방식의 차이

✓ IN / SOME / ANY / ALL

Sub query의 결과를 distinct 해서 temp table 로 메모리에 만들어놓고 Main query 와 JOIN 수행.
sub query 내의 결과를 distinct할 때 sorting 발생

✓ Exists

Main query를 수행하면서 추출된 결과를 하나씩 sub query의 조건식과 비교하여 만족할 경우 행을 반환
만족하지 않을 경우 결과 행 버림.

Main query의 결과 건수만큼 Subquery로의 random access 발생

Q : 그런 내부적인 동작 방식까지는 몰랐어요.. 어떻게 알 수 있죠?

A : 실행계획에 다~ 나와요.

Q : 그런데, 결과만 같으면 되는 것 아니에요? 동작 방식이 달라서 뭐가 어떻게 달라지는데요?

A : UNION 과 UNION ALL 의 예를 생각해보세요.

함수/연산자 사용에서 “기능” 보다 중요하게 생각해야 하는 것이 “동작방식” 입니다.

IN : Sub query의 결과를 distinct 하므로 정렬 비용이 들어가고 메모리 공간을 사용.

Exists : Main query에서 Sub query의 테이블로 random access 비용 발생.

사용 방법의 차이

✓ INLINE-VIEW **VS** SCALAR SUB QUERY

```
SELECT a.code, a.name, g.lastdate
FROM athlete a
INNER JOIN (SELECT distinct athlete_code,max(game_date) as lastdate
            FROM game
            WHERE host_year = 2004
            group by athlete_code) as g ON a.athlete_code = a.code
WHERE a.nation_code = 'KOR' AND a.event = 'Archery'
```

```
SELECT a.code, a.name, g.lastdate ,(SELECT max(game_date) from game g1 where host_year = 2004 and g1.athlete_code
= a.code) as lastdate
FROM athlete a
WHERE a.nation_code = 'KOR' AND a.event = 'Archery'
```

동작 방식의 차이

✓ INLINE-VIEW

SUB QUERY의 결과를 메모리 공간에 임시테이블로 만들어놓고 MAIN TABLE과 JOIN 수행, 정렬은 하지 않음

✓ SCALAR SUBQUERY

MAIN QUERY 를 수행하면서 추출된 결과를 하나씩 sub query의 조건식과 비교하여 만족할 경우 결과값을 반환
만족하지 않을 경우 NULL 을 반환.

Q : 그럼 이건 언제 사용해야 하나요?

A : IN / Exists 의 경우와 유사하게 생각하시면 됩니다.

고려해야 하는 것은 데이터 분포, 인덱스 컬럼, filter 조건이 없는지!!

테이블의 관계, sub query에 따라 적절한 sub query 를 사용하는 기법도 중요한 튜닝 기법의 하나

INLINE-VIEW는 Main query의 데이터 건수가 늘어날 수 있음.

(데이터 중복을 막기 위해 Distinct를 사용해야 할 수도 있음)

SCALAR SUBQUERY의 경우 Main query와의 관계가 1:0~1 인 경우에만 사용할 수 있음.

부록 : CUBRID 의 SET 연산자

✓ 종류 및 사용 방법

SELECT ,**LIST** (SELECT ...) FROM .. WHERE ...

SELECT ,**SET** (SELECT ...) FROM .. WHERE ...

SELECT ,**MULTISET** (SELECT ...) FROM .. WHERE ...

SELECT ,**SEQUENCE** (SELECT ...) FROM .. WHERE ...

SELECT a.code, a.name, a.gender,LIST(SELECT game_date from game g1 where host_year = 2004 and g1.athlete_code = a.code)

FROM athlete a

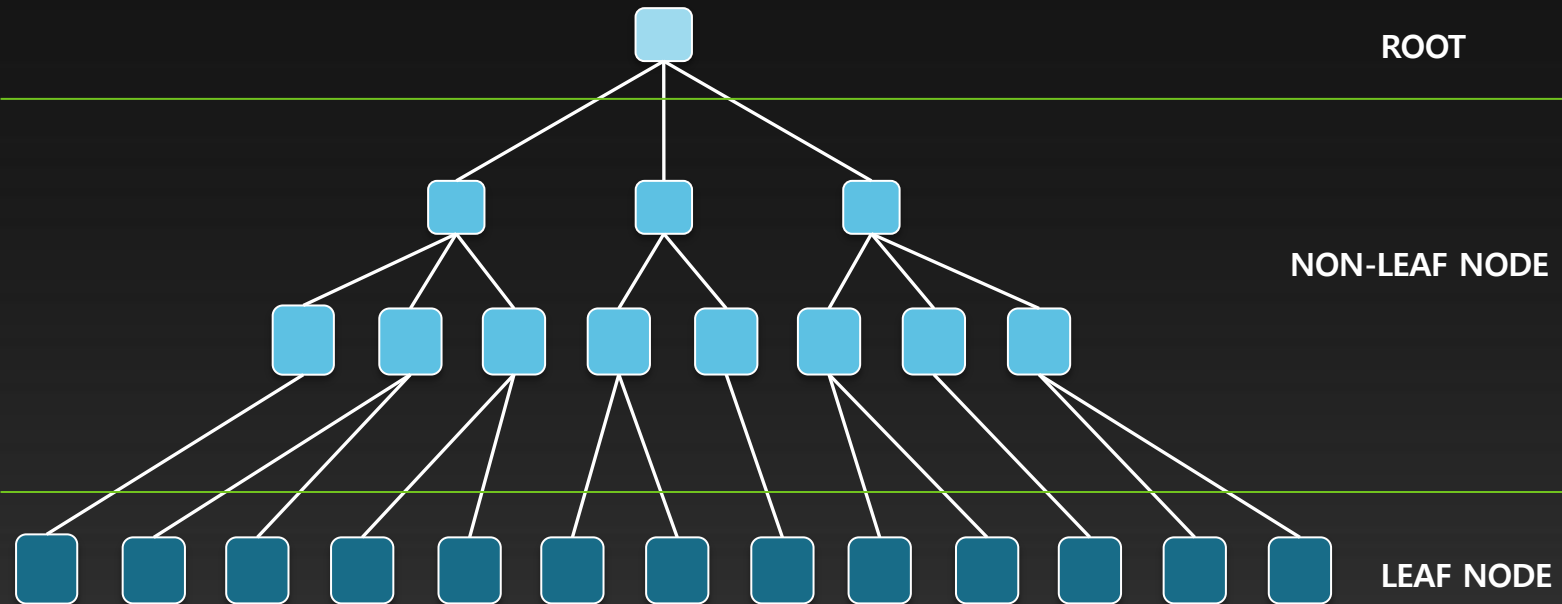
WHERE a.nation_code= 'KOR' and a.event='Archery'

타입	설명	실제 데이터	출력되는 데이터
SET	중복을 허용하지 않는 합집합	{'c','c','c','b','b', 'a'} {3,3,3,2,2,1,0,'c','c','c','b','b', 'a'}	{'a','b','c'} {0,1,2,3,'a','b','c'}
MULTISET	중복을 허용하는 합집합	{'c','c','c','b','b', 'a'} {3,3,3,2,2,1,0,'c','c','c','b','b', 'a'}	{'a','b','b','c','c','c'} {0,1,2,2,3,3,3,'a','b','b', 'c','c','c'}
LIST SEQUENCE	중복을 허용하고, 데이터 입력 순서대로 저장하는 합집합	{'c','c','c','b','b', 'a'} {3,3,3,2,2,1,0,'c','c','c','b','b', 'a'}	{'c','c','c','b','b','a'} {3,3,3,2,2,1,0,'c','c','c','b','b','a'}

5. 인덱스의 이해

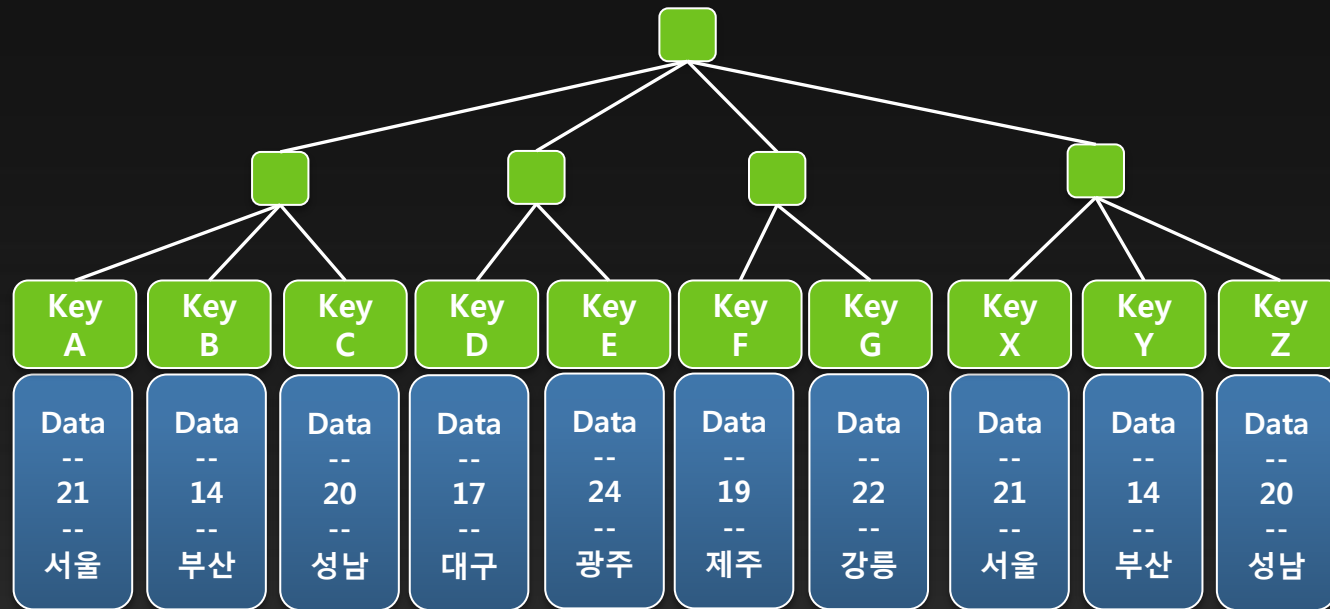
1) ROOT, NON-LEAF NODE, LEAF NODE

- ROOT : 인덱스 TREE 의 최 상단 NODE
- NON-LEAF NODE : 인덱스 TREE의 중간 NODE
- LEAF-NODE : 인덱스 TREE의 최하위 NODE



2) Clustered Index

- INDEX KEY 값으로 데이터가 정렬되어 저장되어있음.
- LEAF NODE = 데이터페이지
- 책자의 맨 앞 “목차” 에 해당하는 인덱스. (목차 순서 = 실제 데이터 순서)
- 테이블당 0 또는 1개만 존재

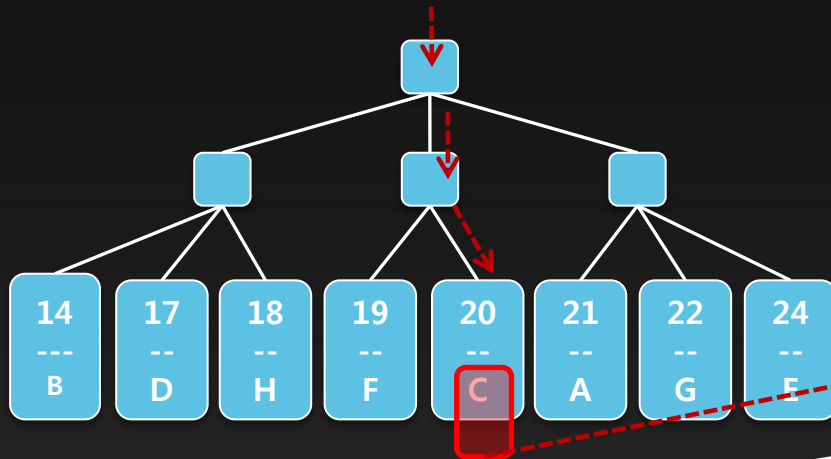


INDEX Key값의 순서 = 데이터 페이지의 순서

3) Non-clustered Index

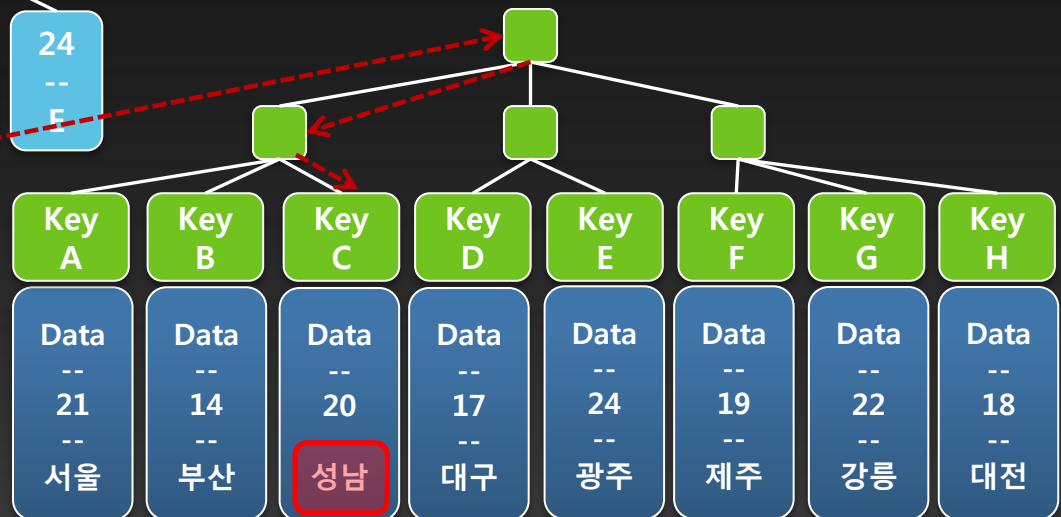
- LEAF NODE = 데이터 페이지의 주소값 혹은 Clustered Index의 Key값
- 책자 맨 뒤의 “색인”에 해당.
- 색인은 여러 개를 만들 수 있음. ex) 알파벳 순, 생성일 순, 작성자 순 등

