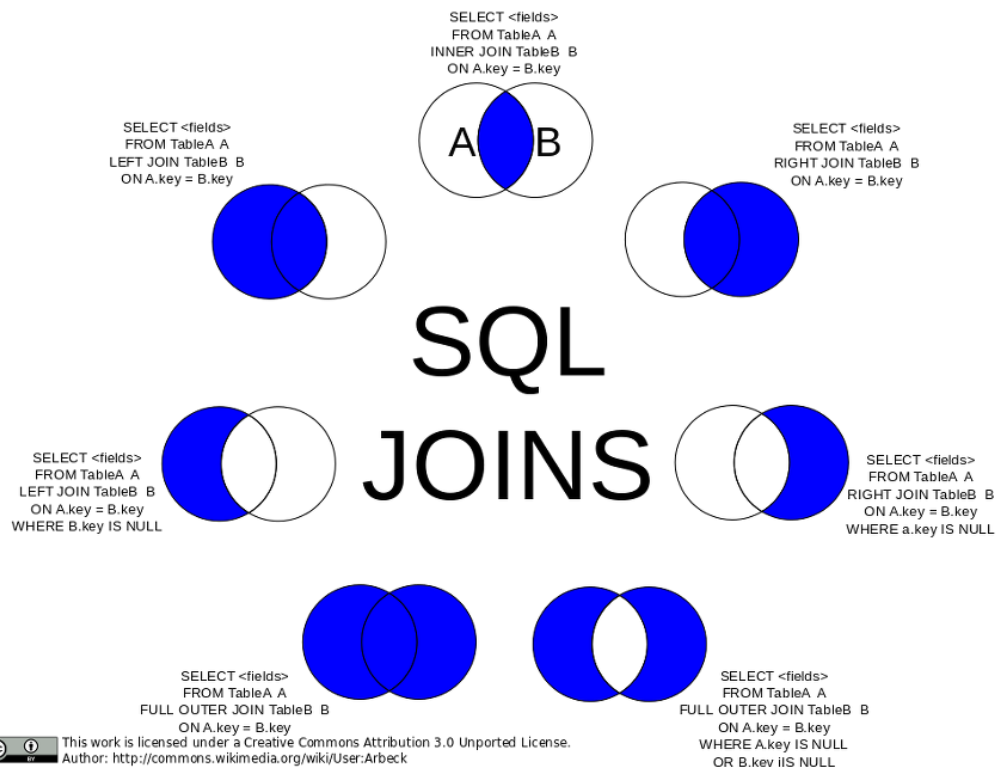


## 06. Join



# ● 1. Join 사례

---

## ▣ 사례1

윈도우 NT 환경에서 운영중인 DBMS가 비정상 종료 되어 N시간 동안 본사, 전체 지사 와 대리점 전산망이 마비가 되는 상황 발생

### 원인

- Temporary Tablespace를 구성하는 물리적 파일 손상(Media Failure), Temporary Tablespace는 정렬

연산 수행시 데이터량으로 Memory내에서 정렬을 수행할수 없는 경우 Server Process가 임시로

(Temporary) 사용하는 Disk내의 File 공간.

- DBMS 비정상 종료의 원인은 Temporary Tablespace를 구성하는 파일의 크기가 1G에서 8G로 확장

당시 NT의 파일 시스템은 최대 크기는 8G (NTFS : 16TB – 2TB)

- 비정상 종료시점(Media failure → Instance Failure)에 실행된 SQL의 실행계획

MERGE JOIN (CARTESIAN) (Cost=xxxxx Card=3600000000 (36억건) Bytes=xxxxxxx)

- 카티션곱(Cartesian Product)을 통한 Merge Join 발생

# ● 1. Join 사례

---

## ■ 사례2

대국민 민원서비스를 제공하는 웹서비스로 메인 조회시 10초 이상 소요, 전자민원 조회시 30초 이상 소요. 메인 화면에서는 공지사항,행사교육,보도자료,입찰정보등 4가지 주요 정보에 대해 각각 가장 최근의 5개 데이터 제공

## 원인

- 메인에서는 4종류 20개의 데이터를 보여 주고 로직과 디자인이 간단하고 가볍다. 오랜 시간 운영으로 누적된 데이터가 많은 상황에서 10초가 소요되는 메인 화면의 SQL들은 2~3개의 테이블을 조인하는 비교적 간단한 조인 구문

- 메인 화면의 4개 조인 SQL 전부가 Sort Merge Join 방식으로 실행계획이 수립 된다. 전체 데이터중에 가장 최근의 5개 데이터를 보여주는 업무 형태에 (Sort) Merge Join은 비효율적

SQL 조건절에서 인덱스 컬럼의 변형이 발생하여 인덱스를 사용할수 없게 되자  
옵티마이저는 Nested Loop Join 대신 부득이 하게 (Sort) Merge Join으로 실행계획 수립

- 4개의 SQL문의 실행계획을 (Sort) Merge Join 에서 Nested Loop 유도 ➡ 응답시간 1~3초
- 위의 사례에서
  - ① 전체 데이터중 소량의 데이터를 조인 처리시 Sort Merge Join 보다는 Nested Loop Join이 효율적 일수 있다.
  - ② Index 사용 여부에 따라 옵티마이저는 Join처리 알고리즘을 선택

# ● 1. Join 사례

---

## ▣ 사례3

VOIP 서비스를 하는 통신 회사의 빌링시스템으로 전화 1 Call이 발생 할때마다 일별, 월별 집계와 빌링 계산을 위해 N개의 레코드가 발생하며 이를 CDR(Call Detail Record)이라 한다.

