# Join Methods

## Task 2: Nested Loop Join

The nested join is sufficient for a small subsets of data, here is also included big subset of data **which is filtererd** (top 10 for example)

### Query, where from the huge table are selected only 10 rows.

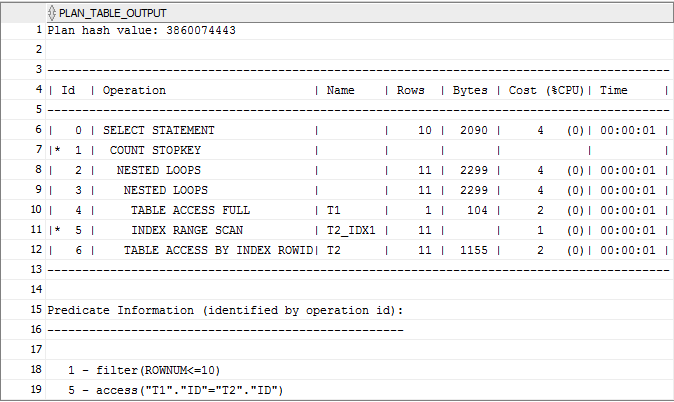
select \* from (

select \*

from t2

inner join t1

on t1.id=t2.id

) where rownum<=10

### Query, where all rows are selected.

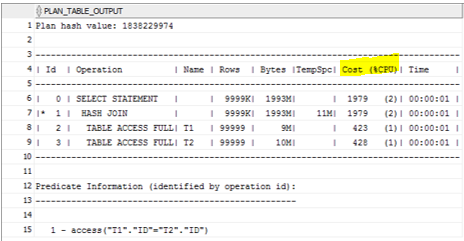
If we remove rownum<=10,a hash join will be used, because a large amount of data is joined (9998901 rows)

select \*

from t2

inner join t1

on t1.id=t2.id



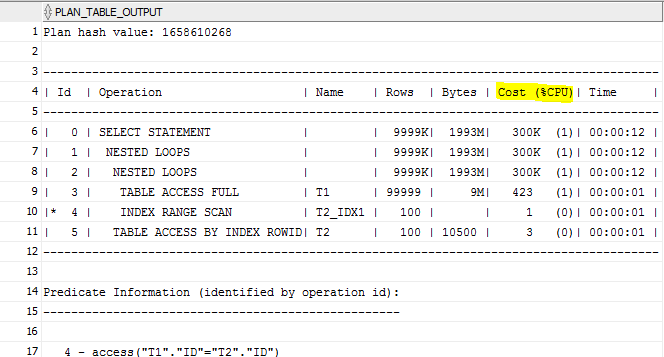
Here the same query if we use **nested loop join**:

select /\*+ leading(t1,t2) use\_nl(t2) \*/ \*

from t2

inner join t1

on t1.id=t2.id



As we can see the difference is tremendous.

## Task 3: Sort-Merge Join

select \* from merge\_emp f

inner join merge\_emp\_1 s

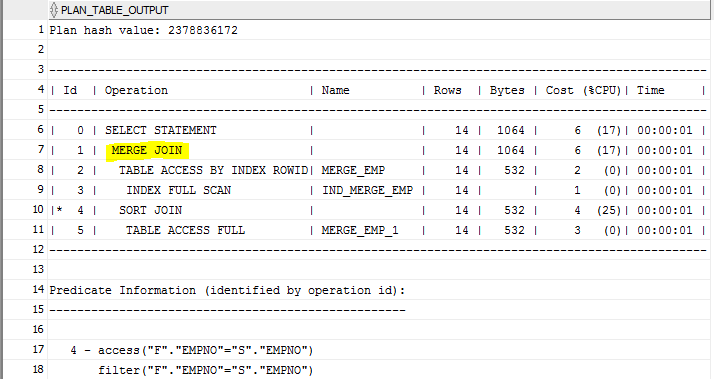
on f.empno=s.empno

**Sort-merge join** is a variation of nested loop, which is performed when 2 tables are sorted (due to indexes or user commands).

**N.B.** It’s not optimal when 2 tables are sorted by user , because sorting is difficult operation for a database.

Two tables were created: one has index on empno, and one (with the same structure and data) doesn’t.

So, the optimizer performed full table access to the second table (without index) and sorted accessed data, then retrieved all data from the first table by index rowid (it’s already sorted) and then merged it.

