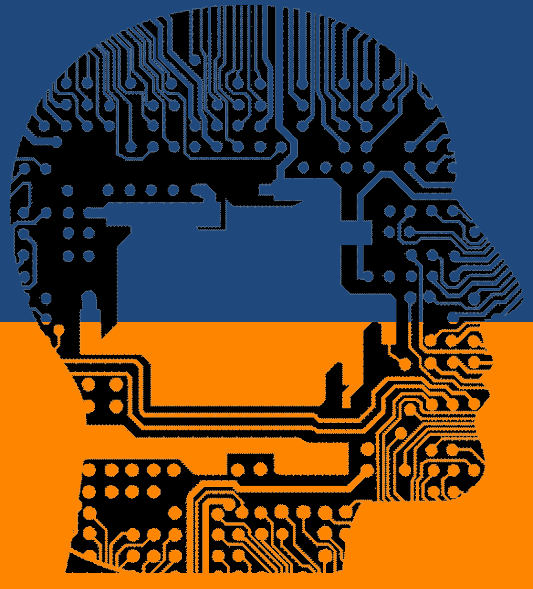
한국IT진흥부설 정보보호교육학원 아이섹

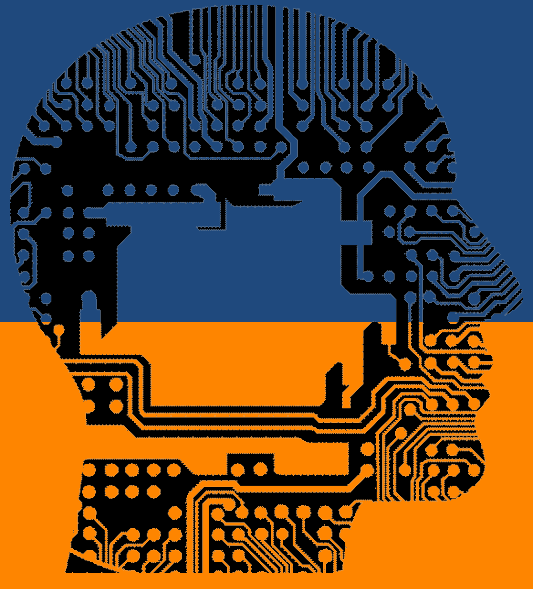


JAVA 웹 개발자 양성과정
DataBase - SQLD

5강 - SQL최적화

By SoonGu Hong

한국IT진흥부설
정보보호교육학원 아이섹



JAVA 웹 개발자 양성과정
DataBase

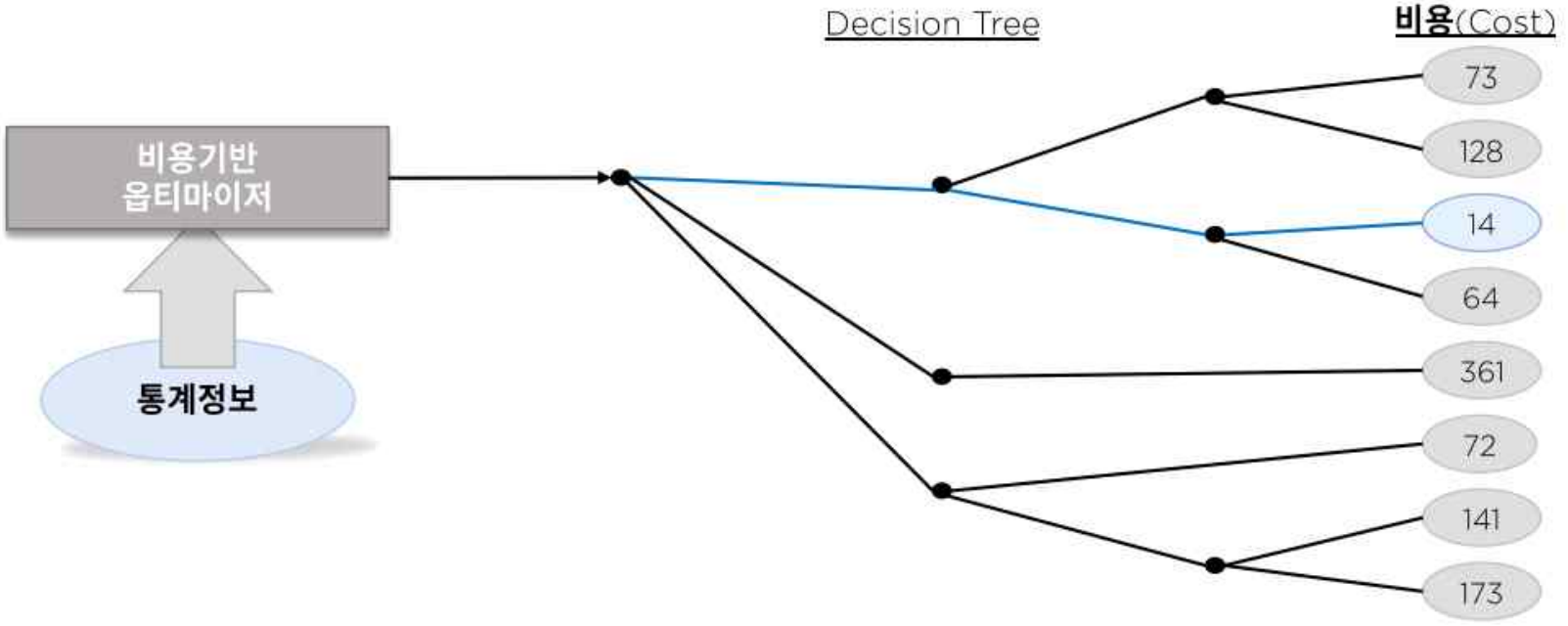
1. 옵티마이저와 실행 계획

➢ 옵티마이저란

① 사용자가 질의한 SQL문에 대한 최적의 실행방법을 결정하는 역할을 수행한다. 이러한 최적의 실행방법을 실행 계획이라고 한다.
② 사용자의 요구사항을 만족하는 결과를 추출할 수 있는 다양한 실행 방법이 존재함
③ 다양한 실행 방법들 중에서 최적의 실행방법을 결정하는 것이 옵티마이저의 역할이다.

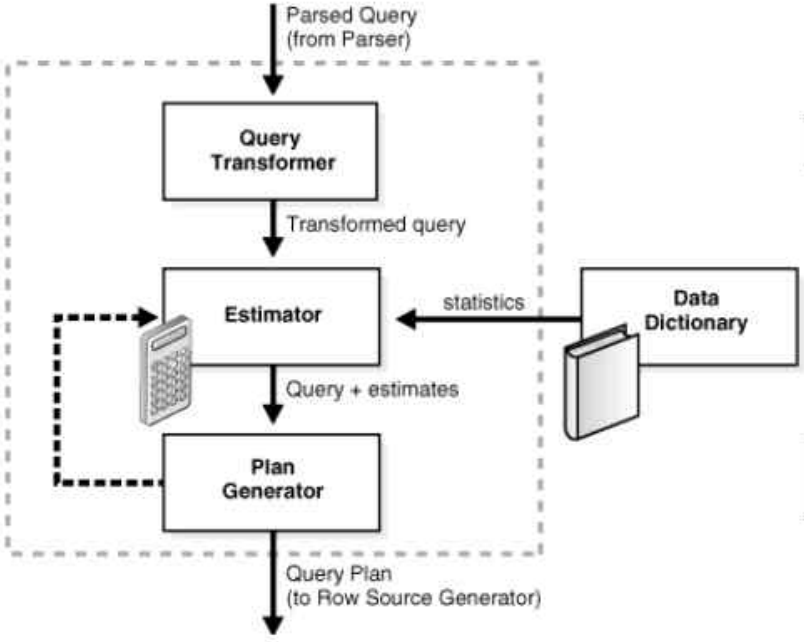
➢ 비용기반 옵티마이저

① SQL문을 처리하는데 비용이 가장 적게 드는 실행계획을 선택하는 방식이다. 비용이란 SQL문을 처리하는데 예상되는 시간 또는 자원 의미한다.
② 테이블, 인덱스 등의 통계 정보와 시스템 통계정보를 이용하여 최적의 실행계획을 도출한다.
③ 인덱스를 사용하는 비용이 전체 테이블 스캔 비용보다 크다고 판단되면 테이블 풀 스캔을 유도 하게 된다.