빌링시스템의 업무를 데이터 처리 방식으로 나뉘보면 서비스 특성상 소량의 데이터를 실시간 처리 해야 하는 업무, 많은 CDR 데이터를 한꺼번에 처리해야 하는 업무로 구분.

트랜잭션의 특성에 따라 2가지 유형으로 구분.

OLTP(OnLine TransactionProcessing)는

소량의 데이터를 실시간에 처리하는 업무에 해당하는 Transaction.

DSS(Decision Support System)은

많은 데이터를 한꺼번에 처리하는 업무에 해당 하는 Transaction.

일별/월별 집계나 빌링의 요금정산 업무는 많은 데이터를 한번에 집계처리 하는 특성 인터넷 전화의 주 고객은 기업 고객으로 일별 사용한 통화 시간 및 요금을 집계한후 웹상으로 일별 사용내역을 조회할수 있는 서비스 제공. 일별 요금정산 처리는 새벽 2~3시에 완료 되었지만 인터넷 전화 고객의 증가에 따라 통화량이 증가함에 따라 DBMS서버가 설치된 H/W서버를 고사양으로 업그레이드 하고 DB를 Migration. 업그레이드후 일별 요금 정산 처리가 오전 9~10시에 완료되었고 이로 인해 전화요금을 익일 오전중에는 원활히 조회를 할수 없는 상황 발생 DBMS의 성능저하 문제로 기업 비즈니스 운영 문제 발생

# ● 1. Join 사례

## ■ 사례3

### 원인

- H/W 신규 도입 → Oracle DBMS 업그레이드 → 옵티마이저의 정책 변경 → 실행계획 변경
- 확인 결과 Migration은 정상적으로 수행 되었으며 DBMS 업그레이드(X)
- 기존 시스템은 개발 시스템으로 사용 .  
기존 시스템과 신규시스템에서 인스턴스 레벨의 SQL Trace → 비교 →  
외부 자문을 통해 신규 시스템의 Join SQL에 /\*+ USE\_NL(~) \*/ 힌트  
추가  
→ 모든 Join 구문에 USE\_NL 힌트 적용

- OLTP 트랜잭션 업무에서 USE\_NL 힌트를 사용하여 조인 처리를 Nested Loop Join 방식으로 유도해서 성능향상이 되는 것을 경험한후 신규 시스템 도입시 배치성(DSS)트랜잭션 업무인 일별/월별 집계에도 USE\_NL 힌트를 사용하여 Nested Loop Join 방식으로 조인 처리 유도
- 기존 시스템의 일별/월별 집계는 Hash Join방식으로 조인 처리
- 위의 사례에서 SQL 작성시 다음의 교훈을 얻을수 있다.
  - ① 대량의 데이터를 조인으로 처리 할때는 Nested Loop Join 보다는 Sort Merge Join 이나 Hash Join이 효율적 일수 있다
  - ② OLTP성 트랜잭션 유형에는 Nested Loop Join이 일반적으로 적절하고 배치(DSS)성 트랜잭션 유형에는 Sort Merge Join 이나 Hash Join이 일반적으로 적절하다.
  - ③ 힌트 일괄 적용(X)
  - ④ DBMS 버전 변경시 옵티마이저의 정책이나 내부 알고리즘 변경 가능 → 실행계획 변경
  - ⑤ DBMS 성능 저하는 기업 비즈니스 생산성 및 경쟁력 저하

## ● 2. 현실세계 소개팅 방법

---

조인은 남녀간의 미팅을 생각하면 쉽게 이해가 쉽다.

당신은 미팅 주선자이다.

남자 집합 A(100명), 여자 집합 B(100명) 있다고 가정 하자. 크게 2가지 방식을 생각 할수 있다.

**먼저 대쉬** 하는 그룹에서 미팅을 주도하는 방식과 **평등**하게 하는 방식이 있다.

### (1) 주도적(Driving)으로 대쉬 하는 방식 (순서 0)

남녀그룹 대표가 가위 바위보를 해서 이긴 쪽이 상대방 그룹에 주도(Driving)적으로 대쉬한다.

남자 집합 A가 미팅을 주도하는 그룹이 되었다고 가정 하자. 대쉬도 2가지 방식이 있다.

① A 그룹의 첫번째 사람이 B 그룹의 사람들을 전부 짝 보고 마음에 드는 사람을 찾는다.

두번째 사람이 B 그룹의 사람들을 전부 짝 보고 마음에 드는 사람을 찾는다.

N번째 사람까지 반복한다.