<Non-Clustered Index>



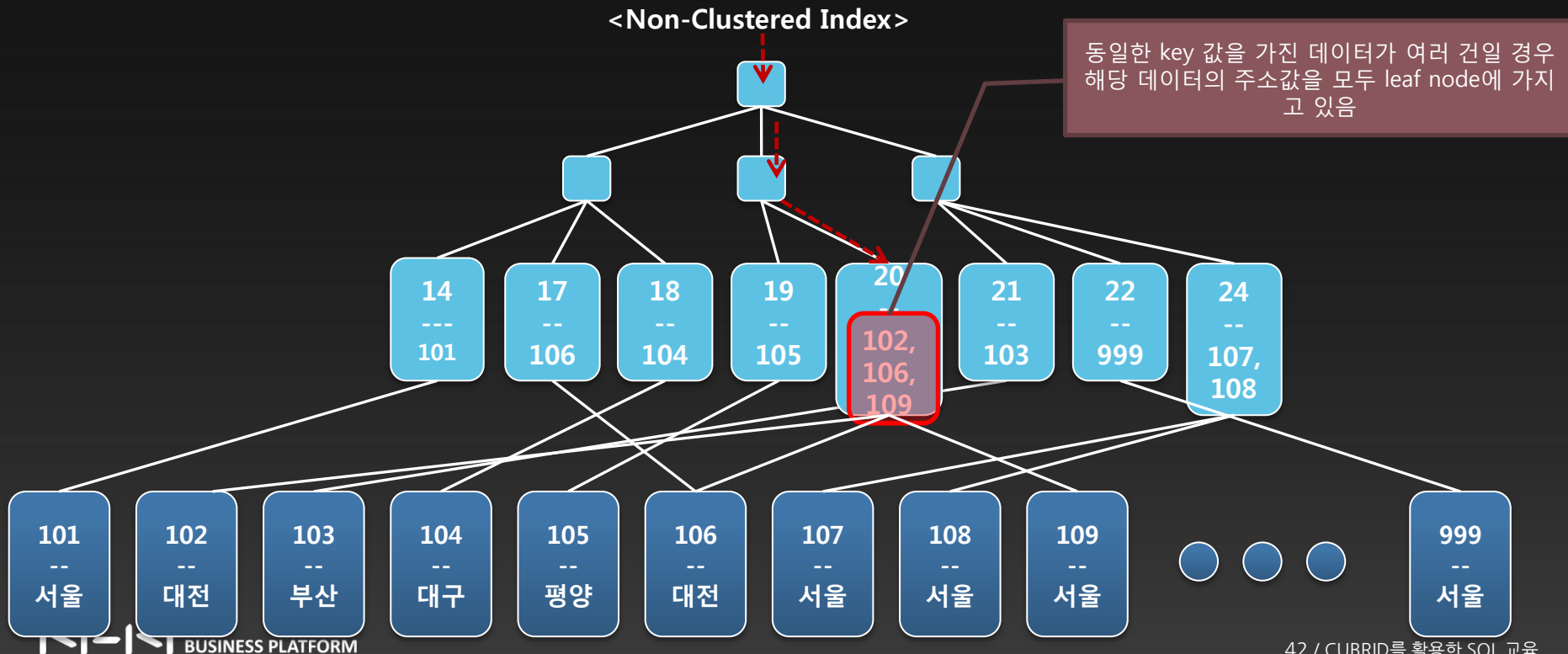
데이터를 읽기 위해 클러스터 인덱스로 점프.
클러스터 인덱스에서 다시 해당 데이터를 찾음.

<Clustered Index>



4) Heap

- Clustered Index 가 없는 테이블을 Heap 이라고 명명
- Heap 구조의 테이블에서 Non-clustered Index 의 leaf node 는 해당 Index Key 값이 속한 데이터 페이지의 주소값을 가지고 있음
- Clustered Index 가 있는 테이블일 경우 데이터 순서는 Clustered Index의 Key 정렬 순서와 동일하지만 Heap 일 경우에는 데이터 인덱스 key 값과 무관하게 heap file 에 데이터가 들어감.
- 데이터 추출 범위가 넓을 수록 다량의 random access가 발생
- **CUBRID 는 Clustered Index 를 제공하지 않으므로 모든 테이블은 Heap 구조임!**



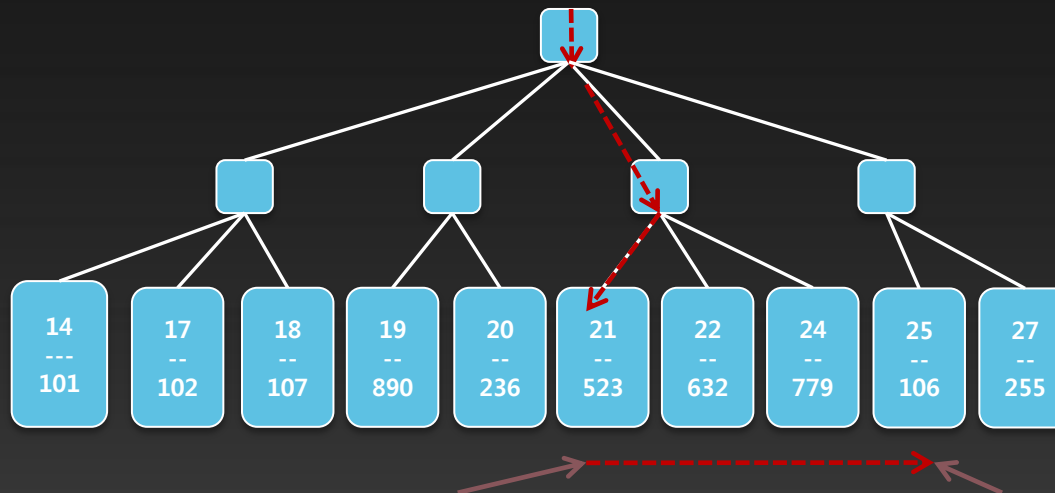
5) prefix B+ tree

- B+ tree : B Tree의 한 종류, Leaf Node에 Key와 key에 대응하는 데이터의 포인터를 저장
- Leaf node 는 Double Linked List로 연결되어 있어 범위 검색과 같은 순차 처리가 가능
- Prefix B+ Tree :
인덱스 컬럼이 string type일 경우 인덱스의 중간 node 에는 string을 모두 저장하지 않고 검색을 위한 분기를 할 수 있을 정도의 최소한의 string 만을 저장함

6) Index range scan

- 인덱스가 Key column 순으로 정렬되어 있기 때문에 특정 위치에서 검색을 시작해서 검색 조건이 일치하지 않는 값을 만나는 순간 스캔을 멈춤
- 동작 방식 :

SELECT * FROM TABLE WHERE age between 21 and 24



Range scan 시작

일치하지 않는 값을 만나면 중지

7) OID

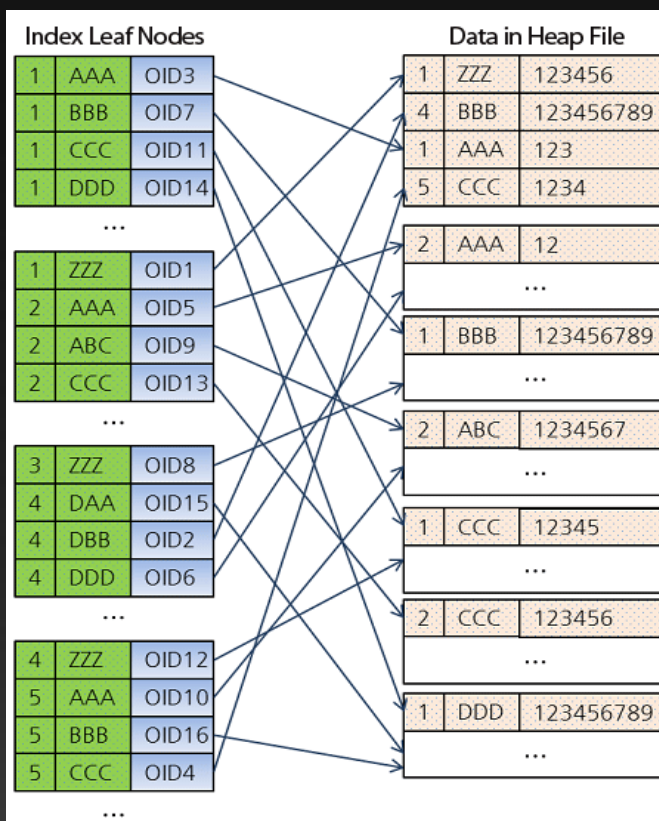
- CUBRID 에서 데이터를 저장할 때 사용하는 내부적인 주소 값

질의 처리 과정 이해

1) CUBRID의 인덱스와 테이블 데이터의 관계

```
CREATE TABLE tbl (a INT NOT NULL, b STRING, c BIGINT);
CREATE INDEX idx ON tbl (a, b);
```

왼쪽 인덱스 LEAF NODE 에는 인덱스 키와 키에 대응되는 OID(레코드의 물리적 주소 값)가 저장되어 있음



질의 처리 과정 이해

2) CUBRID 범위 스캔 방식

쿼리문 :

```
SELECT * FROM tbl WHERE a > 1 AND a < 5 AND b < 'K' AND c > 10000 ORDER BY b;
```

WHERE 절 구문 분석 :

1. **Key range** : 인덱스 스캔 범위로 활용되는 조건 ($a > 1$ and $a < 5$)
2. **Key filter** : Key range에 포함될 수 없지만 인덱스 키로 처리 가능한 조건 ($b < 'K'$)
3. **Data Filter** : 인덱스를 사용할 수 없는 조건.
인덱스 페이지에서 실제 데이터 페이지를 읽으려 random access가 발생한 후 적용된다. ($c > 10000$)

CUBRID의 질의 처리 과정 :

1. 인덱스를 스캔하여 먼저 Key Range와 Key Filter를 적용하여 조건에 부합하는 OID 리스트를 만들어냄.
이 과정은 Key Range의 시작부터 끝까지 계속 된다.
2. OID 를 이용하여 데이터 페이지에서 해당 레코드를 읽어 Data Filter 를 적용
3. SELECT List에 기술된 컬럼 값을 읽어와 결과를 저장하는 임시 페이지에 기록
4. ORDER BY나 GROUP BY가 없을 경우 임시 페이지의 주소 값을 client 에 반환
5. ORDER BY나 GROUP BY 가 있을 경우 임시 페이지의 레코드를 정렬하여 최종 결과를 생성 후 해당 페이지의 주소값을 client 에 반환

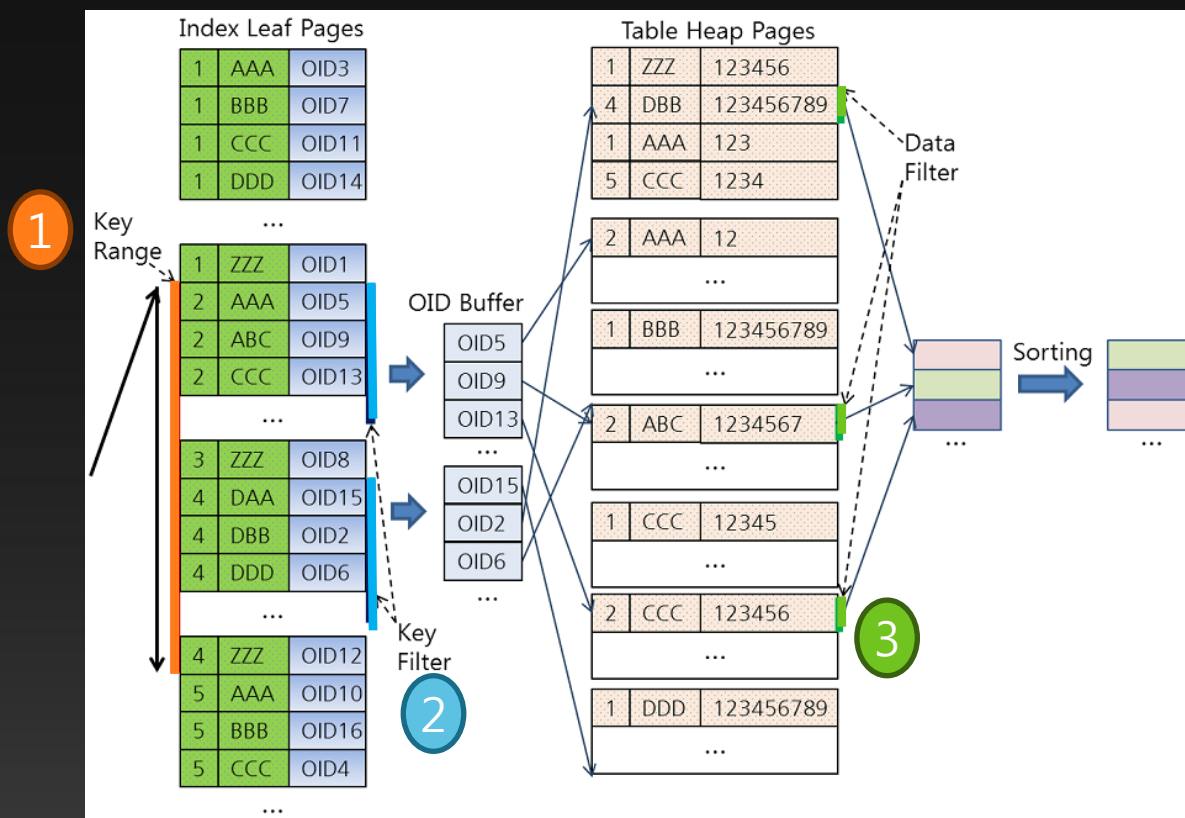
:

질의 처리 과정 이해

2) CUBRID 범위 스캔 방식

질의 처리 과정 :

SELECT * FROM tbl WHERE a > 1 AND a < 5 AND b < 'K' AND c > 10000 ORDER BY b;



KEY FILTER

1. KEY FILTER란 ?

Key range에 포함될 수 없지만 인덱스 키로 처리 가능한 조건
복합 컬럼으로 구성된 인덱스에 대해 선두 조건이 equal 이 아닌 범위 조건이거나 (IN 조건 포함)
인덱스 컬럼 중 중간에 있는 컬럼에 대한 조건절이 없는 경우 해당 컬럼 뒤에 위치한 인덱스 컬럼의 조건은
KEY FILTER 로 작용함
KEY FILTER는 KEY RANGE에 만족하는 모든 인덱스 페이지를 읽으면서 filtering을 하기 때문에 KEY RANGE에서
걸러지는 결과 건수가 많을 경우 KEY FILTER 에도 비용이 많이 소요됨.

