1. Nested Loop Join

Nested Loop Join의 방식은 **두개의 테이블의 행을 각각 모두 확인하여 조인하는 방법입니다.** 표현하자면 **중첩된 for문**입니다. inner와 outer loop이 있듯이 조인에는 dirving과 driven 테이블이 있습니다. 실행계획에서 먼저 실행되는 테이블이 driving 테이블이고 나중에 실행되는 것이 driven 테이블입니다. 중첩된 for 문이라고 표현한 이유는 각각 테이블을 모두 읽고 확인하는 것이아니라 각 행별로 확인하기 때문에 **행이 적은 테이블을 driving 테이블로 선정하는 것이 빠른결과**를 얻을 수 있는 방법입니다. 또한 **조인 컬럼에 인덱스가 있어야 테이블 전체를 탐색하지 않고 필요한 행에대해서만 탐색하여 효율적**입니다.

인덱스가 없는 두테이블에 대해서 Nested Loop Join을 하는 경우 driving table의 각각 행에 대해 driven table을 Full Scan하기 때문에 인덱스에 많이 의존하게 됩니다. 반면, Sorted Merge Join은 인덱스가 없어도 조인칼럼을 기준으로 정렬한 후 inner table 조회시 outer table의 값에 대해서만 조인하면 되기 때문에 인덱스가 없는 경우에 사용될 수 있고 이러한 정렬 작업에 대한 부하를 감수한다면 Nested Loop Join 보다 유리할 수 있습니다.

예시

```
JAMONG@orcl> create table test as select * from employees;

JAMONG@orcl> insert into test select * from test;
JAMONG@orcl> insert into test select * from test;
JAMONG@orcl> insert into test select * from test;
JAMONG@orcl> insert into test select * from test;
JAMONG@orcl> insert into test select * from test;
JAMONG@orcl> insert into test select * from test;
JAMONG@orcl> insert into test select * from test;
JAMONG@orcl> insert into test select * from test;
JAMONG@orcl> insert into test select * from test;
JAMONG@orcl> insert into test select * from test;
JAMONG@orcl> insert into test select * from test;
JAMONG@orcl> insert into test select * from test;

JAMONG@orcl> create index test_dept_idx on test(department_id);
```

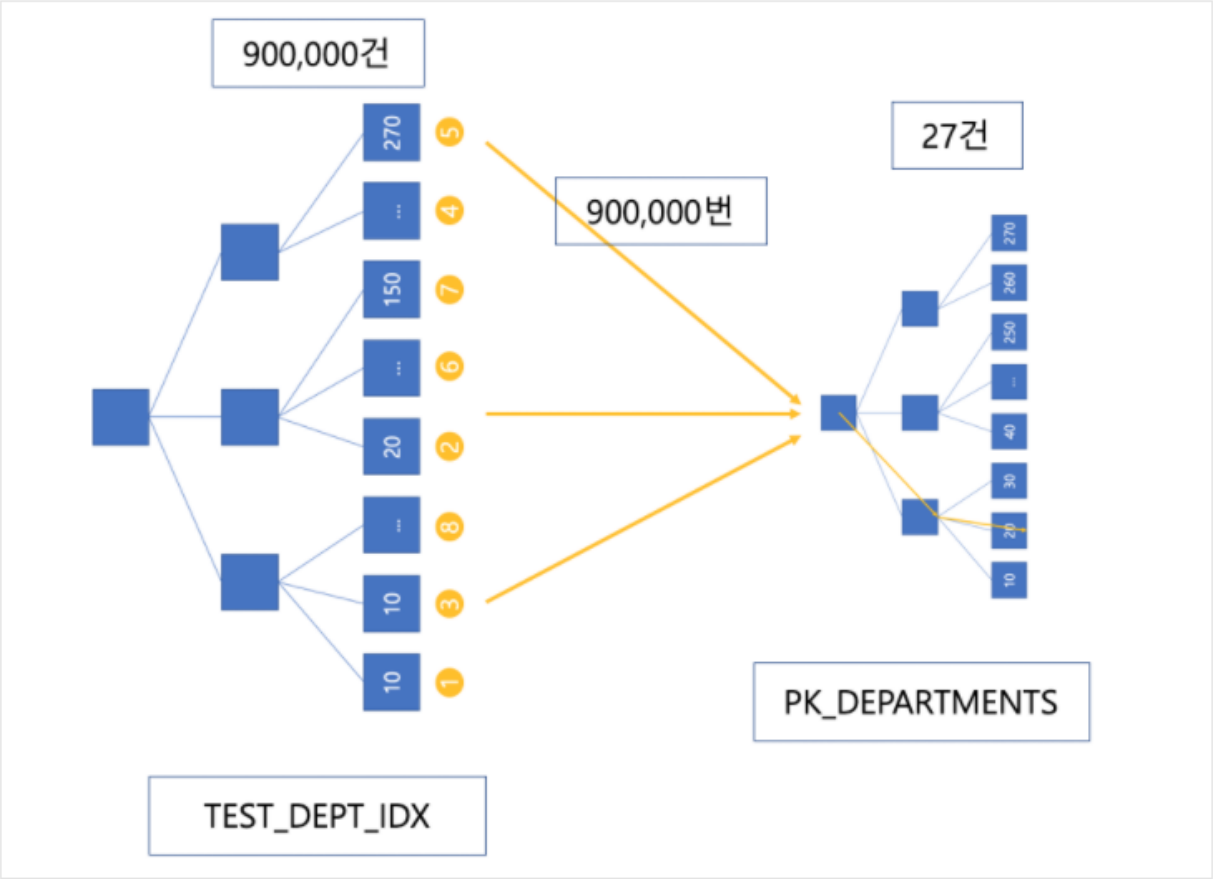
성능의 확연한 차이를 확인하기 위해 test테이블을 90만건 정도의 테이블로 생성했습니다. 27건의 데이터를 갖고 있는 departments 테이블과 조인해서 결과를 확인해볼것입니다.

```
JAMONG@orcl> @start
JAMONG@orcl>
select /*+ leading(e) use_nl(d) */ count(*)
from test e, departments d
where e.department_id = d.department_id;
JAMONG@orcl> @end
```