현실 세계와 다른점은 A그룹 사람이 B 그룹의 상대방을 중복적으로 선택할수 있다.

② A 그룹의 사람들 전부가 **중앙 테이블(Hash table)**에 자신의 전화번호를 적은 자신만의

고유한 소장품을 둔다. B 그룹의 첫번째 사람부터 순차적으로 중앙 테이블의 소장품중

## ● 2. 현실세계 소개팅 방법

---

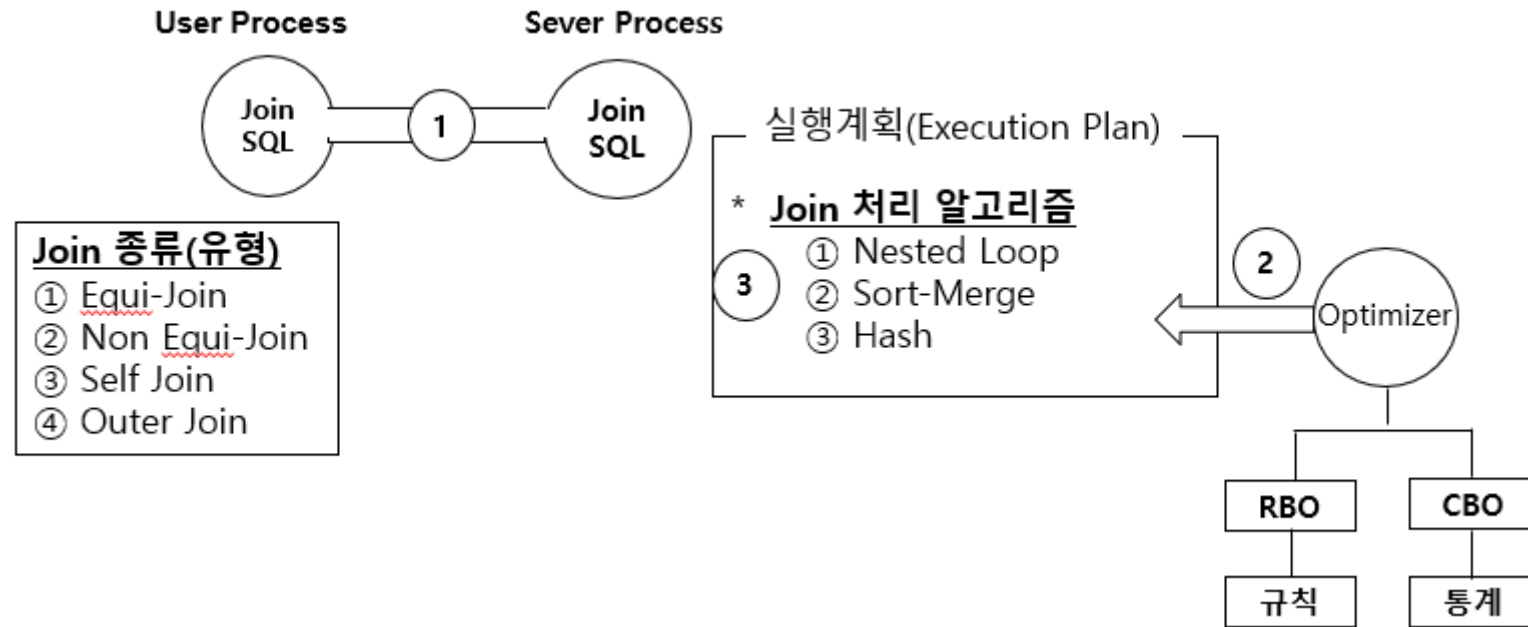
자신의 마음에 드는 소장품을 선택한다. 소장품에는 그의 연락처가 있으니 상대방을 찾을 수 있다. 현실 세계와 다른점은 A그룹 사람이 마음에 드는 B 그룹의 상대방을 중복적으로 선택할 수 있다.

### (2) 평등한 방식 (순서 X)

그룹 A와 그룹 B를 서로의 **공통 관심 사항을 기준으로** 순서대로(**정렬, sort**) 줄을 선다한다. 예를 들면 취미(**공통 속성=조인 조건**)를 기준으로 A, B 그룹 각각 줄을 선다 그룹 A, B는 각각 사람의 위치(사랑의 작대기)를 가리키는 포인터를 가진다.

그룹 A의 첫번째 사람이 그룹 B에서 같은 취미를 가진 사람들을 모두 찾은 후, A그룹의 포인터는 두번째 사람을 지정한다. 그룹 A에서 B 그룹으로 매칭되지 않는 경우 B 그룹의 포인터는 두 번째 취미를 가진 첫 번째 사람을 지정한다. 그룹 A의 두번째 사람이 그룹 B의 포인터가 지정하는 같은 취미를 가진 사람들을 찾는다. 현실 세계와 다른점은 A그룹 사람은 같은 취미를 가지는 B 그룹의 여러 상대방을 중복적으로 선택할 수 있다

### ● 3. Join 처리과정



① User process가 SQL(Join)을 Server process에게 전송.