2. 실행계획

ex) SELECT * FROM tbl WHERE a > 1 AND a < 5 AND b < 'K'

iscan

class: tbl node[0]

index: idx term[1]

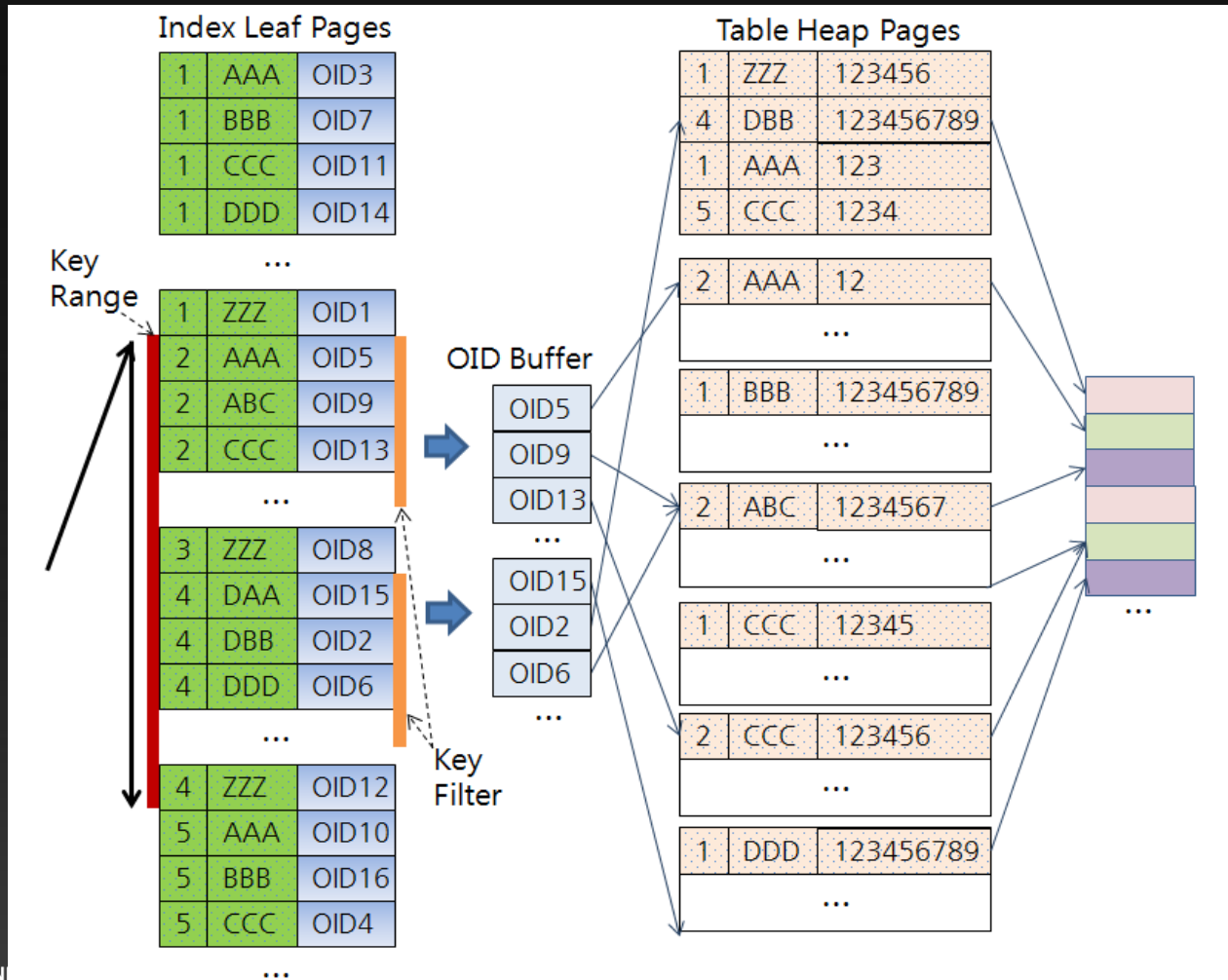
filtr: term[0]

cost: fixed 0(0.0/0.0) var 1(0.0/1.0) card 1

KEY FILTER

3. 동작방식

SELECT * FROM tbl WHERE a > 1 AND a < 5 AND b < 'K'



여기서 잠깐!!!



Q : KEY RANGE, KEY FILTER, DATA FILTER .. 어떤 게 제일 좋은 거예요?

A : KEY RANGE > KEY FILTER > DATA FILTER 순으로 성능이 좋음

따라서 튜닝의 기본적인 철학은 KEY FILTER, DATA FILTER 를 되도록 없애고
KEY RANGE의 범위를 줄이는 것이 핵심!!

COVERING INDEX

1. COVERING INDEX란?

SELECT , WHERE , GROUP BY , ORDER BY 등을 구성하고 있는 컬럼이 모두 인덱스에 포함되어
데이터 페이지로의 random access 가 발생하지 않는 경우

※ 기존에 R4.0 미만의 버전인 경우 COVERING INDEX의 조건을 만족하더라도 항상 데이터 페이지로의
random access 가 발생했음.

2. 실행계획

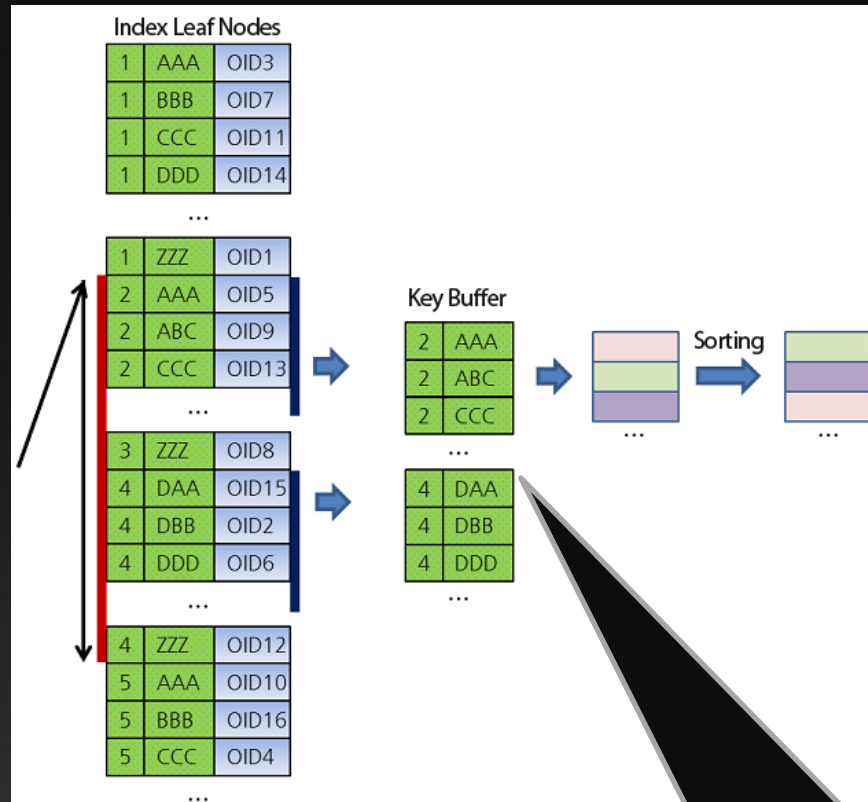
ex) create index on athlete (gender, nation_code)
select gender, nation_code from athlete where gender = 'M' and nation_code = 'ESP'

iscan
class: athlete node[0]
index: i_athlete_gender_nation_code term[0] (covers)
cost: fixed 2(0.0/2.0) var 1(0.0/1.0) card 1

COVERING INDEX

3. 동작방식

SELECT a, b FROM tbl WHERE a > 1 AND a < 5 AND b < 'K' ORDER BY b;



만족하는 데이터가 인덱스 leaf node에 모두 존재
하므로 데이터 페이지로의 접근은 발생하지 않음

PREFIX INDEX

1. PREFIX INDEX란?

컬럼 타입이 string 또는 bit string인 경우 컬럼 값의 앞 일부를 prefix 로 지정하여 인덱스를 생성
 검색하기에 충분한 정도의 길이만 저장하고 나머지는 저장하지 않음으로써 **인덱스 공간 절약**
 단, 정렬조건으로 사용 시 인덱스를 활용한 SKIP ORDERBY 를 활용하지 못하므로 ORDERBY절 명시해야함
 인덱스를 활용하기 위해서는 LIKE 'value%' 를 사용해야 함
 COVERING 안됨.

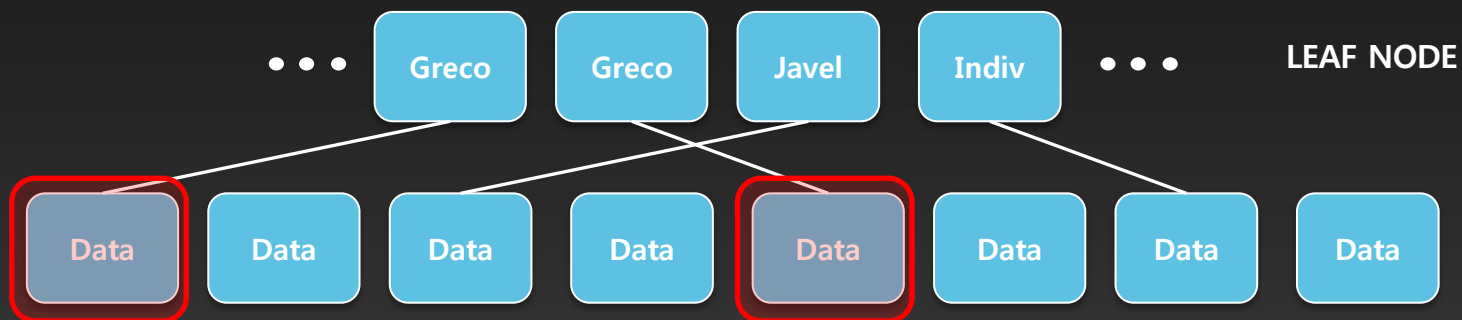
2. 실행계획

ex) create index on event (name(5))
 select name from event where name like 'Greco%'

class: event node[0]
 index: **i_event_name** term[0]

3. 동작방식

i_athlete_name(name (5))



인덱스 leaf node에는 전체 string text가 없으므로 항상 data page 를 access 한다.

NULL 처리

1. CUBRID 에서 인덱스의 NULL 처리?

CUBRID에서는 NULL 값을 **인덱스에 포함하지 않음** → 인덱스 통계 정보에 NULL 의 개수만 관리
단일 컬럼인 경우에는 NULL 값을 인덱스에 포함하지 않고
다중 컬럼인 경우 인덱스를 구성하는 모든 컬럼이 NULL인 경우에는 인덱스에 포함하지 않음.
따라서 IS NOT NULL 혹은 IS NULL 등의 비교 연산은 인덱스를 활용할 수 없음

2. 실행계획

ex) select * from olympic where mascot is not null

```
sscan    class: olympic node[0]  
sargs: term[0]
```

3. 인덱스 활용 TIP

column IS NULL 비교인 경우에는 인덱스 활용이 불가능.

column IS NOT NULL 인 경우 다음과 같이 우회하여 인덱스 활용이 가능

- string type인 경우 : column > ''
- (unsigned) integer type인 경우 : column > 0

1. KEY LIMIT란?

LIMIT 를 이용하여 INDEX SCAN 시 필요한 만큼의 데이터만 읽고 SCAN을 마치는 기법.
쿼리가 사용하는 인덱스가 어떤 것인지 정확히 알고 읽어와야 하는 데이터 건수를 알고있을 경우
KEYLIMIT 를 사용하면 불필요한 INDEX SCAN을 방지할 수 있다.