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2     70.01      70.24          0       2441          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4     70.01      70.24          0       2441          0           1

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 92

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=2441 pr=0 pw=0 time=0 us)
 868353   NESTED LOOPS  (cr=2441 pr=0 pw=0 time=63278384 us cost=711 size=19474728 card=749028)
 868353    INDEX FAST FULL SCAN TEST_DEPT_IDX (cr=2437 pr=0 pw=0 time=7029097 us cost=686 size=9814181 card=754937)(object id 74640)
 868353    INDEX UNIQUE SCAN PK_DEPARTMENTS (cr=4 pr=0 pw=0 time=0 us cost=0 size=13 card=1)(object id 74631)
```



힌트를 사용해서 leading --> test 테이블을 driving 테이블로 설정하고 특별한 필터링없이 Nested Loop Join으로 실행하고 실행계획을 그림으로 그려봤을 때 위와 같습니다. 실행하는데 70초정도가 소요되고 그럼 diving과 driven 테이블을 바꾸고 실행한 후 비교해보겠습니다.

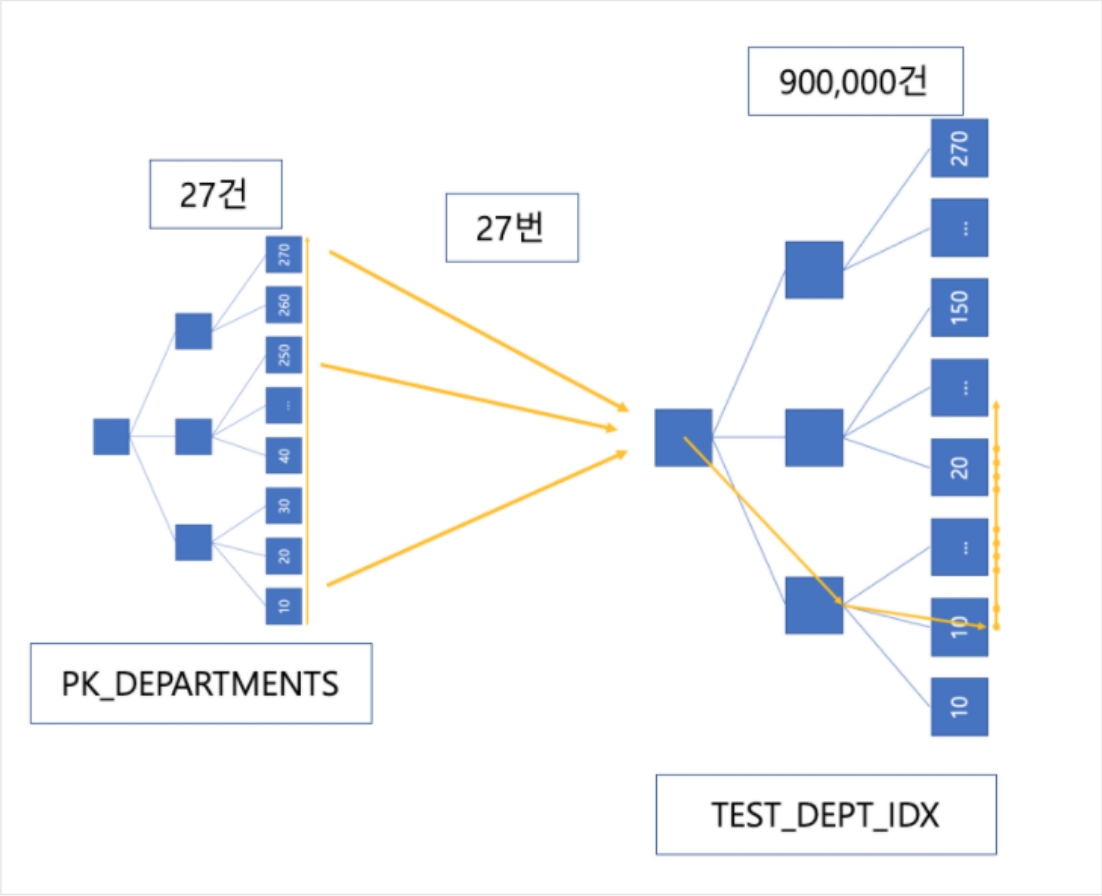
```
JAMONG@orcl> @start
JAMONG@orcl>
select /*+ leading(d) use_nl(e) */ count(*)
from test e, departments d
where e.department_id = d.department_id;
JAMONG@orcl> @end
```

copy sql

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	27.15	27.22	0	2417	0	1
total	4	27.15	27.22	0	2417	0	1