\* Equi-Join , Outer Join 등의 조인 종류는 연산의 형태나 기능에 의한 분류

② 옵티마이저가 ㉠ Query Transformation(Query Rewrite) ㉡ Query Optimization

㉢ Generate Execution Plan 3단계를 거쳐 실행계획 생성.

\* 예를들면 서브쿼리(Sub Query)를 Query Transformation을 하여 조인으로 변경한다

③ Query Optimization 단계에서 Nested Loop Join ,Soft Merge Join, Hash Join 알고리즘 중 효율적인 알고리즘 선택.



## ● 4. Join 처리 알고리즘 용어

성능 최적화

### ■ Driving ,Driven

- Driving은 추진하는,운전하는(조종하는)의 사전적인 의미를 가지며  
조인 연산시 먼저(선행) 적극적으로 주도하여 움직이는 테이블
- Driven은 내몰린, 구동되는(조종당하는)의 사전적 의미를 가지며 조인 연산시 나중(후행)에  
소극적으로 움직이는 테이블

Join 처리 알고리즘	선행 테이블(Driving Table)	후행 테이블(Driven)
Nested Loop	Outer Table	Inner Table
Sort Merge	N/A	N/A
Hash	Build Table	Probe Table

\* 먼저 실행되는 선행 테이블의 선택에 따라 처리해야할 일량의 차이가 발생 하기 때문에  
선행 테이블의 결정은 성능에 주요한 영향을 주게 된다.

```
* SELECT TABLE_NAME,INDEX_NAME,COLUMN_POSITION,COLUMN_NAME FROM USER_IND_COLUMNS
WHERE TABLE_NAME IN ('CUSTOMER','EMP')
ORDER BY 1,2,3;
CHECK INDEX : CUSTOMER.ID , EMP.EMPNO
```

## ● 5. Join 처리 알고리즘 – Nested Loop

---

### ▣ Nested Loop (중첩 루프)

```
for( l=2 ; l <=9 ; l++ ) {
```

-- Outer Loop ,선행(Driving)

```
    for( J=1 : J <= 9 ; J++ ) {
```

-- Inner Loop, 후행(Driven)

```
        printf(" %d * %d = %2d Wn", l , J);
```

```
    }
```

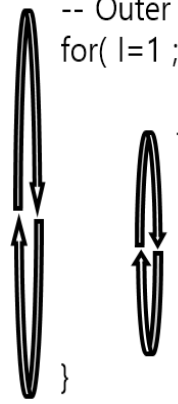
```
}
```

### ▣ Nested Loop Join

조인은 항상 2개의 테이블(데이터 결과 집합)사이에 수행된다. l, J를 테이블이라고 간주하면 첫 번째 테이블 l에서 첫 번째 레코드를 가지고 테이블 J에서 상호 연관되는 레코드를 찾고 테이블 l에서 두 번째 레코드를 가지고 테이블 J에서 연관된 레코드를 반복해서 찾는다. 조인 처리 방식이 중첩 Loop의 처리방식과 동일하게 처리가 되어 Nested Loop Join 라고 한다.

## ● 5. Join 처리 알고리즘 – Nested Loop

### ■ 실행방법

C 언어 중첩 Loop	Nested Loop 실행계획																																												
<pre>-- Outer Loop for( l=1 ; l &lt;=9 ; l++ ) {      -- Inner Loop     for( J=1 : J &lt;= 9 ; J++ ) {          printf(" %d * %d = %2d \n", l , J);      }  }</pre> 	<table><tr><th>OPERATION</th><th>OBJECT_NAME</th><th>CARDINALITY</th><th>COST</th></tr><tr><td>SELECT STATEMENT</td><td></td><td>1</td><td></td></tr><tr><td>NESTED LOOPS</td><td></td><td>1</td><td></td></tr><tr><td>TABLE ACCESS (BY INDEX ROWID)</td><td>CUSTOMER</td><td>1</td><td></td></tr><tr><td>INDEX (UNIQUE SCAN)</td><td>CUSTOMER_ID_PK</td><td>1</td><td></td></tr><tr><td>Access Predicates</td><td></td><td></td><td></td></tr><tr><td>C.ID='00000999'</td><td></td><td></td><td></td></tr><tr><td>TABLE ACCESS (BY INDEX ROWID)</td><td>EMP</td><td>1</td><td></td></tr><tr><td>INDEX (UNIQUE SCAN)</td><td>EMP_EMPNO_PK</td><td>1</td><td></td></tr><tr><td>Access Predicates</td><td></td><td></td><td></td></tr><tr><td>C.ACCOUNT_MGR=E.EMPNO</td><td></td><td></td><td></td></tr></table>	OPERATION	OBJECT_NAME	CARDINALITY	COST	SELECT STATEMENT		1		NESTED LOOPS		1		TABLE ACCESS (BY INDEX ROWID)	CUSTOMER	1		INDEX (UNIQUE SCAN)	CUSTOMER_ID_PK	1		Access Predicates				C.ID='00000999'				TABLE ACCESS (BY INDEX ROWID)	EMP	1		INDEX (UNIQUE SCAN)	EMP_EMPNO_PK	1		Access Predicates				C.ACCOUNT_MGR=E.EMPNO			
OPERATION	OBJECT_NAME	CARDINALITY	COST																																										
SELECT STATEMENT		1																																											
NESTED LOOPS		1																																											
TABLE ACCESS (BY INDEX ROWID)	CUSTOMER	1																																											
INDEX (UNIQUE SCAN)	CUSTOMER_ID_PK	1																																											
Access Predicates																																													
C.ID='00000999'																																													
TABLE ACCESS (BY INDEX ROWID)	EMP	1																																											
INDEX (UNIQUE SCAN)	EMP_EMPNO_PK	1																																											
Access Predicates																																													
C.ACCOUNT_MGR=E.EMPNO																																													