➢ 옵티마이저의 구성 요소

구성 요소	설명
질의 변환기 (Query Transformer)	• 사용자가 작성한 SQL문을 처리하기에 보다 용이한 형태로 변환
비용 예측기 (Estimator)	• 대안 계획 생성시에 의해서 생성된 대안 계획의 비용을 예측하는 모듈 • 대안 계획의 정확한 비용을 측정하기 위해서 연산의 중간 집합의 크기 및 결과 집합의 크기, 분포도 등의 예측을 함, 보다 나은 예측을 위해서 정확한 통계 정보가 필요함
대안계획생성기 (Plan Generator)	• 동일한 결과를 생성하는 다양한 대안 계획을 생성하는 모듈 • 대안 계획은 연산의 적용 순서, 연산방법변경, 조인 순서 변경 등을 통해서 생성 • 동일한 결과를 생성하는 가능한 모든 대안 계획을 생성해야 보다 나은 최적화를 수행할 수 있음



SQL은 기본적으로 구조적이고 집합적이고 선언적인 질의 언어다.

결과 집합을 만드는 과정은 절차적이며, 각각의 프로시저를 만드는 역할을 옵티마이저가 담당한다.

➢ 네비게이션과 옵티마이저

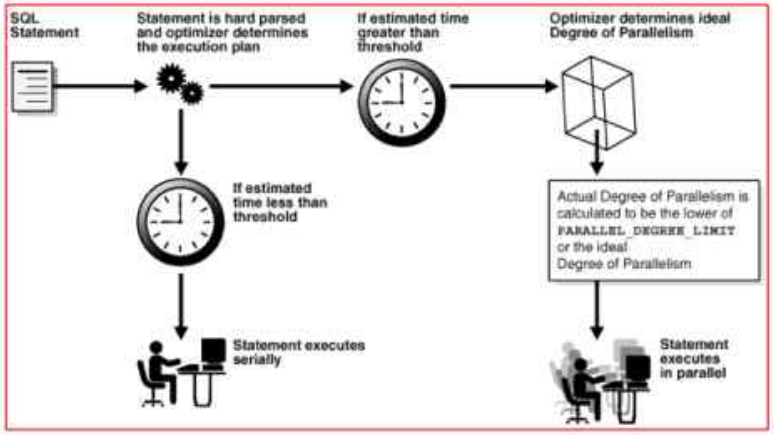
❖ 네비게이션

❖ 경로 탐색 과정에서 내비게이션이 착안하는 정보

- GPS 위치정보, 지도, 주소 정보, 도로정보
- 통행 요금, 구간별 평균/제한 속도, 실시간 교통 정보
- 공사 구간이나 시위, 도보 행진, 기타 행사로 인한 임시 교통 통제 구간 정보

❖ 수많은 정보를 바탕으로 여러 개의 경로를 알아낸 후 최적의 경로로 안내해주는 역할을 함

❖ 옵티마이저



❖ SQL 실행 계획 수립 시 옵티마이저가 착안하는 정보

- 테이블, 컬럼, 인덱스 구조에 관한 기본 정보
- 오브젝트 통계(테이블 통계, 인덱스 통계, 히스토그램 통계
- 시스템 통계 : CPU속도, Single Block I/O 속도, Multi block I/O 속도
- 옵티마이저 관련 파라미터

❖ 하나의 쿼리 수행 시 후보 군이 될만한 무수히 많은 실행계획을 도출, 짧은 순간 각각의 효율성을 판단해야 하는 무거운 과정

➢ 실행계획 예시

❖ TB_EMP 테이블의 인덱스 : PK_TB_EMP(EMP_NO)
❖ TB_DEPT 테이블의 인덱스 : PK_TB_DEPT(DEPT_CD)
❖ TB_EMP_CERTI 테이블의 인덱스 : PK_TB_EMP_CERTI(EMP_NO,CERTI_SN)

```
SELECT A.EMP_NO
     , A.EMP_NM
     , B.DEPT_CD
     , C.CERTI_CD
  FROM TB_EMP A
     , TB_DEPT B
     , TB_EMP_CERTI C
 WHERE B.DEPT_CD = '100004'
   AND A.DEPT_CD = A.DEPT_CD
   AND A.EMP_NO = C.EMP_NO;
```

- 위에서 아래로, 바깥쪽에서 안쪽으로 읽는다.
- 실행계획에는 사용 객체, 조인 방법, 조인 순서, 액세스패턴등의 정보가 출력된다.

```
************************[Explain Plan Time: 2020/06/30 17:52:16]************************
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=6 Card=123 Bytes=6K)
   1    0   HASH JOIN (Cost=6 Card=123 Bytes=6K)
   2    1     NESTED LOOPS (Cost=3 Card=46 Bytes=1K)
   3    2       INDEX (UNIQUE SCAN) OF 'PK_TB_DEPT' (INDEX (UNIQUE)) (Cost=0 Card=1 Bytes=7)
   4    2       TABLE ACCESS (FULL) OF 'TB_EMP' (TABLE) (Cost=3 Card=46 Bytes=1K)
   5    1     TABLE ACCESS (FULL) OF 'TB_EMP_CERTI' (TABLE) (Cost=3 Card=123 Bytes=2K)
----------------------------------------------------------

Predicate information (identified by operation id):
----------------------------------------------------------
   1 - access("A"."EMP_NO"="C"."EMP_NO")
   3 - access("B"."DEPT_CD"='000004')
----------------------------------------------------------
```

➢ 옵티마이저의 선택 - INDEX RANGE SCAN

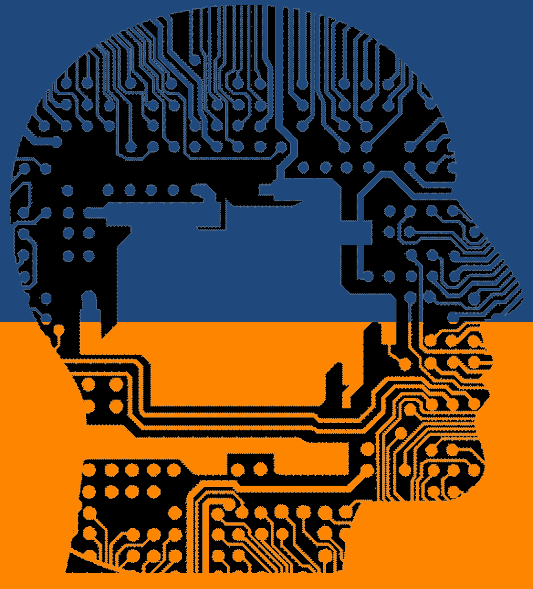
```
select *
  from t
 where deptno = 10
   and no = 1;

Execution Plan
----------------------------------------------------------

Plan hash value: 2369825647

-------------------------------------------------------------------------------------------
| Id  | Operation                           | Name  | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |       |     5 |   205 |     2   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID BATCHED| T     |     5 |   205 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN                  | T_X01 |     5 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("DEPTNO"=10 AND "NO"=1)
```

옵티마이저의 선택

❖ 비용(Cost)은 쿼리를 수행하는 동안 발생 될 것으로 예상하는 I/O 횟수 또는 예상 소요시간을 표현한 값이다.