KEY LIMIT를 사용하지 않을 경우 CUBRID INDEX SCAN 동작 방식 :

SELECT * FROM TABLE WHERE ID > 'A' LIMIT 2

- 1) 쿼리를 수행했을 경우 ID가 있는 인덱스 페이지를 찾아가 ID > 'A' 인 데이터의 OID LIST 를 수집
개수는 DBMS 설정 값 만큼
- 2) 이 중 2개의 OID를 취하고 나머지는 버림

→ OID 를 비 효율적으로 많이 읽게 되므로 R4.0 부터 이를 개선한 KEY LIMIT 동작 방식 출현

2. 실행계획

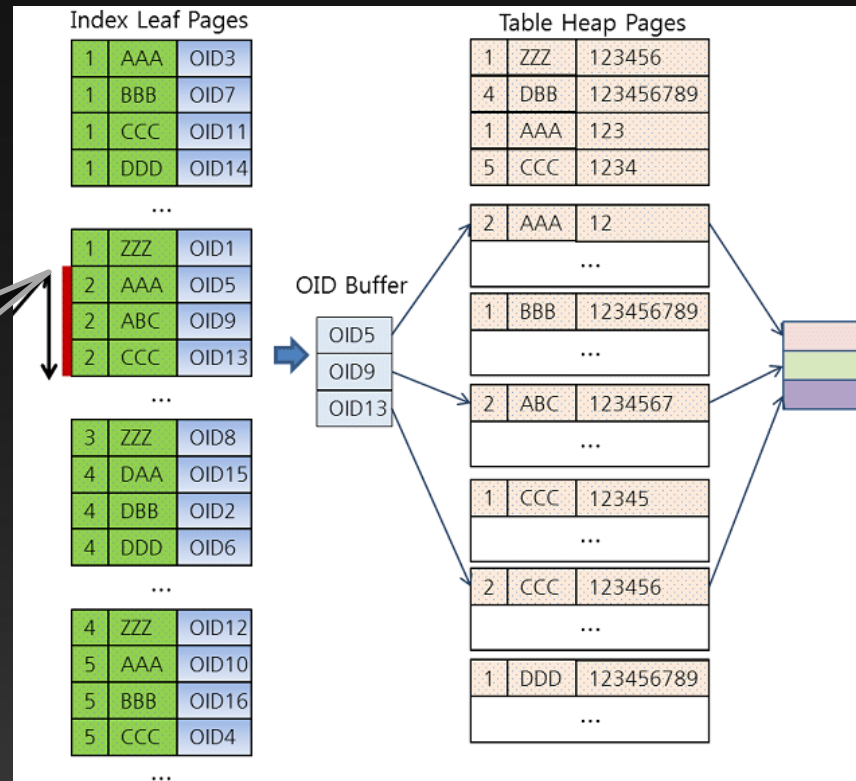
ex) SELECT* FROM athlete WHERE name > 'C' USING INDEX i_athlete_name_gender(+) KEYLIMIT 2;

class: athlete node[0]
index: i_athlete_name_gender keylimit 2 term[0]
cost: fixed 0(0.0/0.0) var 0(0.0/0.0) card 668

3. 동작 방식

```
SELECT * FROM tbl WHERE a = 2 AND b < 'K' ORDER BY b LIMIT 3;
```

a = 2 이고 b < 'K' 인 데이터를 3개 만나면 즉각 스캔 작업을 중단



In-Place Sorting

1. In-Place Sorting 이란?

IN 을 사용한 질의에서 LIMIT 를 사용할 경우 multi range 에 대한 KEY LIMIT 기법을 적용하는 방식

CUBRID는 인덱스 컬럼이 IN절에 사용되면 Key Range를 IN에 사용된 개수만큼 생성하고 각각에 대해 인덱스 스캔을 수행

이 때 LIMIT 와 함께 IN 을 사용할 경우 LIMIT 개수만큼만 Key Range 를 하고 인덱스 스캔을 중단 즉, 각각의 IN 절의 파라미터에 있는 인덱스 스캔에 대해서 KEY LIMIT 최적화가 진행됨

※ 최근 SNS 등에서 내 친구들의 최신 글 모아보기를 할 때 유용하게 사용할 수 있음

2. 동작 방식

ex) `SELECT * FROM tbl WHERE a IN (2, 4, 5) ORDER BY b LIMIT 3;`

- 1) 첫 번째 range(a = 2)에 대한 스캔을 통해 3건의 OID를 확보
- 2) 두 번째 range(a = 4)에 대한 스캔을 시도하는데,
이 range의 첫 번째 키가 1)에서 스캔한 키 값보다 클 경우 바로 스캔을 중단
- 3) 마찬가지로 다음 세 번째 range인 a = 5 에서도 동일하게 동작
- 4) 수집한 3개의 OID 목록으로 Data page 에 접근하여 결과 값을 가지고 옴

In-Place Sorting

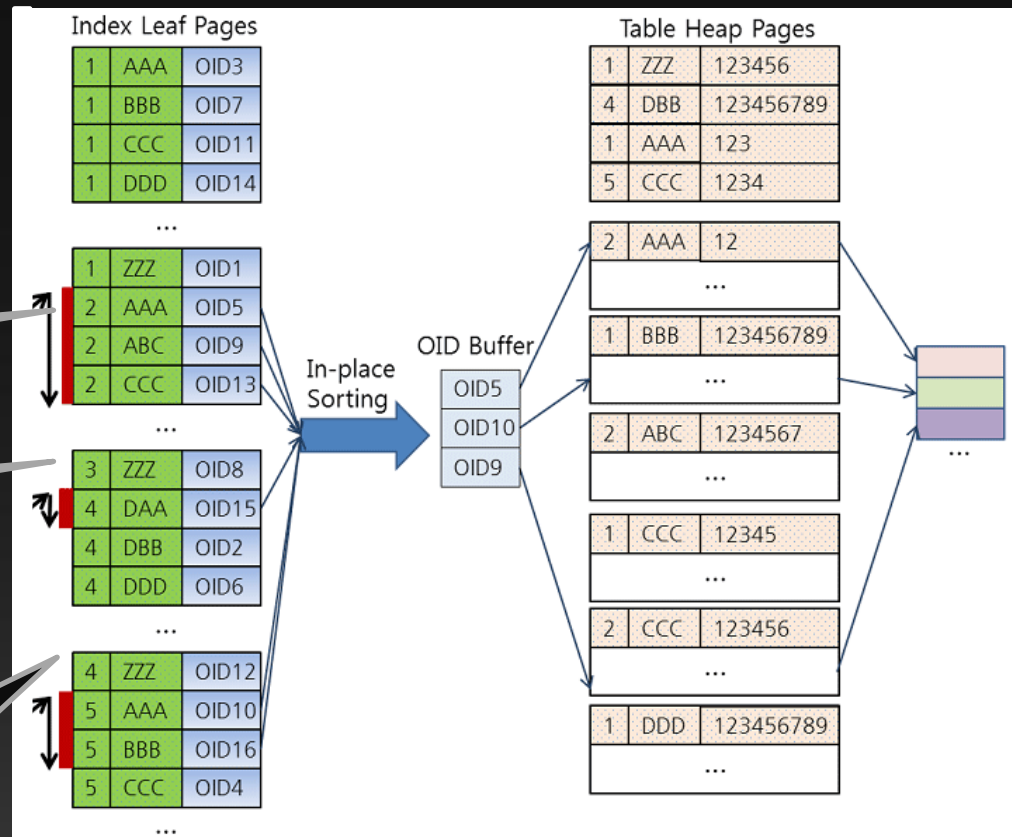
2. 동작 방식

```
SELECT * FROM tbl WHERE a = 2 ORDER BY b LIMIT 3;
```

a = 2 인 b를 3개 수집 후 스캔 중단

a = 4 일 때 b 값이 DAA 로 앞서 수집한 AAA, ABC, CCC 보다 값이 크므로 이후 인덱스는 스캔하지 않고 바로 중단

a = 5 인 경우 인덱스 스캔을 시도. 2,AAA / 2,ABC / 5,AAA 에서 만족하는 3개의 값을 만났으므로 BBB를 만났을 때 스캔 중단
만약 a=5 , B=AAB 였을 경우 계속 스캔



In-Place Sorting

3. 활용 예

요구사항 : 내 친구들의 최근 댓글을 20개 모아서 보여주세요..!!

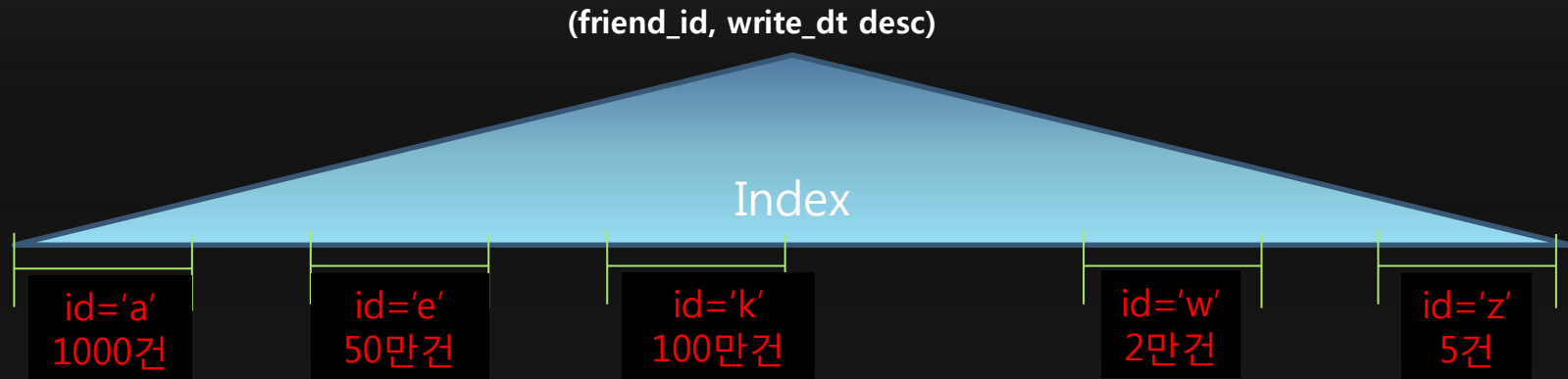
미투데이 쿼리 :

```
SELECT * FROM tbl WHERE friend_id IN ('a','e','k','w','z') ORDER BY write_dt desc LIMIT 20;
```

결과는 ?

DB 죽음

Why?

In-Place Sorting

- MYSQL 및 CUBRID R4.0 미만의 버전에서 IN 쿼리의 동작 방식 :

- 1) id in ('a','e','k','w','z') 에 속하는 100건 + 50만건 + 100만건 + 2만건 + 5건 = 1520105 건을 모두 수집
- 2) 1520105 건을 write_dt 역순으로 정렬
- 3) 최종 데이터 20건을 추출

우회방안 : 아래와 같은 UNION ALL 을 이용해 friend_id 만큼 UNION

```
SELECT * FROM (
SELECT * FROM tbl WHERE friend_id = 'a' ORDER BY write_dt desc LIMIT 20
UNION ALL
SELECT * FROM tbl WHERE friend_id = 'b' ORDER BY write_dt desc LIMIT 20
UNION ALL
...
) as a
LIMIT 20
```

SKIP ORDER BY

1. SKIP ORDER BY란 ?

쿼리문의 정렬 컬럼 순서와 인덱스의 컬럼 순서가 일치할 경우 정렬 작업을 생략하는 기법

2. 실행계획

ex) `SELECT * FROM tbl WHERE a = 2 AND b < 'K' ORDER BY b;`

iscan

class: tbl node[0]

index: idx term[0] AND term[1]

sort: 2 asc

cost: fixed 0(0.0/0.0) var 1(0.0/1.0) card 1

Query stmt:

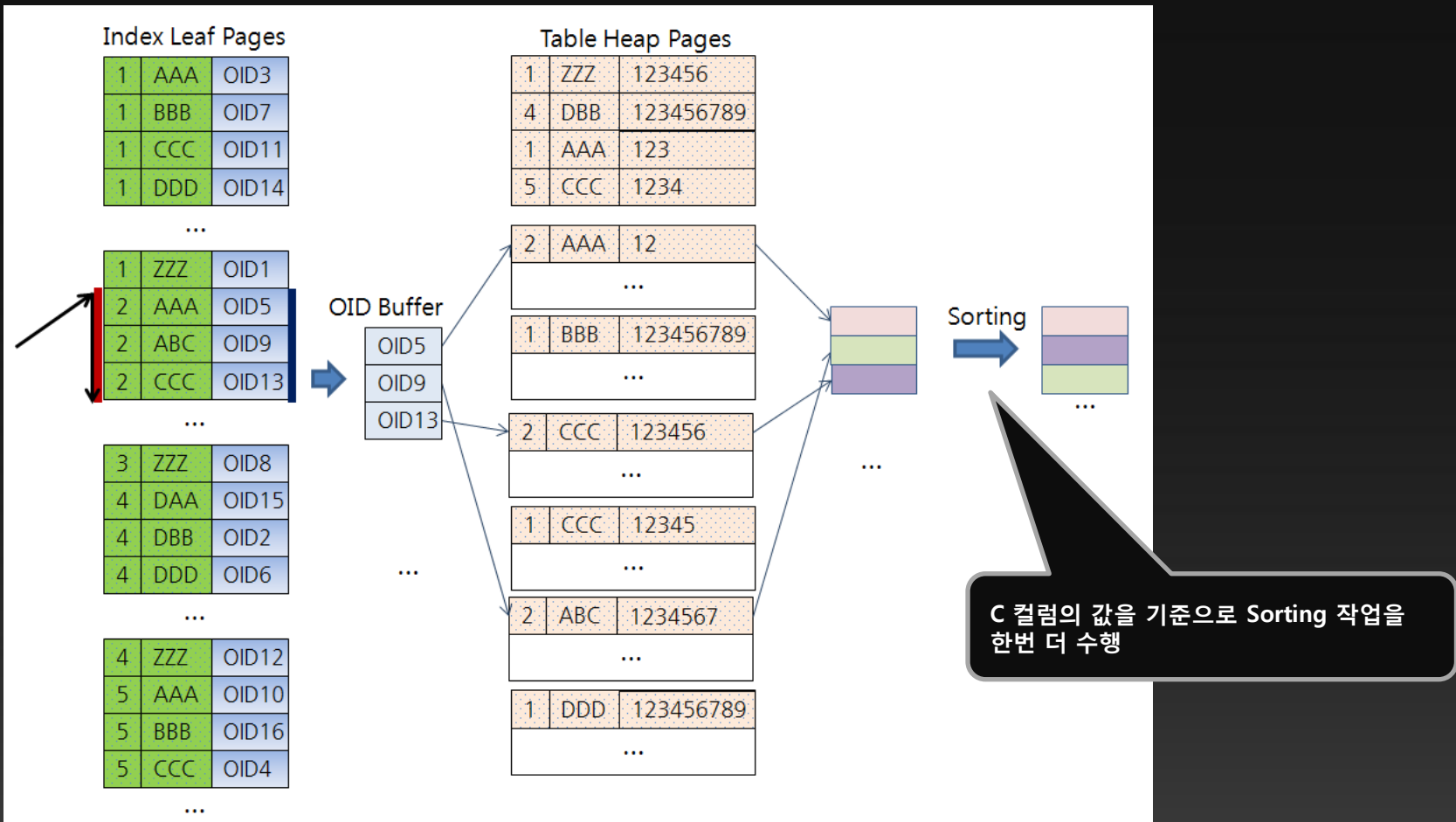
`select tbl.a, tbl.b, tbl.c from tbl tbl where ((tbl.b < ? :0) and tbl.a = ? :1) order by 2`