왼쪽의 Outer Loop와 오른쪽의 TABLE ACCESS CUSTOMER BY INDEX ROWID Operation 와 매핑 하고 Inner Loop와 TABLE ACCESS ORDERS BY INDEX ROWID Operation을 매핑 시켜보면 Nested Loop 조인을 쉽게 이해 할수 있다. CUSTOMER 테이블이 선행(Driving) 테이블이며 Outer Loop에 해당 하고 ORDERS 테이블이 후행(Driven) 테이블이며 Inner Loop에 해당 한다. Outer Loop가 처음 실행 되듯이 TABLE ACCESS CUSTOMER BY INDEX ROWID Operation이 처음 실행되면서 TABLE ACCESS ORDERS BY INDEX ROWID Operation 에 반복적으로 접근 한다.

## ● 5. Join 처리 알고리즘 – Nested Loop

### ■ 실행방법

실행계획에서 + 를 클릭한후 자세한 실행계획을 보자

```
SELECT C.NAME,C.CREDIT_LIMIT,C.EMAIL,
       C.BIRTH_DT,E.ENAME AS MP_NAME,
       E.JOB,E.DEPTNO
FROM   CUSTOMER C , EMP E
WHERE  C.ID = '00000999' and
       C.ACCOUNT_MGR = E.EMPNO;
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	4
NESTED LOOPS		1	4
TABLE ACCESS (BY INDEX ROWID)	CUSTOMER	1	3
INDEX (UNIQUE SCAN)	CUSTOMER.ID_PK	1	2
Access Predicates			
C.ID='00000999'			
TABLE ACCESS (BY INDEX ROWID)	EMP	1	1
INDEX (UNIQUE SCAN)	EMP.EMPNO_PK	1	0
Access Predicates			
C.ACCOUNT_MGR=E.EMPNO			

선행 테이블(CUSTOMER)

① Access Predicates: C.ID = '00000999' 조건을 통해 대상 데이터에 접근

② Nested Loop Join 수행

③ CUSTOMER 테이블에 ①를 사용하여 생성된 결과 집합중 1번째 레코드의 ACCOUNT\_MGR를 가지고 EMP에

④ Access (EMP\_EMPNO\_PK를 통해)

⑤ Filter (위의 SQL에는 Filter 조건이 없다)

⑥ 매칭 성공시 조인 결과 집합에 입력

⑦ CUSTOMER 테이블의 결과 집합중 N번째 레코드 까지 반복

## ● 5. Join 처리 알고리즘 - Nested Loop

### ■ 실행순서 와 성능

바깥쪽 Loop(Outer loop)가 1번 실행 될 때 마다 안쪽 Loop(Inner Loop)는 반복적으로 Loop를 전부 수행해야 한다. Outer Loop의 실행 횟수가 중요 하다. Outer Loop가 100번 실행 되는 것과 10,000번 실행되는 것은 전체 처리 일량을 결정한다. 또한 Inner Loop의 처리가 빨라야 하는 이유는 Outer Loop 반복실행시 마다 Inner Loop가 실행되므로 처리가 느리게 되면 전체 처리가 느려지게 된다.

**선행 테이블(Driving Table)은 데이터 처리 범위가 작은게 효율적**

**후행 테이블(Driven Table)은 데이터에 빠른 접근을 할수 있는게 효율적**

Nested Loop Join의 처리 방식을 보면