❖ 비용(Cost)은 언제까지나 예상 치일 분이다. 실행 경로를 선택하기 위해 옵티마이저가 통계 정보를 활용해서 계산해낸 값이다.

한국IT진흥부설
정보보호교육학원 아이섹



JAVA 웹 개발자 양성과정

DataBase

2. 인덱스 기본

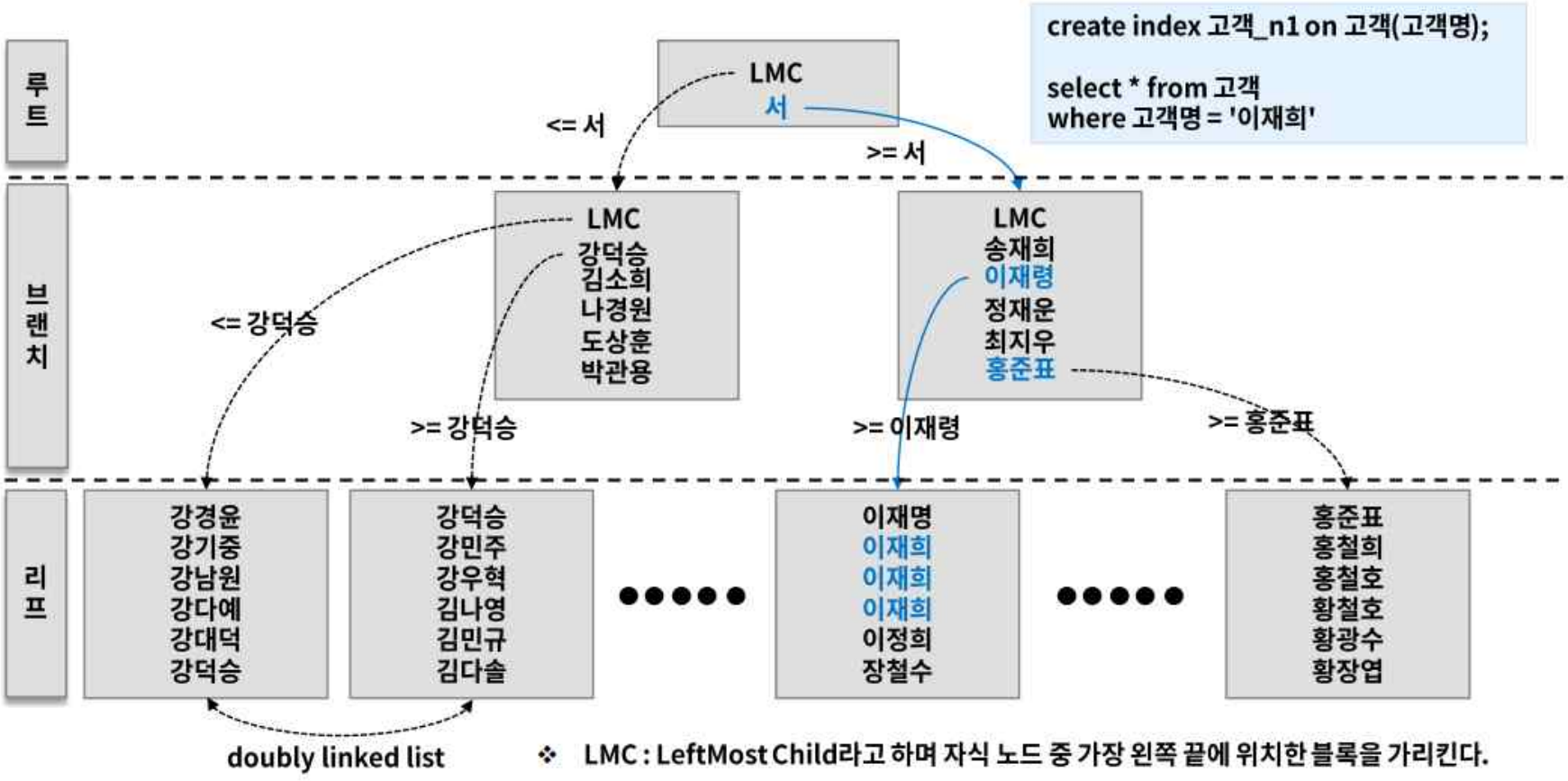
➢ 인덱스 란?

① 인덱스는 원하는 데이터를 쉽게 찾을 수 있도록 돕는 책의 찾아보기와 유사한 개념이다.
② 검색조건에 부합하는 데이터를 효과적으로(빠르게) 검색할 수 있도록 돕는다.
③ 한 테이블은 0개~N개의 인덱스를 가질 수 있다.
④ 한 테이블에 과도하게 많은 인덱스가 존재하면 INSERT, UPDATE, DELETE와 같은 DML작업 시 부하가 발생한다.

➢ 인덱스 란?

① DBMS에서 널리 사용되는 가장 일반적인 인덱스이다.
② 루트 블록, 브랜치 블록, 리프 블록으로 구성된다.
③ 가장 상위에 존재하는 블록이 루트 블록이고 브랜치 블록은 분기를 목적으로 하는 블록이다.
④ 리프 블록은 트리의 가장 아래 단계에 존재하는 블록이다.
⑤ 리프 블록은 인덱스를 구성하는 칼럼의 데이터와 해당 데이터를 가지고 있는 행의 위치를 가리키는 레코드 식별자 인 ROWID로 구성되어 있다.

➢ B*Tree 구조


create index 고객_n1 on 고객(고객명);
select * from 고객
where 고객명 = '이재희'
루트
브랜치
리프
LMC
서
<= 서
>= 서
LMC
강덕승
김소희
나경원
도상훈
박관용
<= 강덕승
>= 강덕승
LMC
송재희
이재령
정재운
최지우
홍준표
>= 이재령
>= 홍준표
강경윤
강기중
강남원
강다예
강대덕
강덕승
강덕승
강민주
강우혁
김나영
김민규
김다솔
이재명
이재희
이재희
이재희
이정희
장철수
홍준표
홍철희
홍철호
황철호
황광수
황장엽
doubly linked list
❖ LMC : LeftMost Child라고 하며 자식 노드 중 가장 왼쪽 끝에 위치한 블록을 가리킨다.

➢ 인덱스 구조 상세

① 루프와 브랜치 블록에 있는 각 레코드는 하위 블록에 대한 주소 값을 갖는다. 키 값은 하위 블록에 저장된 키 값의 범위를 나타낸다.
② LMC가 가리키는 주소로 찾아간 블록에는 키 값을 가진 첫번째 레코드보다 작거나 같은 레코드가 저장돼 있다.
③ 리프 블록에 저장된 각 레코드는 키 값 순으로 정렬돼 있을 뿐만 아니라 테이블 레코드를 가리키는 주소값 즉 Rowid를 갖는다.
④ 인덱스 키 값이 같으면 Rowid순으로 정렬된다.
⑤ 인덱스를 스캔하는 이유는 검색조건을 만족하는 소량의 데이터를 빨리 찾고 거기서 Rowid를 얻기 위해서이다

➢ ROWID의 구성

항목	구성
Rowid	데이터 블록 주소 + 로우 번호
데이터 블록 주소	데이터 파일 번호 + 블록 번호
블록 번호	데이터 파일 내에서 부여한 상대적 순번
로우번호	블록 내 순번