`/* ---> skip ORDER BY */`

SKIP ORDER BY

3. 동작 방식 – SKIP ORDER BY를 하지 않을 경우

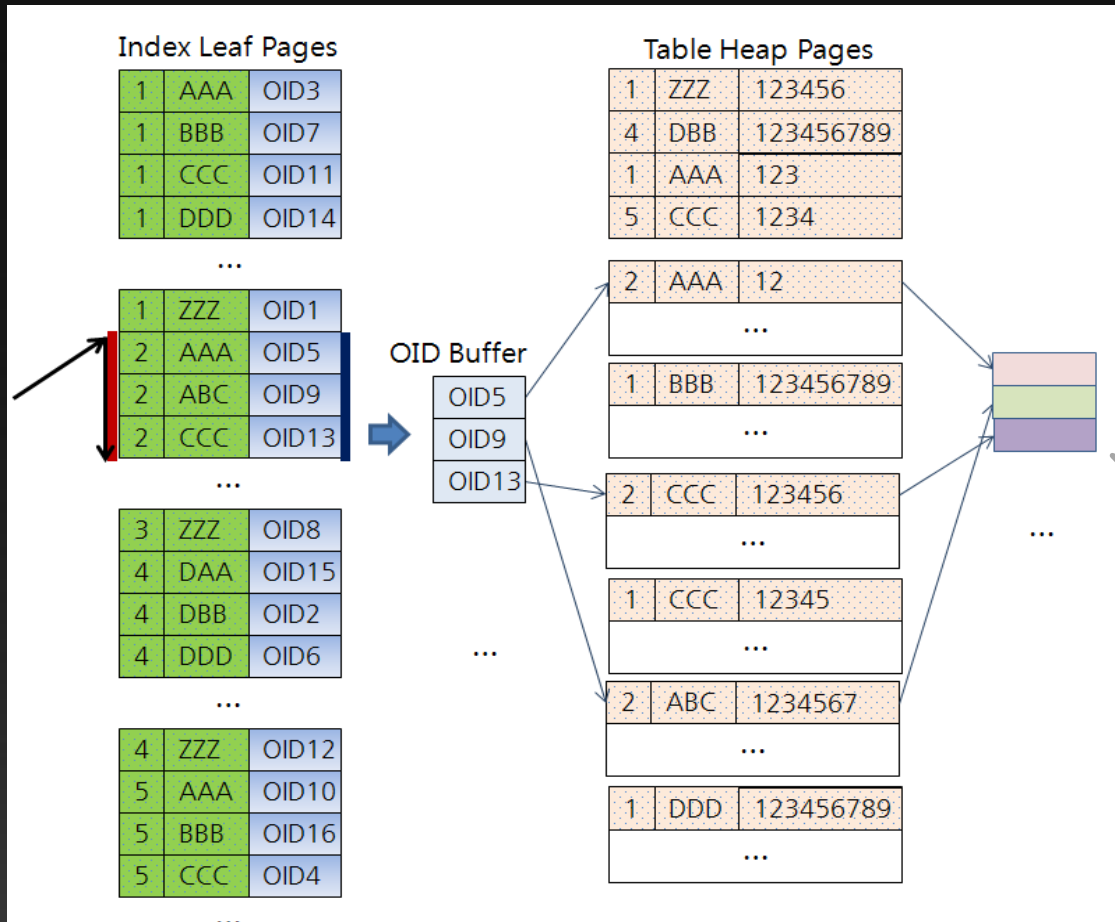
SELECT * FROM tbl WHERE a = 2 AND b < 'K' ORDER BY c ;



SKIP ORDER BY

3. 동작 방식 – SKIP ORDER BY가 있을 경우

SELECT * FROM tbl WHERE a = 2 AND b < 'K' ORDER BY b ;



인덱스 컬럼 순서가 정렬 컬럼과 동일하기 때문에 별도의 sort 작업 생략

SKIP GROUP BY

1. SKIP GROUP BY란 ?

쿼리문의 결과를 특정 컬럼 값을 기준으로 GROUP BY 할 경우 해당 컬럼 및 검색 조건이 인덱스 컬럼의 정렬 순서와 일치하면 GROUP BY를 위한 별도의 정렬 작업을 수행하지 않고 인덱스에 정렬된 값을 활용해서 GROUP BY 결과를 추출하는 기법

2. 실행계획

ex) `SELECT COUNT(*) FROM tbl WHERE a > 1 AND a < 5 AND b < 'K' AND c > 10000 GROUP BY a;`

iscan

```
class: tbl node[0]  
index: idx term[2]  
filtr: term[0]  
sargs: term[1]  
cost: fixed 0(0.0/0.0) var 1(0.0/1.0) card 1
```

Query stmt:

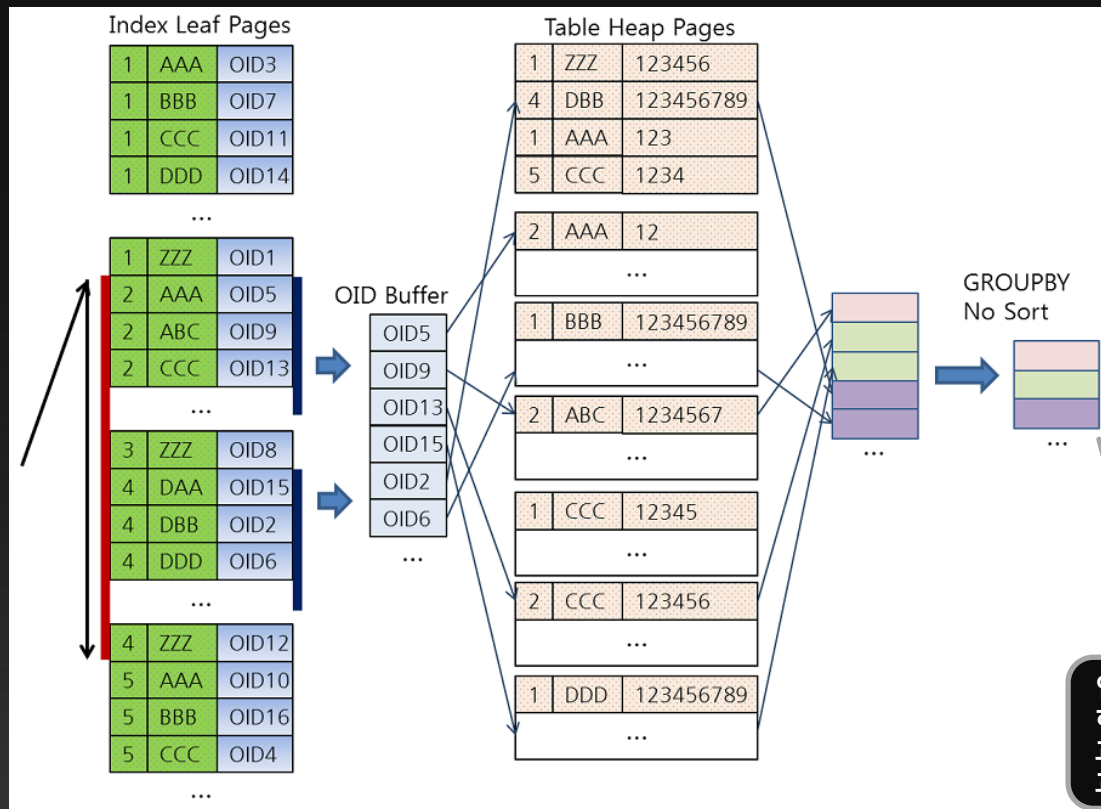
```
select count(*) from tbl tbl where ((tbl.a> ?:0 and tbl.a< ?:1 ) and (tbl.b< ?:2 ) and (tbl.c> ?:3 )) group by  
tbl.a
```

```
/* ---> skip GROUP BY */
```


SKIP GROUP BY

3. 동작 방식

SELECT COUNT(*) FROM tbl WHERE a > 1 AND a < 5 AND b < 'K' AND c > 10000 GROUP BY a;



DESCENDING INDEX SCAN

1. DESCENDING INDEX SCAN 이란 ?

인덱스 정렬 조건 (ex. ASC, DESC) 으로 인덱스가 생성되어있을 경우
그와 반대되는 정렬 조건이 들어왔을 때 INDEX 를 앞에서 뒤로 SCAN하지 않고 뒤에서 앞으로 SCAN
하여 결과를 가져오는 기법

2. 실행계획

ex) select * from tbl where a = 2 and b < 'K' order by b **desc** ;

iscan

class: tbl node[0]
index: idx term[0] AND term[1] (**desc_index**)
sort: 2 asc
cost: fixed 0(0.0/0.0) var 1(0.0/1.0) card 1

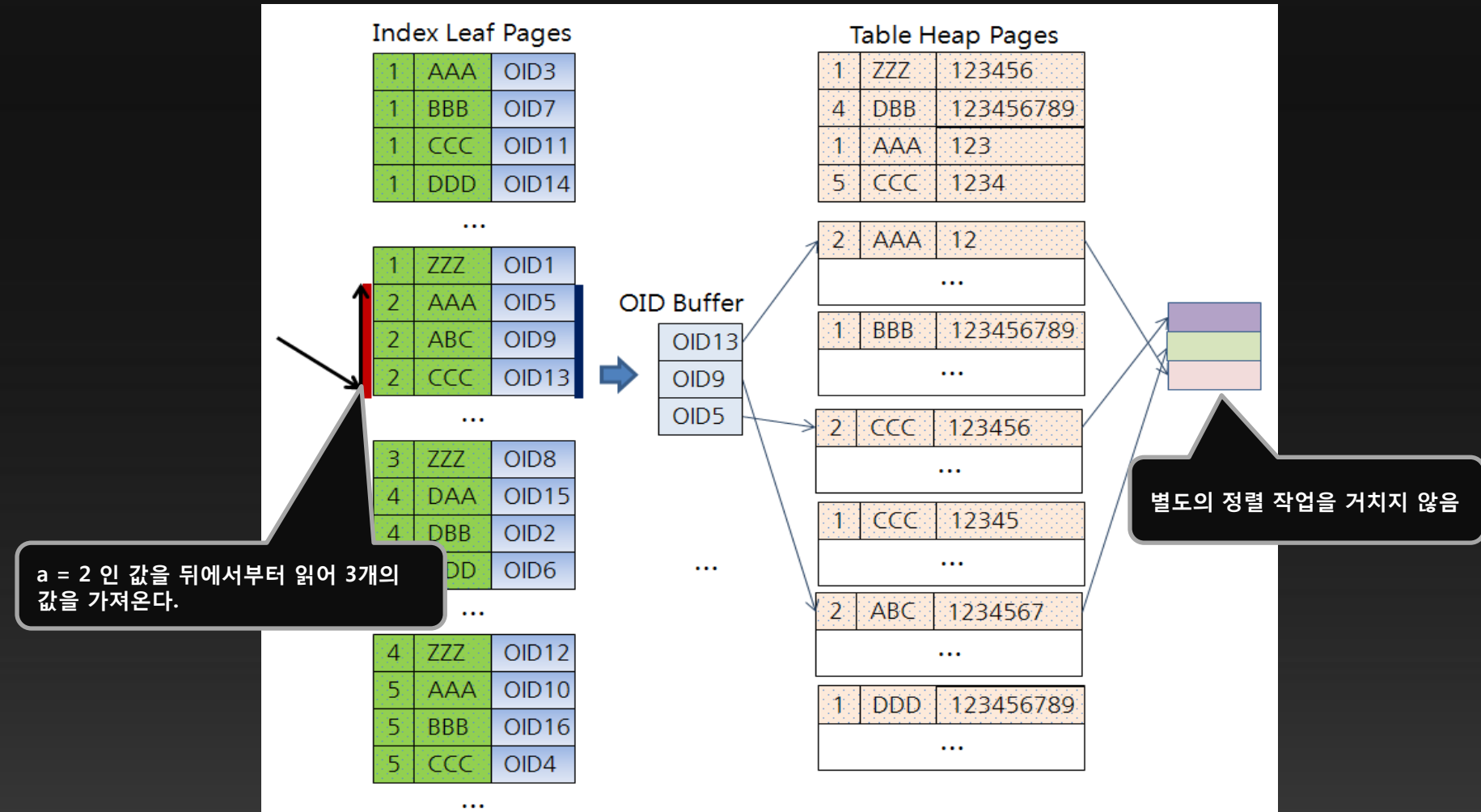
Query stmt:

select tbl.a, tbl.b, tbl.c from tbl tbl where ((tbl.b< ?:0) and tbl.a= ?:1) order by 2 desc