① 선행 테이블(Driving)에 따라 일량이 달라진다.

Ex) 선행 테이블의 대상 데이터가

100건 후행 테이블(Driven)에 대한 Access가 100번

1만건 후행 테이블(Driven)에 대한 Access가 1만번

② 후행 테이블에 Access시 Index Scan 으로 처리가 되어야 적절한 응답시간을 보장한다.

Full Table Scan시에는 최악의 결과가 나타나기 때문에 후행 테이블에서 Full Table Scan으로 데이터 Access를 해야 하는 상황이 발생하면 Nested Loop 이외의 조인 처리 알고리즘 선택

## ● 6. Join 처리 알고리즘 - Hash

### ■ Hash

이전 실행계획 실습에서 Group By 연산 수행시

9i: SORT(GROUP BY) vs 10g : HASH GROUP BY

실행계획을 기억 해보자

버전	Operation	전체 Cost	Group By Cost	비율
9i	SORT (GROUP BY)	850	89	69.2%
10g	HASH (GROUP BY)	655	22	3.3%

Sort 연산은 처리 비용이 비싼 연산으로 10g 부터는 GROUP BY 처리 알고리즘이 SORT ➔ HASH 로 변경  
인덱스에서 실행계획 실습을 상기하면 Sort는 비싼 연산이기에 옵티마이저는 정렬 연산을 회피하기 위해  
Index의 정렬된 구성을 사용하려고 하였다.

Hash Join

Sort Merge Join 에 대한 성능 향상을 위해 Oracle 7.3 버전에서 등장한 알고리즘

## ● 6. Join 처리 알고리즘 - Hash

### ■ 실행방법

-----			
Id		Operation	Name
-----			
0		SELECT STATEMENT	
* 1		HASH JOIN	
2		TABLE ACCESS FULL	DEPART
3		TABLE ACCESS FULL	EMP_LARGE
-----			

Hash Join은 선행 테이블 (Driving) 에 따라 성능 편차가 발생하므로 어떤 테이블이 선행테이블인지 확인 해야 한다.

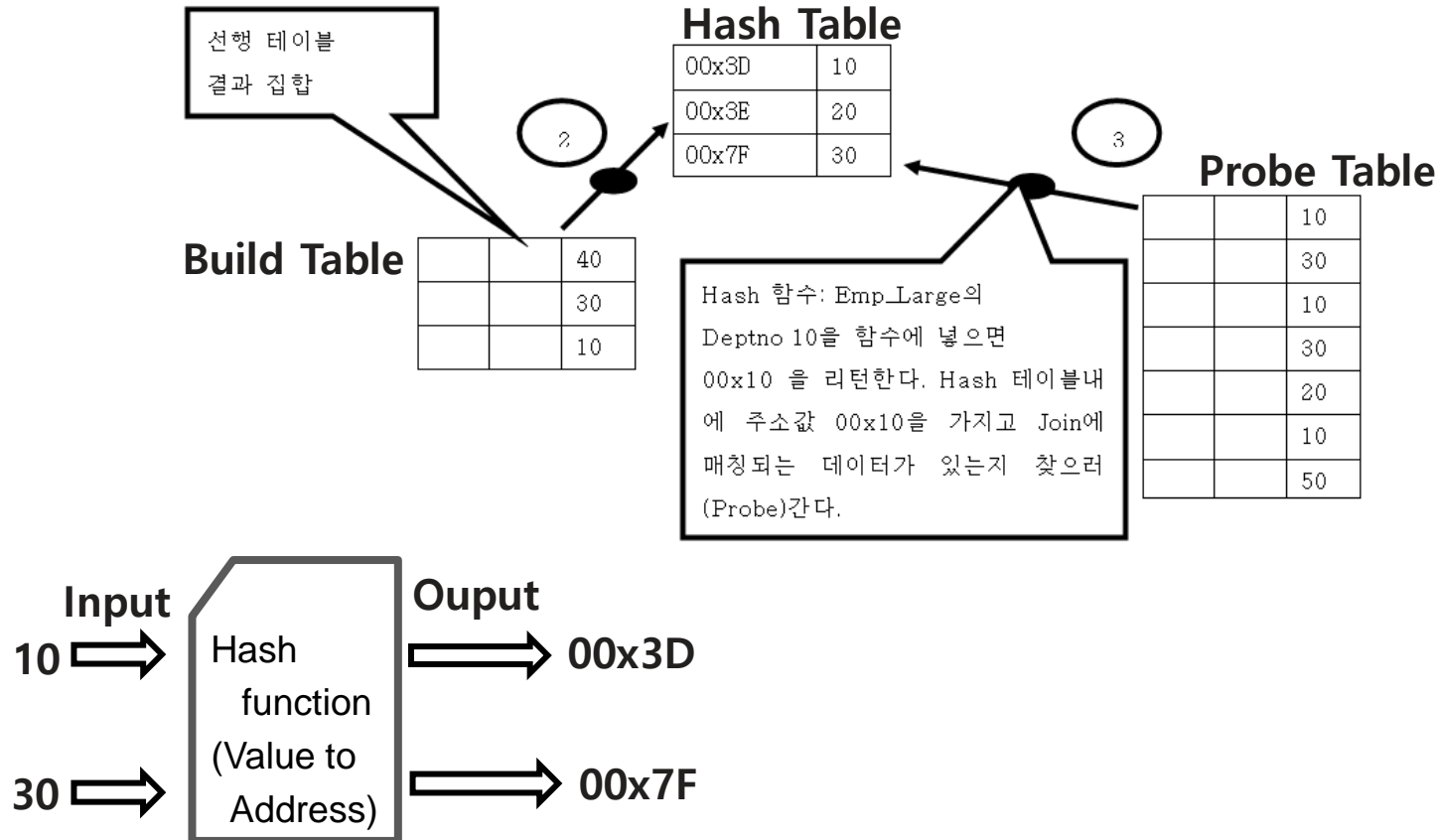
① ID 2,3 은 동일한 Level로 ID 2 가 먼저 실행된다. DEPART 가 선행(Deriving) 테이블이다.

Hash 조인에서는 선행 테이블을 Build 테이블이라 하고 후행 테이블은 Probe 테이블이라 한다  
Hash Join은 Hash 함수와 Hash 테이블(메모리상에 생성되는 테이블)을 사용하여 조인 한다.

② Build 테이블인 DEPART 테이블의 deptno 컬럼(조인 조건에 사용된)의 값을 읽어 메모리상에 Hash 테이블(메모리 구조체)을 생성(Build) 한다. Build 테이블이라는 용어가 쓰인 이유이다.  
Hash 테이블을 생성할 때 deptno의 값을 Hash 함수에 넣어서 생성된 주소 위치에 해당 값을 저장 한다.

## ● 6. Join 처리 알고리즘 - Hash

### ■ 실행방법



- ③ Probe는 사전적 의미로 찾다, 조사하다 탐침하다는 의미를 가진다.  
후행테이블에 해당하는 탐침(Probe) 테이블 Emp\_large의 전체 대상 데이터를 읽으면서  
조인 연결 컬럼인 Deptno 컬럼의 값을 Hash 함수에 넣어서 Hash 테이블 내의 주소 값을  
계산하여 매칭되는 데이터가 있는지 찾아낸다.



## ● 6. Join 처리 알고리즘 - Hash

### ■ 실행방법

실행계획에서 + 를 클릭한후 자세한 실행계획을 보자