➢ 인덱스 스캔 효율화

❖ 시력이 1.0~1.5인 "이정민" 학생을 찾는 경우

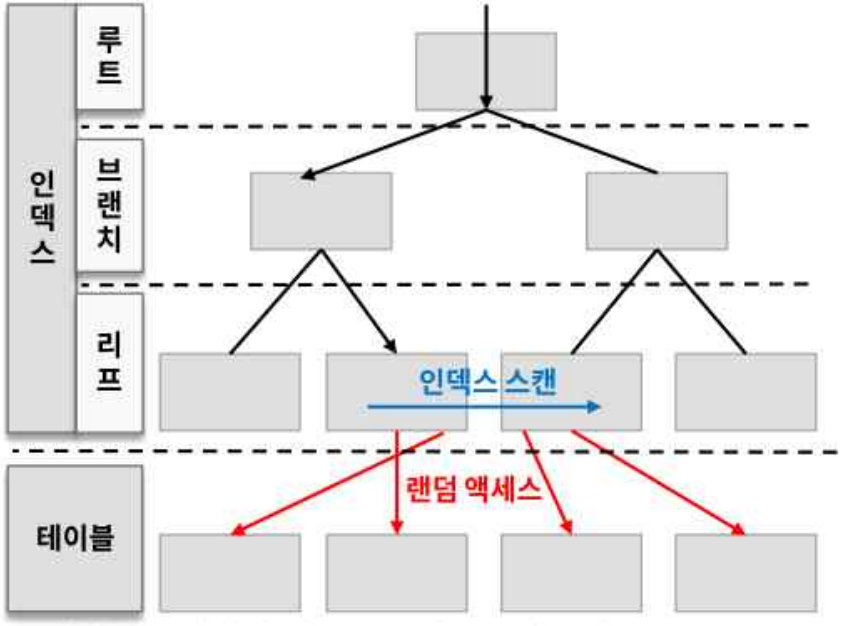
시력	이름	학년-반-번호
0.5	김지성	3학년 2반 13번
...	...	...
1.0	이정민	5학년 1반 16번
1.5	이경오	4학년 3반 37번
1.5	남경희	6학년 4반 19번
1.5	이정민	1학년 5반 15번
1.5	...	...
2.0	이정민	2학년 6반

✓ "시력+이름+학년-반-번호" 로 이루어진 인덱스는 스캔 비효율이 존재함

이름	시력	학년-반-번호
이경오	1.5	4학년 3반 37번
김지성	0.5	3학년 2반 13번
...	...	...
남경희	1.5	6학년 4반 19번
...	...	...
이정민	1.0	5학년 1반 16번
이정민	1.5	1학년 5반 15번
이정민	2.0	2학년 6반 24번

✓ "이름+시력+학년-반-번호"로 이루어진 인덱스는 스캔 효율이 좋음

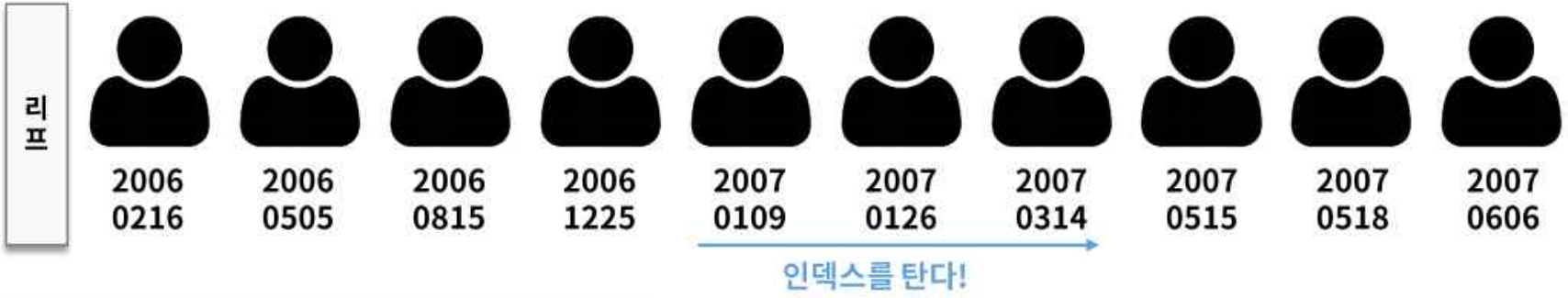
➢ 랜덤 액세스 최소화



- 인덱스 스캔 후 추가 정보를 가져오기 위해 Table Random Access를 수행한다.
- 해당 작업은 DBMS 성능 부하의 주 요인이 되며 SQL튜닝은 곧 Random I/O와의 전쟁이라 할 수 있다.

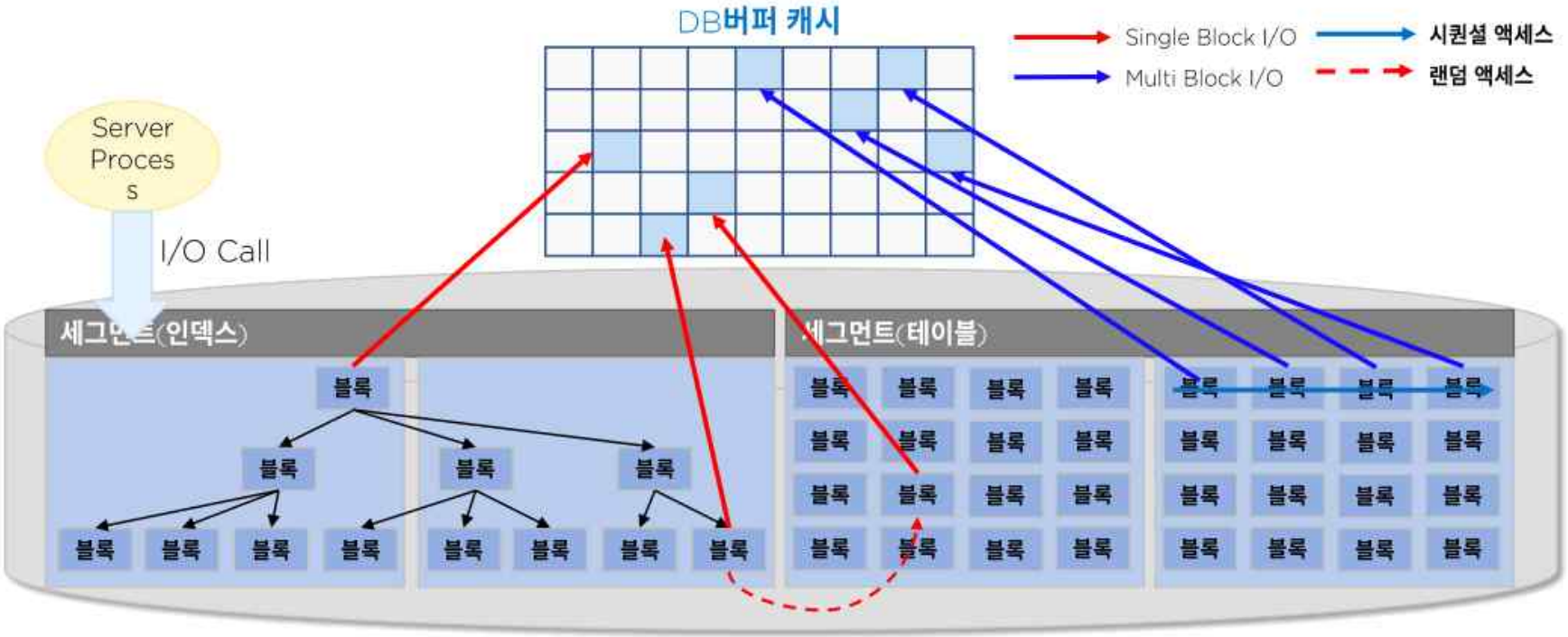
➢ 인덱스를 탄다 VS 인덱스를 안탄다

where 생년월일 between '20070101' and '20070131'



where substr(생년월일, 5, 2) = '05'