/* ---> skip ORDER BY */

DESCENDING INDEX SCAN

3. 동작 방식



인덱스 사용 시 주의할 점

1. 옵티마이저가 인덱스를 사용하기 위해서는 WHERE절에 반드시 Range 조건이 있어야 함

Range 조건 ? 값의 비교조건. > , < , >= , <= , =

2. WHERE 절에 인덱스 키의 첫 번째 컬럼이 사용되어야 함
3. 대소비교가 아닌 것은 B+ Tree 를 통해 값을 찾을 수 없음
부정형인 경우 index filter

ex1.) <> , != 혹은 IS NOT NULL or IS NULL

ex2.) select * from table where column is not null
select * from table where grade <> 'A'

4. 인덱스 첫 번째 컬럼을 가공하는 경우에도 인덱스를 사용할 수 없음

ex) select * from table where substring(column, 1,4) = '2011'

5. NL-JOIN의 경우 JOIN KEY 가 되는 inner 테이블의 컬럼은 인덱스가 생성되어있는것이 성능에 유리함
6. JOIN KEY 는 서로 데이터타입이 동일해야 함. (자동 형변환으로 인한 성능 저하 가능)
7. 복합 컬럼일 경우 MIN/MAX 함수 사용 시 인덱스 페이지 전체 scan 발생

5.4 인덱스 구성 전략

인덱스 선정 절차

1. 모든 쿼리의 Access Pattern을 분석

- WHERE절, ORDER BY절, GROUP BY절 등

2. 인덱스 구성 컬럼 선정

- 사용 빈도 및 횟수
- 검색 조건의 종류 파악
Exact Match / Range / 부정형
- 데이터 분포도 파악
- Dynamic Query의 경우 필수/옵션 조건 파악
- 컬럼의 READ/WRITE 비율 및 UPDATE 빈도 파악
- JOIN 연결 고리 여부
- ORDER BY, GROUP BY 참여 여부

3. 인덱스 컬럼 선정 및 순서 배치

- Equi 조건 > Range > 부정형 순
- ORDER BY, GROUP BY 컬럼은 맨 뒤에 배치
- Dynamic 의 경우 가장 빈번하게 사용되는 패턴을 위주로 인덱스 구성

인덱스 선정 시 유의사항

1. 새로 추가된 인덱스는 기존 쿼리의 실행계획에 영향을 미칠 수 있음
 - 하나의 인덱스 추가 건이라 하더라도 전체 인덱스 및 Access pattern 분석이 필요
2. 지나치게 많은 인덱스는 데이터 변경에 더욱 많은 비용을 유발할 수 있음
 - 데이터갱신 시 인덱스 페이지도 함께 갱신
 - 여러 인덱스에 추가된 컬럼일 수록 갱신 비용은 높아짐
3. 데이터 분포도가 좋지 않은 컬럼의 경우 index scan의 random access 로 인한 오버헤드가 더 높아질 수 있음
 - index scan의 경우 OID(또는 주소값)을 이용해 data page(혹은 clustered index page) 에 접근
 - 때로는 full table scan (sscan)이 유리한 경우가 있음
4. 대체적으로 INT > BIGINT > NUMERIC > STRING 순으로 검색 속도가 좋음
 - 데이터 사이즈가 I/O 비용에 영향을 줌
 - INT 와 VARCHAR 의 경우 평균적으로 약 10% 의 응답속도 차이를 보임

6. 실행계획 읽기

✓ 실행계획이란?

- DBMS가 SQL을 처리하기 위한 [절차]와 [방법]
- 동일한 SQL에 대해 결과를 낼 수 있는 다양한 처리 방법(실행계획)이 존재할 수 있음
ex) 서울에서 부산까지 가는데 자가용이 빠를까? 버스가 빠를까? 고속도로가 빠를까? 국도가 빠를까?
- 옵티마이저는 다양한 실행계획 중 가장 비용이 적은 실행계획을 선택하게 된다.
ex) 그날의 교통 상황 및 날씨 등등 주변 상황에 따라 [실시간 빠른 길]을 알려주는것



✓ 실행계획의 구성 요소

Join 종류, Join 순서, Join Method, Scan 방법, 쿼리 수행 비용, 기타 연산 등으로 구성

- **Join 종류**

- INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, CROSS JOIN

- **Join 순서**

- Join 작업을 할 때 참조하는 테이블의 순서
ex) From 절에 두 개의 테이블이 있을 경우 $A \rightarrow B$ 로 읽거나 $B \rightarrow A$ 로 읽음
Join 순서는 From절에 나오는 테이블의 $n!$ 개수만큼 존재

- **JOIN Method**

- CUBRID에서 지원하는 Join의 종류 : Nested Loop Join, Sort Merge Join, Index Join
- Index Join : NL Join 에서 조인 컬럼에 Index가 걸려있는 경우

✓ 실행계획의 구성 요소

▪ Scan 방법:

종 류	설 명
Sequential Scan (sscan)	<ul style="list-style-type: none"> - 테이블의 모든 페이지를 read - 테이블 스캔(full scan)을 의미
Temp(List File) Scan (temp)	<ul style="list-style-type: none"> - 실행 결과를 별도의 공간에 저장해서 임시 테이블로 사용함을 의미 - 파생된(Derived) Table (inline view) - Temporary Results <ul style="list-style-type: none"> . Sorting : distinct, union, order by, group by ...
Index scan (iscan)	<ul style="list-style-type: none"> - Index page read + 데이터파일(heap file) access - 인덱스를 탈 수 있는 조건(term)이 있어야 함 - Key 범위 검색과 Key filters 수행

▪ 쿼리 수행 비용

예상되는 CPU/IO 및 cardinality

✓ 실행계획 보는 방법

가장 안쪽의 가장 윗줄부터 읽음 (모든 DBMS 동일)

• 실행계획 보기 형식 : Graphic 보기 / Detail 보기

- Graphic 보기
CUBRID Query Browser 에서 실행계획 보기 아이콘으로 확인

- Detail 보기
Graphic 실행계획 하단에서 확인

1. 실행계획보기 아이콘 클릭 - ①

2. default 는 Graph 보기

② 번 버튼을 클릭하면 Detail보기 가능

3. ③ 번 버튼 클릭 시 다시 Graph 보임

The screenshot shows the CUBRID Query Browser interface. The top pane displays the following SQL query:

```
1 SELECT a.code, a.name, a.gender, g.lastdate
2 FROM athlete a
3 INNER JOIN (SELECT distinct athlete_code,max(game_date) as lastdate
4 FROM game g1
5 WHERE host_year = 2004 and g1.athlete_code = a.code
6 group by athlete_code) as g ON a.code = g.athlete_code
7 WHERE a.nation_code= 'KOR' and a.event= 'Archery'
```

The bottom pane shows the execution plan. The plan is a graph showing the flow of data through various operations. The operations are listed in the left pane, and the right pane shows the details of the operations. The operations are:

- temp(distinct)
- temp(group by)
- idx-join (inner join)
- sscan

The right pane shows the details of the operations, including the CPU usage, the number of rows, and the date. The operations are listed in a table:

유형	테이블	인덱스	검색 조건	CPU (f/v)	#	일자
temp(distinct)				19.5/4.3	1	2011-08-16 18:05:30
temp(group by)				10.2/4.3	2	2011-08-16 18:06:14
idx-join (inner join)	game game	pk_game_host_year_event_code_athlete_code		0.0/5.2	3	2011-08-16 18:07:27
sscan	g			0.0/8.7	4	2011-08-16 18:08:05
sscan	athlete a	pk_athlete_code		0.0/4.3	5	2011-08-16 18:11:43

The bottom pane also shows the join graph segments and terms. The segments are:

- seg[0]: [0]
- seg[1]: athlete_code[0] (f)
- seg[2]: game_date[0] (f)
- seg[3]: host_year[0]

The join graph nodes are:


- node[0]: game game(8653/24) (sargs 0)

The join graph terms are:

- term[0]: (game.host_year=2004) (sel 0.2) (sarg term) (not-join eligible) (indexable host_year[0]) (loc 0)

✓ Graphic 보기

- ICON 설명

이름	아이콘	설명
iscan		index scan
sscan		sequential scan (full scan)
상위 디렉토리		

- 부분 용어 설명

- + 유형 : scan 유형, join 방식
- + 테이블/인덱스/검색조건 : 접근 테이블, 사용한 인덱스, 사용된 검색 조건
- + CPU, IO : 쿼리 수행에 소요된 CPU/IO 비용
- + r : 예상 row 수, p : 예상 page 수

유형	테이블	인덱스	검색 조건	CPU (f/v)	IO (f/v)	전체 (r/p)
temp(group by)				21.7/0.0	2.0/1.0	
idx-join (inner join)				0.0/16.7	2.0/0.0	
nl-join (cross join)				0.0/16.7	0.0/0.0	
iscan	nation n	pk_nation_code		0.0/0.0	0.0/0.0	215/1
term:index			n.code='KOR'			
sscan	athlete a			0.0/16.7	0.0/27.0	6677/27
term:select			a.event='Archery'			
iscan	game g	fk_game_athlete_code		0.0/0.0	2.0/1.0	8653/24
term:index			(a.code=g.athlete_code and n.code='KOR' and g.ho...			
term:select			g.host_year=2004			

✓ Sequential Scan (sscan)

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

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
```

✓ Indexed Scan (iscan)

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

유형	테이블	인덱스	검색 조건	CPU (f/v)	IO (f/v)	전체 (r/p)
iscan	athlete athlete	i_athlete_name		0,0/0,0	2,0/1,0	6677/164
term:index			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 0.000149858) (sarg term) (not-join eligible) (indexable name[0]) (loc 0)

Query plan:

iscan

class: athlete node[0]

index: i_athlete_name term[0]

cost: fixed 2(0.0/2.0) var 1(0.0/1.0) card 1

Query stmt:

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

✓ Nested Loop Join (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 ...(생략)

from olympic olympic, nation nation where olympic.host_nation=nation."name"

Join graph segments (f indicates final):

...생략

Join graph nodes:

node[0]: olympic olympic(25/4)

node[1]: nation nation(215/6)

Join graph equivalence classes:

eqclass[0]: host_nation[0] name[1]

Join graph edges:

term[0]: olympic.host_nation=nation."name"

(sel 0.001) (join term) (mergeable) (inner-join) (loc 0)

✓ Indexed Join (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	2,0/294,0	
sscan	game game			0,0/21,6	0,0/147,0	8653/147
iscan	athlete athlete	pk_athlete_code		0,0/0,0	2,0/1,0	6677/164
term:index			game,athlete_code=athlete,code			

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 2(0.0/2.0) var 1(0.0/1.0) card 6677

cost: fixed 2(0.0/2.0) var 337(43.3/294.0) card 8653

Join graph segments (f indicates final):

...생략

Join graph nodes:

node[0]: game game(8653/147)

node[1]: athlete athlete(6677/164)

Join graph equivalence classes:

eqclass[0]: athlete_code[0] code[1]

Join graph edges:

term[0]: game.athlete_code=athlete.code

(sel 0.000149768) (join term) (mergeable) (inner-join)

(indexable athlete_code[0] code[1]) (loc 0)

Query stmt:

select ...(생략)

from game game, athlete athlete where game.athlete_code=athlete.code

✓ Merge Join (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

✓ Graphic 실행계획

- 비용 = CPU비용 + IO비용
- CPU(f/v) : CPU 비용
 - fixed : 대상 테이블 또는 인덱스를 스캔 하는데 고정적으로 발생하는 비용
 - var : 검색 조건에 따라 달라지는 variable cost
- IO(f/v) : 디스크 I/O 비용
- 전체(r/p)
 - r : 대상 테이블의 레코드 수(cardinality)
 - P : 예상 접근 페이지 수

유형	테이블	인덱스	검색 조건	CPU (f/v)	IO (f/v)	전체 (r/p)
temp(group by)				21.7/0.0	2.0/1.0	
idx-join (inner join)				0.0/16.7	2.0/0.0	
nl-join (cross join)				0.0/16.7	0.0/0.0	
iscan	nation n	pk_nation_code		0.0/0.0	0.0/0.0	215/1
term:index			n.code='KOR'			
sscan	athlete a			0.0/16.7	0.0/27.0	6677/27
term:select			a.event='Archery'			
iscan	game g	fk_game_athlete_code		0.0/0.0	2.0/1.0	8653/24
term:index			(a.code=g.athlete_code and n.code='KOR' and g.ho...			
term:select			g.host_year=2004			

※ Graphic 실행계획에서 보여지는 내용만을 가지고 실제적인 튜닝을 하기에는 정보가 너무 부족하다.
따라서 반드시 Detail 실행 계획을 분석할 수 있어야 한다.

1. 실행계획 예제

쿼리 :

```

SELECT      a.name,max(game_date) as lastdate, n.name
FROM        athlete a
INNER JOIN  game g ON a.code = g.athlete_code
INNER JOIN  nation n on a.nation_code = n.code
WHERE       a.nation_code= 'KOR' and a.event='Archery' and host_year = 2004
GROUP BY    a.name

```

Join graph segments (f indicates final):

- segments (SELECT, WHERE, ORDERBY절 등을 구성하는 컬럼)

Join graph nodes:

- JOIN에 참여하는 node(테이블)

Join graph equivalence classes:

- equi-join일 경우 equi-join node 목록

Join graph edges:

- JOIN edge (JOIN 조건)

Join graph terms:

- 조건 절

Query plan:

- 실행 계획

Query stmt:

- DBMS 내부 rewrite 쿼리

유형	테이블	인덱스	검색 조건	CPU
temp(group by)				14.5
idx-join (inner join)				0.0
iscan	game g	pk_game_host_year_event_code_athlete_code		0.0
iscan	athlete a	pk_athlete_code		0.0

Join graph segments (f indicates final):

```

seg[0]: [0]
seg[1]: name[0] (f)
seg[2]: code[0]
seg[3]: event[0]
seg[4]: nation_code[0]
seg[5]: [1]
seg[6]: game_date[1] (f)
seg[7]: athlete_code[1]
seg[8]: host_year[1]

```

Join graph nodes:

```

node[0]: athlete a(6677/27) (sargs 2 3)
node[1]: game g(8653/24) (sargs 1)

```

Join graph equivalence classes:

```

eqclass[0]: code[0] athlete_code[1]

```

Join graph edges:

```

term[0]: (a.code=g.athlete_code and g.host_year=2004 and a.event='Archery' and a.nation_code='KOR') (sel 0.000149768) (join term[1])
term[1]: g.host_year=2004 (sel 0.2) (sarg term) (not-join eligible) (indexable host_year[1]) (loc 0)
term[2]: a.nation_code='KOR' (sel 0.001) (sarg term) (not-join eligible) (loc 0)
term[3]: a.event='Archery' (sel 0.001) (sarg term) (not-join eligible) (loc 0)

```

Query plan:

```

temp(group by)
subplan: idx-join (inner join)
  outer: iscan
    class: g node[1]
    index: pk_game_host_year_event_code_athlete_code term[1]
    cost: fixed 6(0.0/6.0) var 9(5.2/4.0) card 1731
  inner: iscan
    class: a node[0]
    index: pk_athlete_code term[0]
    sargs: term[2] AND term[3]
    cost: fixed 2(0.0/2.0) var 1(0.0/1.0) card 1
    cost: fixed 8(0.0/8.0) var 18(9.5/8.0) card 1
  sort: 1 asc
  cost: fixed 31(14.5/16.0) var 1(0.0/1.0) card 1

```

Query stmt:

```

select a.[name], max(g.game_date) from athlete a, game g where (a.code=g.athlete_code and g.host_year= 2004 and a.event= 'Archery' and a.nation_code= 'KOR')

```

2. 실행계획 부분 별 분석

A. join graph segment

- 표기법

seg[i] : column_name[j] (f)

- 설명

seg : 컬럼

i : 순서

j : node(table)

select 및 where, order by 등에 나타난 컬럼 목록을 보여줌
(f)inal는 select절에 명시되어있는 컬럼임을 뜻함

- 예제

Join graph segments (f indicates final):

seg[0]: [0]

seg[1]: name[0] (f)

seg[2]: code[0]

seg[3]: event[0]

seg[4]: nation_code[0]

seg[5]: [1]

seg[6]: game_date[1] (f)

seg[7]: athlete_code[1]

seg[8]: host_year[1]

seg[9]: [2]

seg[10]: name[2] (f)

seg[11]: code[2]

2. 실행계획 부분 별 분석

B. join graph nodes

- 표기법

node[i] : table_name t(rows/pages)(sargs j k l ..)

- 설명

node : 테이블

i : 순서

sargs : where절에 참여한 컬럼 목록

j k l ... : Join graph segments 의 seg[j], seg[k], seg[l] 컬럼

rows : 해당 테이블의 전체 예상 데이터 건수

pages : 해당 테이블의 전체 예상 페이지 수

- 예제

Join graph nodes:

node[0]: athlete a(6677/27) (sargs 4 5)

node[1]: game g(8653/24) (sargs 2)

node[2]: nation n(215/1) (sargs 3)

2. 실행계획 부분 별 분석

C. Join graph equivalence classes

- 표기법

eqclass[i]: column[j] column[k]

- 설명

eqclass : equi-join 의

equi-join 인 경우에만 나타나는 부분
equi-join 의 edge(연결 컬럼)을 보여줌

i : node 순서 (Join graph nodes 의 node 번호)

j,k .. : Join에 참여하는 컬럼 목록 (Join graph segments 의 컬럼 목록)

- 예제

Join graph equivalence classes:

eqclass[0]: code[0] athlete_code[1]

eqclass[1]: nation_code[0] code[2]

2. 실행계획 부분 별 분석

D. Join graph edges/join graph terms

- 표기법

term[i]: column1=column2 (sel) (rank) (join term, sarg term) (join_type)
(join_method column1[j] column2[k]) (loc 0)

- 설명

tem : join 혹은 where절 등에 참여한 검색 조건

column1, column2 : join에 참여한 컬럼

(sel) : selectivity

(rank) : 수행하는 비용에 대한 순위 (sel 이 같을 경우 rank 가 낮은 조건을 먼저 적용

(join_term) : join term/ not-join eligible/ dummy join

- join term : join edge로 사용되었을 경우

- not-join eligible : join이 아닌 where절 등에 조건으로 온 term일 경우

- dummy join : outer join이 있는 경우 JOIN 적용 순서

ex) A outer join B on a=b outer join C on a=c 의 쿼리를 실행할 경우 b=c 라는 dummy outer join 생성

(sarg term) : join edge로 쓰이지 않고 순수하게 where절에서 사용된 컬럼일 경우

즉, 테이블 1개와만 관련이 있는 경우(2개 : join term, 3개 이상 : others term)

(join_type) : join term일 경우 join의 종류를 나타낸다. (left/right/inner)

(loc) : outer join이 있는 경우 적용 순서를 표시한다. (inner join일 경우 모두 0으로 나타남)

(join_method) : 가능한 join method (mergeable, indexable)

2. 실행계획 부분 별 분석

E. Query plan

- 표기법

```

outer : join-method (join-type)
      outer : scan_type
            class : class_notation node[i]
            index : index_name term[j], term[k] ...
            filter : term[l] ...
            sargs : term[m] ...
            cost : fixed i(CPU/IO) var j (CPU/IO) card k
      inner : scan_type
      ...
sort n,o,p ...
cost : fixed, var

```

- 설명

실행계획은 “가장 안쪽의 가장 윗 줄부터 읽음”

outer : 선행 테이블 (driving), inner : 후행 테이블

node[i] : 테이블의 node 번호

index : index 를 활용했을 경우 나타남. term[j], term[k] ... : join(혹은 where절)을 수행할 때 사용한 인덱스 컬럼

filter : 인덱스의 중간에 있는 컬럼에 대한 조건이 없거나 선행 조건이 range일 경우 index filtering을 수행

sargs : index를 활용하지 못하고 data page filtering 을 수행한 조건

cost(1) : 해당 테이블의 scan하는데 소요된 cost

fixed : 해당 테이블 또는 인덱스를 스캔하는데 고정적으로 발생하는 비용

var : 검색조건에 따라 달라지는 variable cost

card : return되는 row 수

2. 실행계획 부분 별 분석

E. Query plan

- 표기법

```
outer : join-method (join-type)
      outer : scan_type
            class : class_notation node[i]
            index : index_name term[j], term[k] ...
            filter : term[l] ...
            sargs : term[m] ...
            cost(1) : fixed i(CPU/IO) var j (CPU/IO) card k
      inner : scan_type
      ...
sort n,o,p ...
cost(2) : fixed, var
```

- 설명

sort : 정렬 조건이 있는 경우 sort 에 참여한 컬럼 목록
cost(2) : JOIN 을 수행한 후의 cost

※ cost란?

절대적인 수치가 아니라 페이지 하나를 디스크에서 읽어오는데 드는 비용을 1이라 정하고 그에 따른 상대적인 값을 보여주는 수치임.

따라서 수행하는 서버의 사양이나 페이지 크기에 따라 동일한 쿼리라 할지라도 cost는 달라질 수 있음.

2. 실행계획 부분 별 분석

E. Query stmt

- 표기법

`select from ... where ... /* → skip order by */`

- 설명

실제 rewrite된 쿼리

`skip order by` : 인덱스를 이용하여 return한 순서와 `order by` 컬럼 순서가 일치하여 정렬 작업이 생략되었을 경우 나타난다.

✓ 물리/논리 페이지 IO 확인

- 구문

```
SET @collect_exec_stats = 1;           // 세션 변수 설정 (통계정보 수집)
SELECT ...                             // 쿼리 수행
SHOW EXEC STATISTICS;                  // 통계 정보 출력
```

- 결과

```
data_page_fetches           // 데이터 버퍼 페이지를 fetch(조회) 한 수
data_page_dirtyes           // 데이터 버퍼 페이지를 dirty(변경) 한 수
data_page_ioreads            // 데이터 버퍼에 올리기 위해 디스크에서 읽은 수
data_page_iowrites           // 데이터 버퍼에서 디스크로 쓴 수
```

✓ INDEX HINT 사용법

- 구문

```
SELECT ... FROM ... WHERE ... [USING INDEX {NONE | index_spec [{, index_spec } ...] }  
UPDATE ... SET ... WHERE ... [USING INDEX {NONE | index_spec [{, index_spec } ...] }  
DELETE ... FROM ... WHERE ... [USING INDEX {NONE | index_spec [{, index_spec } ...] }
```

- 동작 방식

- + 순차스캔비용 > 인덱스 스캔 비용일 경우 인덱스 스캔 사용
- + USING INDEX ... 절에 인덱스 목록을 나열할 경우 지정된 인덱스만 대상으로 스캔 비용 산출
USING INDEX 에 지정한 인덱스의 비용이 나쁘다고 판단할 경우 순차 스캔을 진행함.
- + 인덱스 스캔 강제 지정 : USING INDEX 인덱스명(+)
인덱스명 뒤에 (+) 를 붙이면 해당 인덱스의 비용은 0으로 계산
- + 강제로 순차 스캔하도록 지정하려면 : USING INDEX NONE 구문 이용

- 특징

- + 인덱스명을 잘못 입력했을 경우 parsing error 발생

✓ JOIN HINT 사용법

- 구문

```
SELECT /*+ HINT_EXPRESSION [{hint} ...] */ FROM ... WHERE ...
```

- HINT_EXPRESSION 종류

+ USE_NL: nested loop 실행계획을 생성

+ USE_IDX : 인덱스 조인 실행계획을 생성

+ USE_MERGE : merge join 실행계획을 생성

+ ORDERED :

FROM절에 명시된 클래스의 순서대로 조인하는 실행계획을 만든다.

FROM절에서 앞에 오는 테이블이 outer 가 되고 뒤에 오는 테이블이 inner가 된다.

+ USE_DESC_IDX : 내림차순으로 인덱스를 스캔

+ NO_COVERING_IDX : 커버링 인덱스 기능을 사용하지 않음

+ QUERY_CACHE(1) : 해당 질의에 대해서만 query result cache 사용

- 특징

+ USE_NL과 USE_MERGE가 함께 지정된 경우 주어진 힌트는 무시된다.

+ 구문을 잘못 입력하면 주석으로 처리되고 무시된다.

+ JOIN HINT 의 비용이 높을 경우 힌트는 무시된다. (단, ORDERED 의 경우는 항상 동작)

```
SELECT a.name,max(game_date) as lastdate, n.name
FROM athlete a
INNER JOIN game g ON a.code = g.athlete_code
inner join nation n on a.nation_code = n.code
WHERE a.nation_code= 'KOR' and a.event='Archery' and host_year = 2004
group by a.name
```

temp(group by)	-- 16. temp를 사용하여 group by
subplan: idx-join (inner join)	-- 14. g를 join 하는데 idx-join 으로 처리됨
outer: nl-join (cross join)	-- 8. n과 a node는 nl-join으로 처리
outer: iscan	
class: n node[2]	-- 1. n node가 outer로 사용됨
index: pk_nation_code term[3]	-- 2. pk를 사용해서 term[3] 처리
cost: fixed 0(0.0/0.0) var 0(0.0/0.0) card 1	-- 3. 예상 결과 건수는 1건
inner: sscan	
class: a node[0]	-- 4. inner 테이블은 a node
sargs: term[4] AND term[5]	-- 5. 데이터 필터링으로 term[4,5] 처리
cost: fixed 0(0.0/0.0) var 44(16.7/27.0) card 1	-- 6. 예상 결과 건수는 1건
cost: fixed 0(0.0/0.0) var 17(16.7/0.0) card 1	-- 7. 두 테이블을 join하는데 드는 비용 및 card
inner: iscan	
class: g node[1]	-- 9. 두번 째 join 테이블은 g node
index: fk_game_athlete_code term[1]	-- 10.fk 인덱스 사용해서 term[1] 처리
sargs: term[2]	-- 11. term[2] 는 데이터 필터링
cost: fixed 2(0.0/2.0) var 1(0.0/1.0) card 1731	-- 12.예상 결과 건수는 1건
cost: fixed 2(0.0/2.0) var 17(16.7/0.0) card 1	-- 13. n join a 결과와 g 를 join하는데 든 비용
sort: 1 asc	-- 14. 1번째컬럼(a.name) 으로 sorting
cost: fixed 24(21.7/2.0) var 1(0.0/1.0) card 1	-- 15. 전체 sorting 까지 드는 비용

7. 그리고 몇 가지 오해와 진실

7.1 WHERE절에 조건은 많을 수록 좋다? → NO!! 옵티마이저 혼동을 유발하지 맙시다!

대외비

COLUMN :

a int primary key

b int

c string

d string

e int

f int

INDEX : (b, c, d)

```
SELECT COLUMN_LIST
FROM TABLE
WHERE
b = ? and
c = ?
```

```
SELECT COLUMN_LIST
FROM TABLE
WHERE
b = ? and
c = ? and
a > 0
and e > 0
```

iscan

class: table node[0]

index: ink01_table term[b] AND term[c]

iscan

class: table node[0]

index: ink01_tbl_message term[b] AND term[c]

sargs: term[a] term[e]

7.2 WHERE절의 컬럼에 인덱스가 있으면 성능은 문제 없다? → NO! Index Filer를 제거하는 방향으로 인덱스를 “잘” 잡아야 성능이 향상된다!

COLUMN :

a int primary key

b int

c string

d string

e int

f int

INDEX : (b, c, d)

SELECT COLUMN_LIST

FROM TABLE

WHERE

b = ? and

c = ? and

d = ?

SELECT COLUMN_LIST

FROM TABLE

WHERE

b = ? and

c > ? and

d = ?

iscan

class: tbl node[0]

index: ink01_tbl term[0] AND term[1] AND term[2]

iscan

class: tbl node[0]

index: ink01_tbl term[0] AND term[2]

filtr: term[1]

C의 조건이 range, IN clause인 경우에도 마찬가지

7.3 쿼리 RETURN 건수가 많아 페이징 처리를 했으니, 성능이 좋아지겠죠? → NO! 인덱스 안 타면 소용 없어요!

COLUMN :
a int primary key
b int
c string
d string
e int
f int
g timestamp

INDEX : (b, c, d)

// 원본 쿼리