```
Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: 92

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=2417 pr=0 pw=0 time=0 us)
 868353   NESTED LOOPS  (cr=2417 pr=0 pw=0 time=20429136 us cost=516 size=19474728 card=749028)
     27    INDEX FULL SCAN PK_DEPARTMENTS (cr=1 pr=0 pw=0 time=262 us cost=1 size=351 card=27)(object id 74631)
 868353    INDEX RANGE SCAN TEST_DEPT_IDX (cr=2416 pr=0 pw=0 time=6882812 us cost=19 size=360646 card=27742)(object id 74640)
```



결과적으로 봤을때 driving 테이블이 departments일 때 실행 시간 27초로 71초보다 2배 이상 빠른 속도로 결과를 갖고 왔습니다. 실행계획을 봤을때 읽는 블록의 갯수는 크게 차이가 없습니다. 결과는 같으므로 반환되는 row의 갯수도 같은데 속도차이는 크게 납니다. count된 결과를 받는것으로 Table 에 대한 Random Access 필요없이 인덱스만 갖고 결과를 받아왔습니다. 그러면 어떤 이유로 차이가 난걸까요? 바로 driving 테이블의 행의 갯수가 비교적 적다는 이유이고 Nested Loop Join이 레코드 하나씩 순차적으로 조인작업을 한다는 특징이 확실하게 보여집니다.

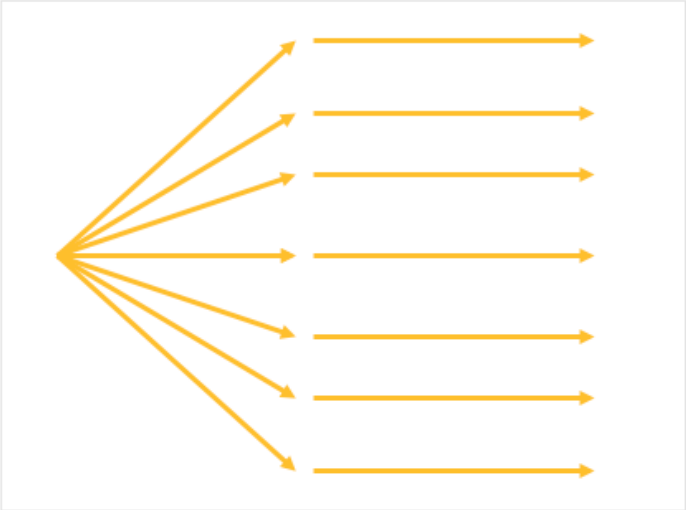
↑

└ · Outer table 에서 한 레코드씩 가져와서, inner table과 join을 실시함.

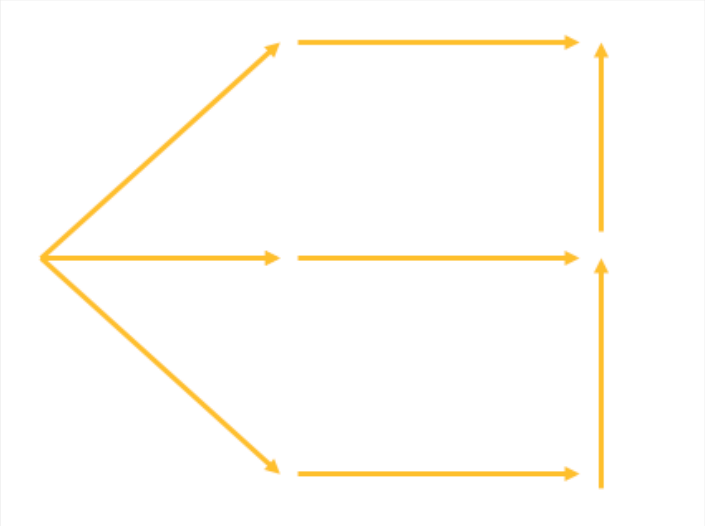
· 이에 따라, 두 테이블의 join 컬럼 조건이 where clause의 filter predicate로 동작함.

=> inner table에 index unique scan 일 경우.

driving 테이블이 test일때의 상황을 먼저 보겠습니다. 레코드 하나씩 확인하기 때문에 먼저 TEST_DEPT_IDX를 각각 90만번 확인하면서 PK_DEPARTMENTS 인덱스를 Unique Scan으로 한번씩만 확인하여 결과를 받아옵니다. 그림으로 표현하자면 아래와같이 뻗어나가면서 데이터를 조인합니다.



driving 테이블이 departments일때는 레코드를 PK_DEPARTMENTS에서 먼저 다 확인하고 각 레코드별로 Range Scan을 통해 TEST_DEPT_IDX를 탐색합니다. 아래와 같은 식으로 순회합니다.



3. Hash Join

Hash Join은 Hash Table을 생성하여 Hash Function에 의한 탐색을 하여 조인합니다. 주로 대용량의 데이터를 사용할 때 사용되며 일반적으로 Nested Loop Join이나 Sorted Merge Join보다 빠르다고 합니다. 알고리즘에서 시간 복잡도의 개념으로 봤을 때도 Hash Function을 사용하게 되면 O(1)의 시간 복잡도를 갖게되니까 빠를수 밖에 없는 것 같습니다. 하지만, 해시 충돌을 방지나 해시 체인의 크기가 커지는 것을 막기 위해 **중복되는 데이터가 적은 경우에 사용되어하고** Hash Table을 생성하는데 Hash Area에 충분히 담길 정도로 데이터 양이 작아야합니다.

해시 테이블을 만들 때 해시 충돌이 발생하면 그런시 'O(1)' 속도가 나오지 않는다.

예시