➢ Single Block I/O vs Multi Block I/O



- Single Block I/O**는 인덱스 루트 블록을 읽을 때, 인덱스 루트 블록에서 얻은 주소로 브랜치 블록을 읽을 때, 인덱스 브랜치 블록에서 읽은 주소로 리프 블록을 읽을 때 , 인덱스 리프블록에서 읽은 주소로 테이블 블록을 읽을 때 발생**
- Multi Block I/O**는 캐시에서 찾지 못한 특정 블록을 읽으려고** I/O Call**을 할 때 디스크 상에 그 블록과 인접한 블록들을 한꺼번에 읽어 캐시에 미리 적재 하는 것이다.** (DB_FILE_MULTIBLOCK_READ_COUNT)

➢ 풀 테이블 스캔과 인덱스 스캔

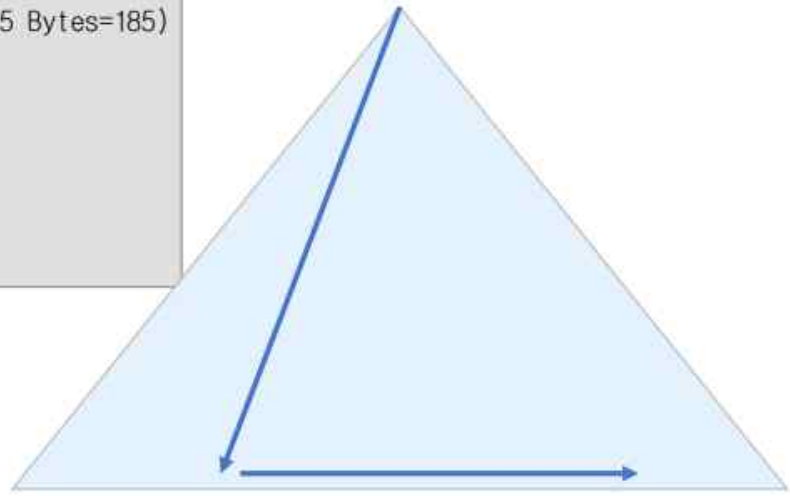
스캔 유형	설명
풀 테이블 스캔	• 테이블에 존재하는 모든 데이터를 읽어가면서 조건에 맞으면 결과로 추출하고 조건에 맞지 않으면 버리는 방식 • HIGH WATER MARK는 테이블에 데이터가 쓰여졌던 블록 상의 최상위 위치로써 테이블 풀 스캔 시는 HWM까지의 블록에 있는 • 모든 데이터를 읽어야 하기 때문에 시간이 오래 걸릴 수 있다. • 풀 테이블 스캔으로 읽은 블록은 재 사용성이 낮다고 보고 메모리 버퍼 캐시에서 금방 제거될 수 있도록 관리한다. • 옵티마이저가 풀 테이블 스캔을 선택하는 경우 - SQL문에 조건이 존재하지 않는 경우 - SQL문의 조건을 기준으로 사용 가능한 인덱스가 없는 경우 - 옵티마이저의 판단으로 풀 테이블 스캔이 유리하다고 판단하는 경우 - 전체 테이블 스캔을 하도록 강제로 힌트를 지정한 경우
인덱스 스캔	• 인덱스 스캔은 인덱스를 구성하는 칼럼의 값을 기반으로 데이터를 추출하는 액세스 기법 • 인덱스 리프 블록은 인덱스를 구성하는 칼럼과 ROWID로 구성 • 인덱스의 리프 블록을 읽으면 인덱스 구성 칼럼의 값과 ROWID를 알 수 있음 • 즉 인덱스를 읽어서 대상 ROWID를 찾으면 해당ROWID로 다시 테이블을 찾아 가야함 • 하지만 SQL문에서 필요로 하는 칼럼이 모두 인덱스 구성칼럼이라면 테이블을 찾아갈 필요 없음 • 일반적으로 인덱스 스캔을 통해 데이터를 추출하면 해당 결과는 인덱스의 칼럼의 순서로 정렬된 상태로 반환됨

➢ 인덱스 범위 스캔

① 인덱스를 이용하여 한건 이상의 데이터를 추출하는 방식
② 인덱스 스캔으로 특정 범위를 스캔하면서 대상 레코드를 하나하나 리턴하는 방식임

```
CREATE INDEX EMP_IDX01 ON EMP(DEPTNO);

SELECT *
FROM EMP
WHERE DEPTNO = 20
```

```
************************[Explain Plan Time: 2018/09/17 14:47:58]************************
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=5 Bytes=185)
   1    0   TABLE ACCESS (BY INDEX ROWID BATCHED) OF 'EMP' (TABLE) (Cost=2 Card=5 Bytes=185)
   2    1     INDEX (RANGE SCAN) OF 'EMP_IDX01' (INDEX) (Cost=1 Card=5)
----------------------------------------------------------

Predicate information (identified by operation id):
----------------------------------------------------------
   2 - access("DEPTNO"=20)
----------------------------------------------------------
```



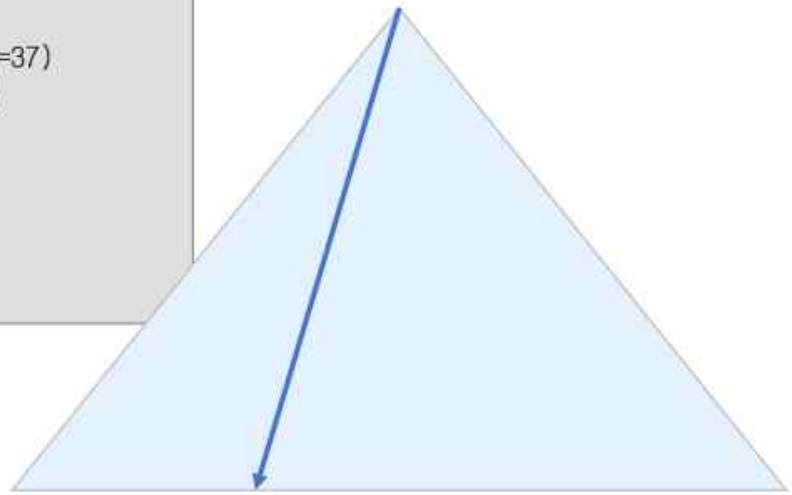
Index Range Scan

➢ 인덱스 유일 스캔

① 인덱스를 사용하여 단 하나의 데이터를 추출하는 방식
② 유일인덱스는 중복 레코드를 허용하지 않음
③ 유일인덱스는 반드시 '='조건으로 조회 해야 됨(그렇게 할 수 밖에 없음)

```
CREATE UNIQUE INDEX EMP_IDX03 ON EMP(EMPNO);

SELECT *
FROM EMP
WHERE EMPNO = 7788;
```

```
************************[Explain Plan Time: 2018/09/17 15:00:27]************************
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=1 Card=1 Bytes=37)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE) (Cost=1 Card=1 Bytes=37)
   2    1     INDEX (UNIQUE SCAN) OF 'EMP_IDX03' (INDEX (UNIQUE)) (Cost=0 Card=1)
----------------------------------------------------------

Predicate information (identified by operation id):
----------------------------------------------------------
   2 - access("EMPNO"=7788)
----------------------------------------------------------
```