```
SELECT  C.NAME,C.CREDIT_LIMIT,C.EMAIL,  
        C.BIRTH_DT,E.ENAME AS MP_NAME,  
        E.JOB,E.DEPTNO  
FROM    CUSTOMER C , EMP E  
WHERE   C.ID >= '00000999' and  
        C.ACCOUNT_MGR = E.EMPNO;
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		5713804	32616
HASH JOIN		5713804	32616
Access Predicates			
C,ACCOUNT_MGR=E,EMPNO			
TABLE ACCESS (FULL)	EMP	15	3
TABLE ACCESS (FULL)	CUSTOMER	5713804	32597
Filter Predicates			
C,ID>='00000999'			
Other XML			

선행 테이블(EMP)

- ① 선행테이블(Build table)인 EMP에서 **조인대상 전체 데이터에 Access**하여 **Hash Table 생성**
  - 조인조건인 EMPNO 컬럼의 값을 Hash functio의 입력키로 사용
- ② 후행테이블(Probe Table)인 Customer에서 조인대상 전체 데이터 (C.ID >= '00000999')를 읽어가면서
  - 조인조건인 ACCOUNT\_MGR 컬럼의 값을 Hash functio의 입력키로 사용하여 Hash Table내에서 매칭되는 데이터 검색

## ● 6. Join 처리 알고리즘 - Hash

---

### ■ 실행순서 와 성능

① 선행 테이블에 따라 성능 편차가 발생 한다. 선행 테이블에서 조인 대상이 되는 전체 데이터를 읽어 메모리상에서 Hash 테이블 생성(Build). 대상 데이터가 많으면 Hash 테이블에 저장되는 레코드를 많이 만들기 위해 메모리 자원 및 CPU 자원 (Hash 함수 연산 및 테이블 생성 작업)을 많이 사용.

② Sort Merge Join과 비교하여

공통점: 조인에 참여하는 전체 대상 데이터를 1번은 전부 읽는다.

차이점: 정렬을 하지 않는다. 조인을 매칭 하기 위해 Hash Function으로 주소를 계산하기 때문에 다른 Join 알고리즘 보다 CPU 자원을 많이 사용.

③ Hash 테이블이 메모리에 생성 되기 때문에 Hash Join이 효율적으로 수행되기 위해서는 DBA 와 협의해서 적절한 메모리 공간이 확보 되어 있는지 확인 필요

④ 다음은 주요 Access PATH

- Full Table Scan      - Index Scan      - Direct Access

이중 Direct Access는 단일 레코드에 접근하는 가장 빠른 방법

Direct Access 는 ① Rowid ② Hash 2가지 방법이 있다.

Hash는 데이터에 직접 접근하는 가장 빠른 방법중 하나

## ● 6. Join 처리 알고리즘 – Merge Join

### ■ (Sort) Merge

각각 테이블에서 조인 조건(컬럼)을 기준으로 Sort후 각각의 결과 집합을 Merge하여 Join

### ■ 실행방법

Id	Operation	Name
0	SELECT STATEMENT	
1	MERGE JOIN	
2	SORT JOIN	
3	TABLE ACCESS FULL	DEPART
* 4	SORT JOIN	
5	TABLE ACCESS FULL	EMP_LARGE

#### (1) Sort 단계

첫번째 대상 테이블 :

- ① Access : 대상 데이터에 접근
- ② Filtering : 불필요한 데이터를 제거
- ③ Sort : 조인 컬럼을 기준으로 정렬 수행하여  
정렬된 결과 집합1 (Result set) 생성

## ● 6. Join 처리 알고리즘 – Merge Join

---

### ■ 실행방법

두번째 대상 테이블 :

- ① Access : 대상 데이터에 접근
- ② Filtering: 불필요한 데이터를 제거
- ③ Sort : 조인 컬럼을 기준으로 정렬 수행 하여  
정렬된 결과 집합2 (Result set) 생성

- 각자 정렬하는 것이므로 선행 테이블, 후행 테이블의 개념이 없습니다.