```
JAMONG@orcl>
select /*+use_hash(d,e)*/
    d.department_id,
    d.department_name,
    e.last_name,
    e.salary
from departments d, employees e
where d.department_id = e.department_id;
JAMONG@orcl> @xplan
```

copy sql

```
select /*+use_hash(d,e)*/ d.department_id,d.department_name,e.last_name,
e.salary from departments d, employees e where d.department_id =
e.department_id

Plan hash value: 2052257371

-------------------------------------------------------------------------------------------------------------
| Id  | Operation          | Name        | Starts | E-Rows | Cost (%CPU)| A-Rows |   A-Time   | Buffers |  OMem |  1Mem | Used-Mem |
-------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |      1 |        |     7 (100)|    106 |00:00:00.04 |       9 |       |       |          |
|*  1 |  HASH JOIN         |             |      1 |    106 |     7  (15)|    106 |00:00:00.04 |       9 |  1079K|  1079K| 1238K (0)|
|   2 |   TABLE ACCESS FULL| DEPARTMENTS |      1 |     27 |     3   (0)|     27 |00:00:00.01 |       3 |       |       |          |
|   3 |   TABLE ACCESS FULL| EMPLOYEES   |      1 |    107 |     3   (0)|    107 |00:00:00.01 |       6 |       |       |          |
-------------------------------------------------------------------------------------------------------------

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

   1 - access("D"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")
```

힌트를 사용해서 결과를 확인해봤습니다. Hash Join 수행시 메모리를 사용하여 Hash Table을 사용하는 것을 확인할 수 있습니다. Hash Join이 빠르다고 하는 이유 또한 **PGA영역을 사용하여 Latch를 확득 과정이 없어 빠르게 결과를 얻을 수 있습니다.**

- `NL_Join'라는 라르게, `join 칼럼 조건'이 where clause의 filter predicate로 동작하지 않는다.

(Outer table의 한 레코드를 찾고, inner table을 여러번 scan하는 방식이 아니기 때문!)