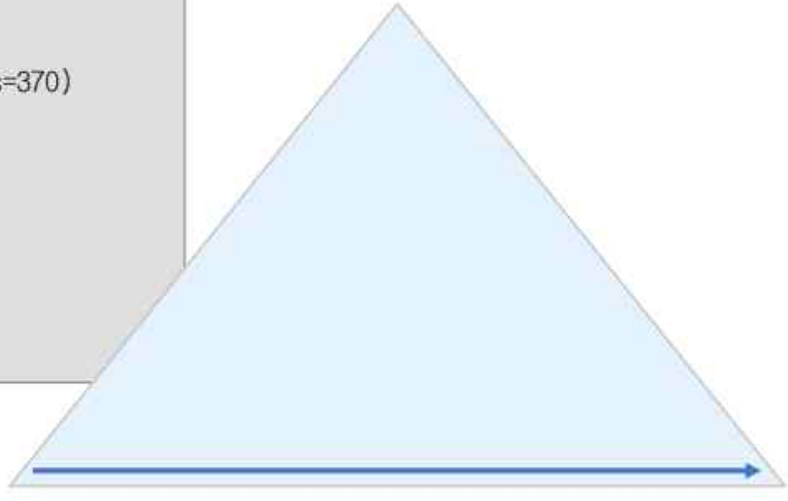
Index Unique Scan

➢ 인덱스 전체 스캔

① 인덱스를 처음부터 끝까지 전체를 읽으면서 조건에 맞는 데이터를 추출함
② 데이터를 추출 시 리프 블록에 있는 **ROWID**로 테이블의 레코드를 찾아가서 조건에 부합하는지 판단하고
③ 조건에 부합되면 해당 행을 리턴 함

```
CREATE INDEX EMP_IDX02 ON EMP(ENAME, SAL);

SELECT *
FROM EMP
WHERE SAL > 2000
ORDER BY ENAME
```

```
************************[Explain Plan Time: 2018/09/17 14:47:41]************************
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=10 Bytes=370)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE) (Cost=2 Card=10 Bytes=370)
   2    1     INDEX (FULL SCAN) OF 'EMP_IDX02' (INDEX) (Cost=1 Card=10)
----------------------------------------------------------

Predicate information (identified by operation id):
----------------------------------------------------------
   2 - access("SAL">2000)
   2 - filter("SAL">2000)
----------------------------------------------------------
```



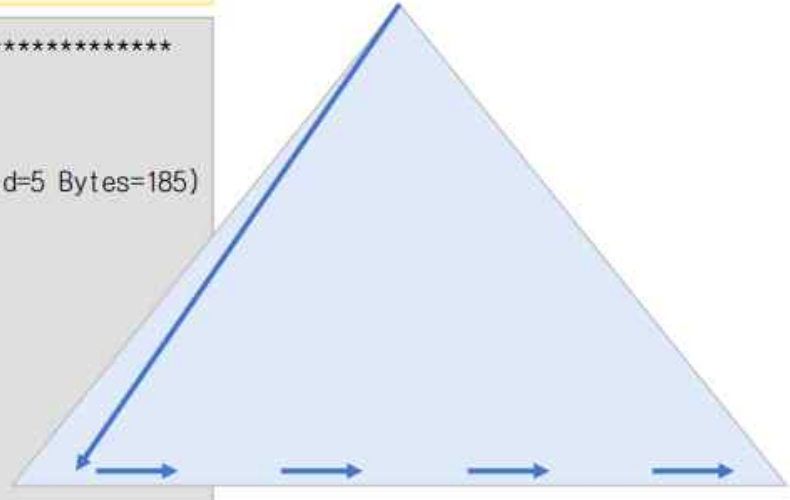
Index Full Scan

➢ 인덱스 스킵 스캔

① 인덱스 선두 컬럼이 조건절에 없어도 인덱스를 활용하는 스캔 방식이다.
② 조건절에 빠진 인덱스 선두 컬럼(성별)의 Distinct Value의 개수가 적고, 후행 컬럼(연봉)의 Distinct Value의 개수가 많을 때 유용
③ Index Skip Scan은 루트 또는 브랜치에서 읽은 컬럼 값 정보를 이용해 조건절에 부합하는 레코드를 포함할 가능성이 있는 리프 블록만 액세스 한다.

```
CREATE INDEX EMP_IDX02 ON EMP(ENAME, SAL);

SELECT /*+ INDEX_SS(EMP EMP_IDX02) */
*
FROM EMP
WHERE SAL BETWEEN 2000 AND 4000
```

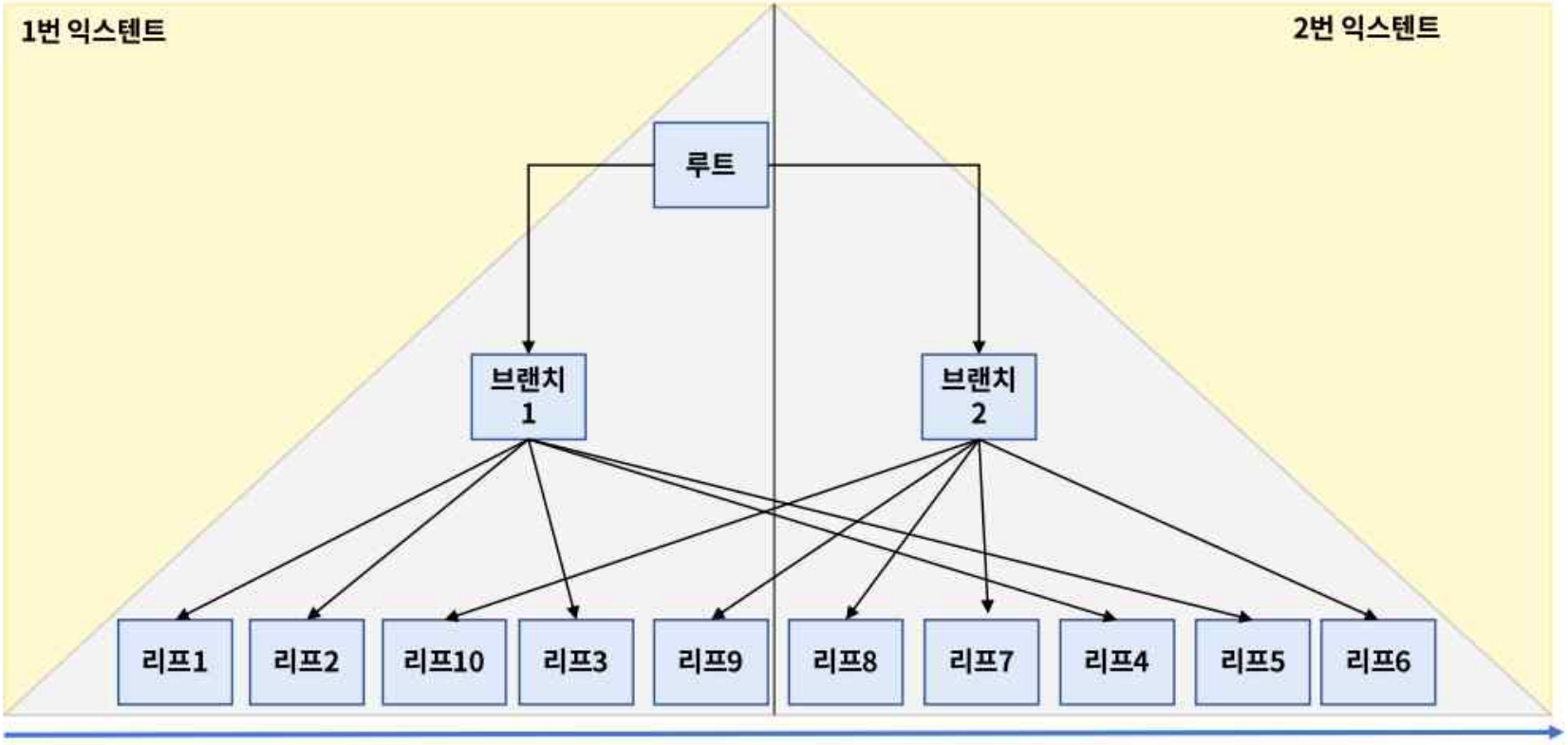
```
************************[Explain Plan Time: 2018/09/17 16:46:28]************************
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=5 Bytes=185)
   1    0   TABLE ACCESS (BY INDEX ROWID BATCHED) OF 'EMP' (TABLE) (Cost=2 Card=5 Bytes=185)
   2    1     INDEX (SKIP SCAN) OF 'EMP_IDX02' (INDEX) (Cost=1 Card=5)
----------------------------------------------------------

Predicate information (identified by operation id):
----------------------------------------------------------
   2 - access("SAL">=2000 AND "SAL"<=4000)
   2 - filter("SAL">=2000 AND "SAL"<=4000)
----------------------------------------------------------
```



Index Skip Scan

➢ 인덱스 고속 전체 스캔

① Index Fast Full Scan은 물리적으로 디스크에 저장된 순서대로 인덱스 리프 블록들을 Multi Block I/O 방식으로 읽어 들인다. 또한 병렬 인덱스 스캔도 가능하다.