```
SELECT COLUMN_LIST
FROM TABLE
WHERE
b = ? and
c = ?
order by g desc
```

temp(order by)

```
subplan: iscan
class: tbl node[0]
index: ink01_tbl term[b] AND term[c]
sort: d asc
cost: fixed 0(0.0/0.0) var 1(0.0/1.0) card 0
```

```
sort: g desc
cost: fixed 6(5.0/1.0) var 1(0.0/1.0) card 0
```

// 페이징 처리

```
SELECT COLUMN_LIST
FROM TABLE
WHERE
b = ? and
c = ?
order by g desc limit 0,10
```

temp(order by)

```
subplan: iscan
class: tbl node[0]
index: ink01_tbl term[b] AND term[c]
sort: d asc
cost: fixed 0(0.0/0.0) var 1(0.0/1.0) card 0
```

```
sort: g desc
cost: fixed 6(5.0/1.0) var 1(0.0/1.0) card 0
```

7.4 인덱스를 전부 타면 성능이 빨라지겠죠? → NO! 성능의 열쇠는 I/O 최소화! 테이블 수직 분할을 통해 SELECT하는 ROW 크기를 줄이고, UPDATE할 때 생성되는 로그를 줄이자!

TBL1:
a int primary key
b int
c int
d varchar(10) -- 10byte string

TBL2:
a int primary key
b int
c int
d string -- 약 1KB string
e string -- 약 1KB string

```
SELECT *  
FROM TBL1  
WHERE a < 10;
```

```
SELECT *  
FROM TBL2  
WHERE a < 10;
```

iscan
class: TBL1 node[0]
index: pk_tbl1_a term[0]

iscan
class: TBL2 node[0]
index: pk_tbl2_a term[0]

show exec statistics;

'data_page_fetches'	24
'data_page_dirties'	2
'data_page_ireads'	0
'data_page_iowrites'	0

show exec statistics;

'data_page_fetches'	6559
'data_page_dirties'	3692
'data_page_ireads'	0
'data_page_iowrites'	0

SELECT 목록에 포함되지 않는 컬럼이 Long String인 경우, 테이블을 수직 분할 → Fetching 페이지 I/O 최소화
UPDATE시 특정 컬럼만 빈번하게 변경하는 경우, 테이블을 수직 분할 → 로그 페이지 I/O 최소화

7.5 데이터가 10만 건도 안되니, 성능 문제는 없겠죠? → NO! 조인 연산 중에는 수천억 건을 연산할 수도 있어요!

TBL1: 1,000 rows
TBL2: 10,000 rows
TBL3: 100,000 rows

QUERY

```
SELECT COLUMN_LIST  
FROM TBL1, TBL2, TBL3  
WHERE TBL1.COLUMN BETWEEN ? AND ?
```

STEP1.

TBL1.COLUMN BETWEEN ? AND ?
→ **result1**

→ 500 rows

STEP2.

result1 cross join TBL2
→ **result2**

→ 500 rows * 10,000 rows
= 5,000,000

STEP3.

result2 cross join TBL3

→ 5,000,000 rows * 100,000 rows
= 500,000,000,000

8. 튜닝 초보자를 위한 몇 가지 TIP

8.1 실행계획에 관심을 가져주세요.

대외비



실행계획은 DBMS와 대화하기 위한 언어

8.2 테이블의 인덱스 구조에 관심을 가져보세요.

- ✓ 조건 절이 인덱스를 효율적으로 활용하는가?

인덱스 컬럼 순서 조정이나 컬럼 추가 등으로 index filter , data filter 를 없앨 만한 부분은 없는지 확인

- ✓ 별도의 정렬 비용 없이 인덱스를 활용하여 정렬이 가능한가?

실행계획에서 SKIP_ORDERBY 를 확인

- ✓ 조건절은 Multi range optimization을 할 수 있도록 작성되어 있는가?

index : a, b, c, d, e desc

조건 : a = ? and b = ? and c = ? and d in (?, ?, ?) order by e desc limit 10;

- ✓ 불필요한 컬럼을 SELECT 하여 Covering Index 를 활용할 수 있음에도 불구하고 data page로의 random access를 유발하고 있지는 않은가?

- ✓ NL-JOIN 시 inner 테이블의 JOIN KEY가 되는 컬럼에는 인덱스가 적절하게 생성되어 있는가?

8.3 활용 가능한 인덱스가 없을 경우 데이터를 잘 분석해보세요.

- ✓ 정렬 컬럼이 인덱스를 활용할 수 없는 경우 해당 컬럼과 같은 속성의 데이터를 가진 다른 컬럼은 없는가?

ex) 날짜 컬럼 대신 auto_increment 컬럼 사용

- ✓ 조건 절에 만족하는 컬럼이 인덱스에 없을 경우 같은 속성의 다른 컬럼을 활용할 수는 없을까?

ex) NHN에서 실제 사용하는 댓글 테이블의 예

스키마 :

게시글번호 (PK)

댓글번호(PK)

상위댓글번호

최상위 댓글 번호

댓글 정렬 컬럼

작성자ID

작성일시

인덱스1 : 게시글번호, 댓글번호(PK)

인덱스2 : 게시글번호 , 최상위 댓글 번호DESC, 댓글정렬컬럼 ASC

댓글 최신순 보기 쿼리 :

```
SELECT *
FROM TABLE
WHERE 게시글번호 = ?
ORDER BY 최상위댓글번호 DESC, 정렬 컬럼 ASC
```


8.3 활용 가능한 인덱스가 없을 경우 데이터를 잘 분석해보세요.

대외비

추가 요구사항 :

알리미 서비스를 위해 내가 작성하지 않은 댓글 중 가장 최근에 작성된 댓글의 작성 일시를 알려주세요!!

```
SELECT 작성일시  
FROM TABLE  
WHERE 게시글번호=? AND 상위댓글번호=0 and 작성자ID <> 'naverid'  
ORDER BY 날짜 DESC  
LIMIT 1;
```

위 쿼리에서 data filter 와 index filter의 개수는?
이를 줄이기 위해서는 어떻게 하면 될까?

댓글 정렬 컬럼의 데이터 생성 규칙을 파악하면 답을 알 수 있다!!

답글이 없는 댓글의 댓글정렬컬럼 = '!!!!!!!!!!!!!!!!!!!!'

최상위 댓글 번호 DESC로 정렬하여 만나는 첫번째 데이터 = 가장 최근에 작성한 댓글 데이터

```
SELECT 작성일시  
FROM TABLE  
WHERE 게시글번호=? AND 댓글정렬컬럼 = '!!!!!!!!!!!!!!!!!!!!' and 작성자ID <> 'naverid'  
ORDER BY 날짜 DESC  
LIMIT 1;
```

8.4 쿼리를 바라보는 시각을 조금만 바꿔보세요.

- ✓ 화면에 반드시 필요한 컬럼인가?
- ✓ 반드시 필요한 JOIN 인가?
- ✓ 불필요한 ROUND TRIP을 하고있지는 않은가?
- ✓ 잦은 COUNT(*) , MIN/MAX 쿼리.. 반정규화 해야 할 조건은 없는가?
- ✓ SELECT 목록 화면에 나타나지 않는 string 컬럼, 테이블을 분할해야 할 필요는 없을까?
- ✓ 테이블의 50개 컬럼 중 유독 1개의 컬럼만 집중적으로 UPDATE가 발생. 문제 없는걸까?

8.5 OUTER TABLE의 access 범위를 줄이는 것을 고민해보세요.

✓ NESTED LOOP JOIN동작 방법 (JOIN 연결고리에 인덱스가 존재할 경우)

- STEP1 . 선행(Outer) 테이블의 인덱스에서 조건 절에 맞는 데이터 추출
- STEP2. data page 에서 sargs 를 filter
- STEP3. JOIN KEY 를 이용하여 후행(Inner) 테이블에 접근
- STEP4. index scan, key filter, data filter 수행
- STEP5. 최종 결과를 메모리에 쓰고 주소값을 client에 전달

➔ 선행 테이블의 데이터 개수 만큼 후행 테이블에 접근하므로 선행 테이블에서 추출된 데이터 건수가 일의 양을 결정

선행 테이블의 데이터 건수를 줄이기 위한 방법

1. 조건절 추가
2. INLINE VIEW 를 이용하여 선행 테이블의 데이터 건수를 줄임
3. FROM 절의 테이블을 WHERE절의 SUB QUERY (IN, Exists) 로 변경 선행 테이블의 건수 줄임

8.6 SUB QUERY? 신중히 생각해서 사용해주세요.

- ✓ DBMS마다 optimizing 방식이 달라 동일한 sub query도 수행하는 방식에는 차이가 있지만 대체적으로 SUB QUERY 는 SUB QUERY별 특성 및 데이터 특성을 정확하게 파악하지 못한 사용자에게 의해 작성되었을 경우 불필요한 연산을 유발하는 패턴이 자주 발견됨.

ex) SELECT TOTAL.*
 FROM (
 SELECT 컬럼 목록
 FROM (SELECT * FROM TABLE1 WHERE ...) as A
 INNER JOIN (SELECT * FROM TABLE2 WHERE ...) as B
 LEFT OUTER JOIN (SELECT * FROM TABLE3 WHERE ..) as C
) AS TOTAL
 WHERE TOTAL.XXX > ?
 ORDER BY TOTAL.COL1, TOTAL.COL2

- ✓ SUB QUERY 를 사용할 경우 JOIN의 순서나 실행계획을 사용자가 원하는데로 제어하는 것이 쉽지 않음

ex) sub query 부분이 가장 먼저 수행되어야 하는 경우 HINT 로 제어할 수 있는 방법이 없음

- ✓ SUB QUERY별 특징 (데이터복제 여부, 동작방식), 데이터 분포도 등에 따라 적절한 SUB QUERY를 사용해야 함.

ex) SUB QUERY 로 Driving table 의 범위를 줄일 수 있는 경우

ex) JOIN KEY에 마땅한 인덱스가 없으나 다른 조건에 해당하는 인덱스는 있는 경우

8.7 SUB QUERY와 JOIN 을 자유롭게 변경할 수 있도록 연습해보세요.

대외비

✓ JOIN → INLINE VIEW로 변경하여 KEYSET을 줄여 성능을 향상한 경우 (TABLE1 : TABLE2 = 1:N)

<< 원본 쿼리 >>

```
SELECT DISTINCT(A.ID)
FROM TABLE1
LEFT OUTER JOIN TABLE2 ON A.ID = B.ID
WHERE B.ID IS NULL;
```

TABLE1 : TABLE2 = 1 : N 관계

실행 순서 :

TABLE1 을 full scan → TABLE2 와 LEFT OUTER JOIN (데이터 중복발생)
→ B와 JOIN 하면서 A에 존재하지 않는 데이터를 filtering
→ 전체 데이터 추출이 끝난 후 A.ID 의 중복을 제거

<< 변경 쿼리 >>

```
SELECT A.ID
FROM (SELECT DISTINCT ID FROM TABLE1 ) as A
WHERE NOT EXISTS
(SELECT * FROM TABLE2 as B WHERE A.ID = B.ID)
```

실행 순서 :

TABLE1 의 DISTINCT한 ID 값을 추출
→ 추출한 ID의 distinct한 개수만큼 TABLE2에 접근하여 비교 연산 수행

8.7 SUB QUERY와 JOIN 을 자유롭게 변경할 수 있도록 연습해보세요.

✓ SUB QUERY → JOIN 으로 변경하여 성능을 개선한 경우 (TABLE1 : TABLE2 가 1:1 관계일 때)

<< 원본 쿼리 >>

```
SELECT (SELECT COL1 FROM TAB2 WHERE T2.KEY = T1.KEY) as COL1
      ,(SELECT COL2 FROM TAB2 WHERE T2.KEY = T1.KEY) as COL2
FROM TABLE1
WHERE COLUMN between ? and ? ;
```

실행 순서 :

TABLE1 의 결과 set 생성 : result1

→ 첫번 째 sub query 에서 result1 만큼 random access하여 COL1 가져옴

→ 두번 째 sub query 에서 result1 만큼 random access하여 COL2 가져옴

<< 변경 쿼리 >>

```
SELECT TAB2.COL1, TAB2.COL2
FROM TABLE1
LEFT OUTER JOIN TABLE2 ON T1.KEY = T2.KEY
WHERE COLUMN between ? and ? ;
```

실행 순서 :

TABLE1 의 결과 set 생성

→ result1 의 결과 값으로 TABLE2 에 NL-JOIN을 수행하여 결과 return

8.8 각 DBMS별 연산자/함수의 동작 방식에 대해 공부해보세요.

대외비

✓ UNION 과 UNION ALL

UNION ALL : 정렬 없음

UNION : 정렬 일어남

✓ multi column index 일 때의 MIN/MAX 의 동작 방식 및 우회 방안

index : (a,b)

select max(b) from table where a = ? → a=?인 인덱스를 전체를 읽고 max 인 값을 찾아냄

우회 쿼리 :

select /*+ USE_IDX_DESC */ b from table where a = ? Limit 1

✓ REPLACE 와 ON DUPLICATE KEY UPDATE

REPLACE : delete 후 insert

ON DUPLICATE KEY UPDATE : insert 및 중복 행 update

단건 update의 경우 on duplicate key update 가 비용이 싸다!

✓ IN/Exists/Scalar Subquery/Inline-view

IN :

EXISTS :

Thank you.