정렬이 먼저 되는 순서가 일량에 영향을 주지 않기 때문에 Join 성능에도 영향을 주지 않는다.

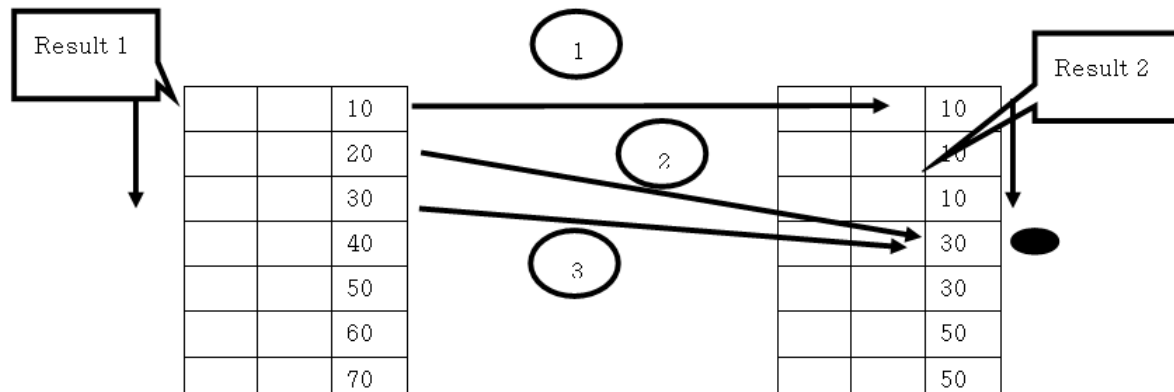
\* Result 1 , Result 2 는 정렬 연산의 결과 집합이라는 의미이며 DBMS가 부여한 결과 집합의 이름은 아니다.

## ● 6. Join 처리 알고리즘 – Merge Join

### ■ 실행방법

#### (2) Merge 단계

- ① Result 1의 첫번째값(10)을 가지고 Result2의 첫번째값(10)부터 매칭되는 값을 차례로 찾고 매칭 되지 않으면 멈춘다. Result 2의 네번째 레코드(30)에서 멈춘다
- ② Result 1의 두번째 레코드(20)를 가지고 Result2의 멈춘 부분(30)과 비교한다.  
매칭되는 데이터가 없다면 Result 1의 다음 레코드로 이동한다.
- ③ Result 1의 세번째 레코드(30)를 가지고 Result2의 멈춘 부분(30)과 비교한다.  
매칭되는 데이터가 있다면 Result 2의 다음 레코드(30)로 이동한다



## ● 6. Join 처리 알고리즘 – Merge Join

### ■ 실행방법

실행계획에서 + 를 클릭한후 자세한 실행계획을 보자

```
SELECT C.NAME,C.CREDIT_LIMIT,C.EMAIL,  
       C.BIRTH_DT,E.ENAME AS MP_NAME,  
       E.JOB,E.DEPTNO  
FROM   CUSTOMER C , EMP E  
WHERE  C.ID >= '00000999' AND  
       C.ACCOUNT_MGR = E.EMPNO  
ORDER BY C.ACCOUNT_MGR;
```

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		5713804	107202
MERGE JOIN		5713804	107202
TABLE ACCESS (BY INDEX ROWID)	EMP	15	3
INDEX (FULL SCAN)	EMP_EMPNO_PK	15	1
SORT (JOIN)		5713804	107199
Access Predicates			
Filter Predicates			
TABLE ACCESS (FULL)	CUSTOMER	5713804	32597
Filter Predicates			
	C.ID>='00000999'		
Other XML			

- ORDER BY C.ACCOUNT\_MGR DESC; 으로 변경후 실행계획 조회  
ORDER BY C.CREDIT\_LIMIT;                      으로 변경후 실행계획 조회

## ● 6. Join 처리 알고리즘 – Merge Join

---

### ■ 실행순서 와 성능

① Merge 연산을 수행할 때 양쪽 결과 집합에 레코드를 가리키는 포인터가 있다. 매칭되는 결과에 따라 각각의 포인터를 하나씩 밑으로 내리면서 Merge를 수행 한다. 이런 방식으로 Merge를 하기

때문에 Result 1, Result 2를 처음부터 끝까지 전부 읽어 가면서 수행한다. 둘중 어느 테이블이 선행

테이블이 되어도 일량은 동일 하다

② Merge 조인은

- 인덱스가 없어도 수행 될수 있다. 인덱스가 있는 경우는 Sort 작업시 성능 효과를 얻을수 있는 가능성이 있고 Merge 연산시에는 관련이 없다.
- 옵티마이저 Mode가 ALL\_ROWS인 경우에 자주 발생 한다.
- Sort 연산에 많은 처리 비용이 발생 한다.