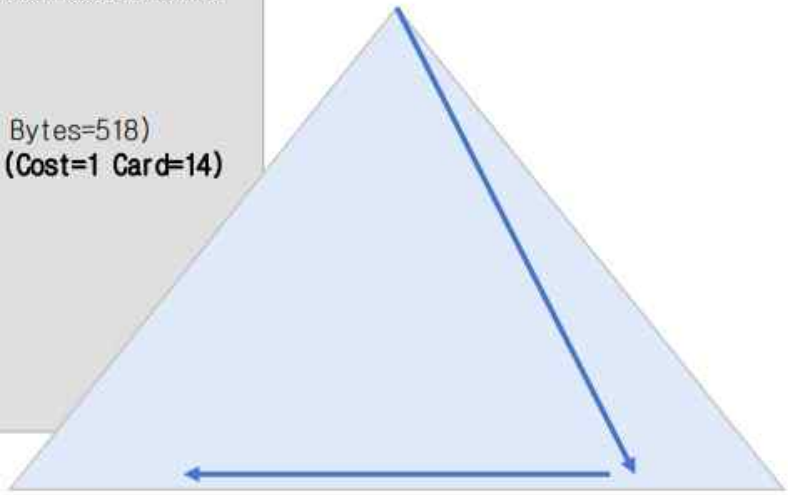
인덱스 리프 노드가 갖는 연결 구조를 무시한 채 데이터를 읽기 때문에 인덱스 정렬 순서의 보장을 하지 못한다.

➢ 인덱스 역순 범위 스캔

① 인덱스 리프 블록은 Doubly Linked List방식으로 저장되어 있음
② 즉 이 성질을 이용하여 인덱스를 역순으로(거꾸로) 읽을 수 있음
③ 인덱스를 뒤에서부터 앞쪽으로 스캔하기 때문에 내림차순으로 정렬된 결과 집합을 얻을 수 있다. (스캔 순서를 제외 하고는 Range Scan과 동일함)

```
SELECT
    *
FROM EMP
WHERE EMPNO > 0
ORDER BY EMPNO DESC
```

```
************************[Explain Plan Time: 2018/09/17 17:55:05]************************
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=14 Bytes=518)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (TABLE) (Cost=2 Card=14 Bytes=518)
   2    1     INDEX (RANGE SCAN DESCENDING) OF 'EMP_IDX03' (INDEX (UNIQUE)) (Cost=1 Card=14)
----------------------------------------------------------

Predicate Information (identified by operation id):
----------------------------------------------------------
   2 - access("EMPNO">0)
----------------------------------------------------------
```



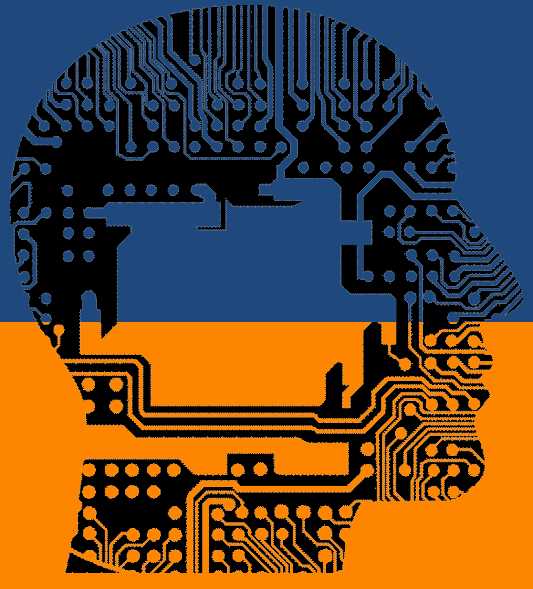
Index Range Scan Descending

➢ 테이블 스캔 VS 인덱스 스캔

풀 테이블 스캔	인덱스 스캔
항상 이용 가능	인덱스가 존재해야만 이용가능
한번에 여러 개의 BLOCK을 읽음	한번에 한 개에 블록만을읽음
많은 데이터를 조회 시 성능 상 유리	극히 일부분의 데이터를 조회 시 유리
Table Random Access부하 없음	Table Random Access에 의한 부하가 발생됨
읽었던 블록을 반복해서 읽는 경우 없음	읽었던 블록을 반복해서 읽는 비효율 발생(논리적인 블록 I/O의 개수도 많아짐)

❖ 인덱스 스캔은 생각 했던 것보다 훨씬 부하가 큰 작업이다.
❖ 즉 반드시 인덱스 스캔이 테이블 풀 스캔보다 성능이 좋다고 생각하는 것은 금물이다.
❖ 데이터 건수가 많은 테이블에서 소량의 데이터를 스캔 할 때 사용해야 한다.
❖ 즉 인덱스 스캔 효율을 높여서 절대적인 논리적 I/O를 줄이는 노력을 해야한다.

한국IT진흥부설
정보보호교육학원 아이섹



JAVA 웹 개발자 양성과정

DataBase

3. 조인 수행 원리

➢ 조인이란?

① 조인이란 두개 이상의테이블을 하나의 집합으로 만드는 연산이다.
② SQL문의 FROM절에 두개 이상의 테이블 혹은 집합이 존재할 경우 조인이 수행된다.
③ 조인은 3개 이상의 테이블을 조인한다고 하더라도 특정 시점에 2개의 테이블 단위로 조인이 된다.
④ A, B, C 집합을 조인한다면 A, B조인 후 해당 결과 집합을 C와 조인 하는 방식이다.
⑤ 각각의 조인 단계에서는 서로 다른 조인 기법이 사용될 수 있다.
⑥ 즉 A, B조인 시에는 NL조인을 수행하고 A, B조인의 결과와 C를 조인 시에는 해시 조인이 수행될 수 있다.

➢ NL 조인

```
SELECT *
FROM EMP A
       , DEPT B
WHERE A.EMPNO = 7369
  AND A.DEPTNO = B.DEPTNO;
```

NL 조인은 작은 집합이 Driving되어야 하고 Inner 테이블의 인덱스 스캔이 매우 중요하다.

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	Reads
0	SELECT STATEMENT		1		1	00:00:00.01	4	1
1	NESTED LOOPS		1	1	1	00:00:00.01	4	1
2	TABLE ACCESS BY INDEX ROWID	EMP	1	1	1	00:00:00.01	2	0
* 3	INDEX UNIQUE SCAN	PK_EMP	1	1	1	00:00:00.01	1	0
4	TABLE ACCESS BY INDEX ROWID	DEPT	1	4	1	00:00:00.01	2	1
* 5	INDEX UNIQUE SCAN	PK_DEPT	1	1	1	00:00:00.01	1	1

- ❖ RANDOM 액세스 위주(인덱스구성이 완벽 해도 대량 데이터 조인 시 불리)
- ❖ 한 레코드 씩 순차 진행(부분 범위 처리를 유도해야 효율적 수행)
- ❖ DRIVING 테이블 처리 범위에 의해 전체 성능이 결정됨
- ❖ 인덱스 유무, 인덱스 구성에 크게 영향 받음
- ❖ 소량의 데이터를 처리하거나 부분범위처리가 가능한 OLTP 환경에 적합

➢ 소트 머지 조인

```
SELECT /*+ FULL(A) FULL(B) USE_MERGE(A B) */ *
 FROM EMP A
        , DEPT B
WHERE A.EMPNO = 7369
  AND A.DEPTNO = B.DEPTNO;
```

정렬 작업을 생략할 수 있는 인덱스가 존재하는 경우 사용!

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
0	SELECT STATEMENT		1		1	00:00:00.01	14			
1	MERGE JOIN		1	1	1	00:00:00.01	14			
2	SORT JOIN		1	1	1	00:00:00.01	7	2048	2048	2048 (0)
* 3	TABLE ACCESS FULL	EMP	1	1	1	00:00:00.01	7			
* 4	SORT JOIN		1	4	1	00:00:00.01	7	2048	2048	2048 (0)
5	TABLE ACCESS FULL	DEPT	1	4	4	00:00:00.01	7			

- ❖ 실시간 인덱스 생성 : 양쪽 집합을 정렬한 다음에는 NL 조인과 같은 오퍼레이션
- ❖ 인덱스 유무에 영향을 받지 않음 : 미리 정렬된 인덱스가 있으면 좀 더 빠르게 수행할 수는 있음
- ❖ 양쪽 집합을 개별적으로 읽고 나서 조인 : 조인 컬럼에 인덱스가 없는 상황에서 두 테이블을 독립적으로 읽어 조인 대상 집합을 줄일 수 있을 때 아주 유리
- ❖ 스캔(Scan) 위주의 액세스 방식 : 양쪽 소스 집합에서 정렬 대상 레코드를 찾는 작업은 인덱스를 이용 Random 액세스 방식으로 처리될 수 있음

➢ 해시 조인

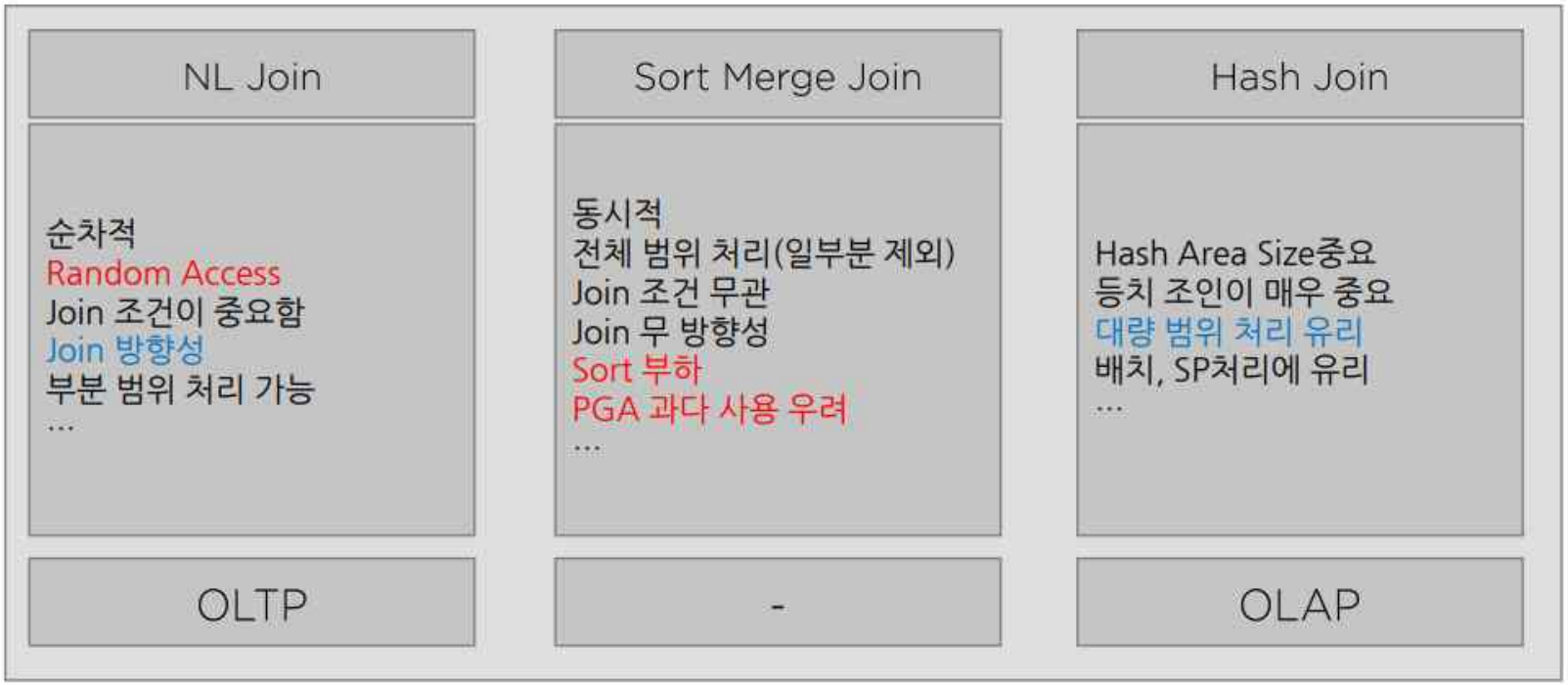
```
SELECT /*+ USE_HASH(A B) */ *
FROM EMP A
       , DEPT B
WHERE A.EMPNO = 7369
  AND A.DEPTNO = B.DEPTNO;
```

작은 집합을 build Input으로 하고 큰 집합을 probe input으로 하는 것이 중요

Id	Operation	Name	Starts	E-Rows	A-Rows	A-Time	Buffers	OMem	1Mem	Used-Mem
0	SELECT STATEMENT		1		1	00:00:00.01	10			
* 1	HASH JOIN		1	1	1	00:00:00.01	10	832K	832K	353K (0)
2	TABLE ACCESS BY INDEX ROWID	EMP	1	1	1	00:00:00.01	2			
* 3	INDEX UNIQUE SCAN	PK_EMP	1	1	1	00:00:00.01	1			
4	TABLE ACCESS FULL	DEPT	1	4	4	00:00:00.01	8			

- ❖ 대량의 데이터 처리가 필요하고 쿼리 수행 시간이 오래 걸리는 대용량 테이블을 조인할 때(배치 프로그램, DW, OLAP성 쿼리) 사용
- ❖ NL 조인처럼 Random 액세스 부하 없음
- ❖ 소트 머지 조인처럼 정렬 부하 없음
- ❖ 해시 테이블을 생성하는 비용에 따라서 Build Input이 (Hash Area에 담을 수 있을 정도로 충분히) 작을 때라야 효과적

➢ 조인 기법 비교



❖ 집합적인 사고를 바탕으로 적절한 Join Method 선정이 중요하다.
❖ 시스템의 특성을 참고하여 적절한 Join Method 선정이 중요하다.

➢ 조인 순서의 중요성

항목	설명	비고
First Table	• 두개의 Table을 조인 할 경우 먼저 처리되는 테이블을 의미한다. • WHERE절에 상수/바인드 변수 조건이 존재하는 것이 성능상 유리하다.	Outer Table Driving Table Build Input
Second Table	• 두개의 테이블을 조인 할 경우 뒤에 처리되는 테이블을 의미한다. • First Table로 부터 입력 값을 받아서 처리하게 된다. • 조인 조건의 여부 및 성질이 조인 조건이 성능에 영향을 미친다. • 조인 조건 및 상수/바인드 변수 조건에 인덱스 존재 여부가 매우 중요하다. (NL Join에 경우)	Inner Table Driven Table Probe Input
최적화된 Join Order	• First Table이 Second Table에 비해서 작은 집합 이어야 성능상 유리하다. (NL, Hash Join에 경우)	

감사합니다
THANK YOU