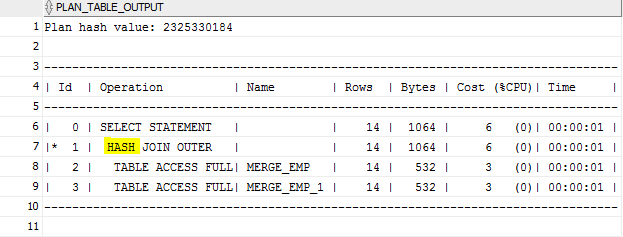


## Sort-merge join vs Hash join

select \* from merge\_emp f

left join merge\_emp\_1 s

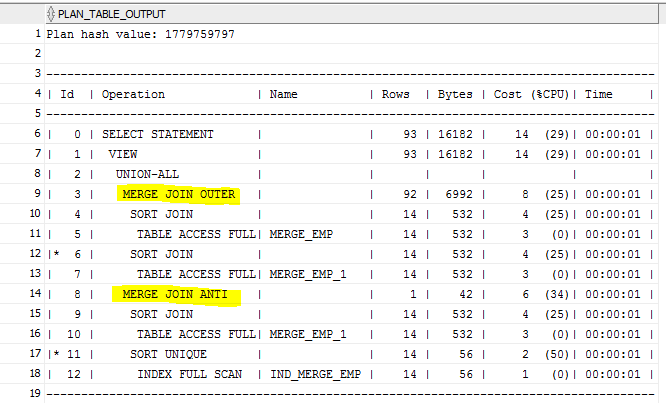
on f.empno=s.empno



select \* from merge\_emp f

left join merge\_emp\_1 s

on f.empno>s.empno



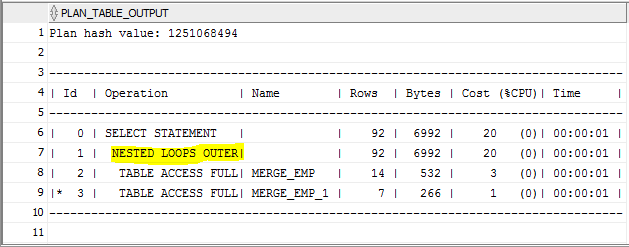
Here is used merge join, because it **isn’t equijoin** anymore.

**N.B. Even if we’ll give hint to use hash join it won’t happen. So, the conclusion is, that equijoin is the must for hash join.**

select /\*+ USE\_HASH(s)\*/ \* from merge\_emp f

left join merge\_emp\_1 s

on f.empno>s.empno

;

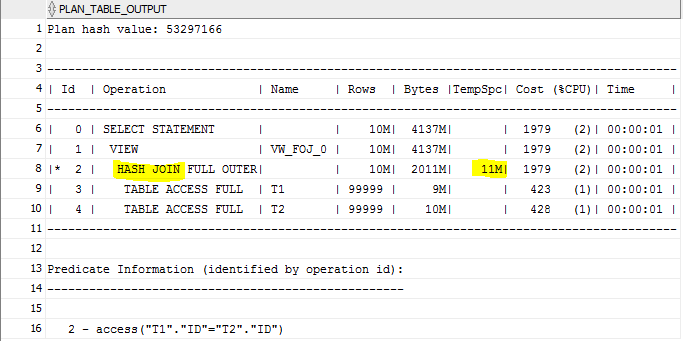
## Task 4: Hash Join

select \*

from t2

full join t1

on t1.id=t2.id



Here hash join is the most optimal, because **large amount of data** should be joined (10088901). Here we can also see that the data set does not fit in memory, so I/O was made to the **temporary tablespace.**

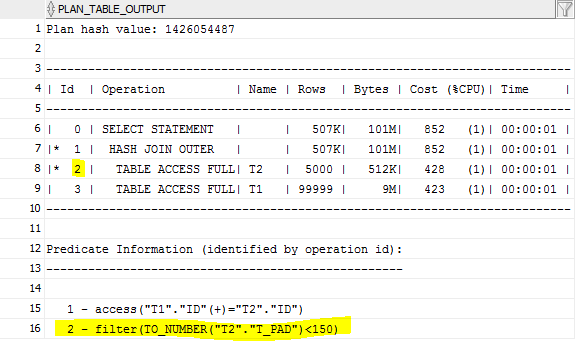
select \*

from t2

full join t1

on t1.id=t2.id

where t2.t\_pad<150



Here we added a criteria in the WHERE clause and see that database has applied where clause and then joined 2 tables. The second table is significantly smaller now and therefore the database **doesn’t need temporary tablespace.**

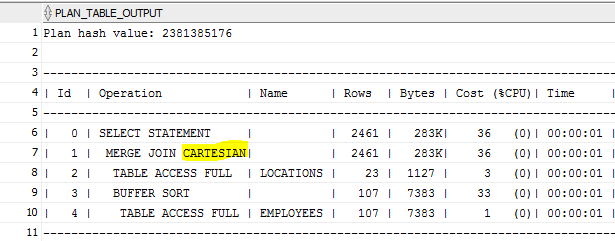
## Task 5: Cartesian Join

Cross(Cartesian) join is when all the rows from one table are joined to all the rows of another table. So the overall number of rows in the end should be equal to multiplication of number of rows of two tables (in my example 107\*23=2461)

select \*

from employees

cross join locations



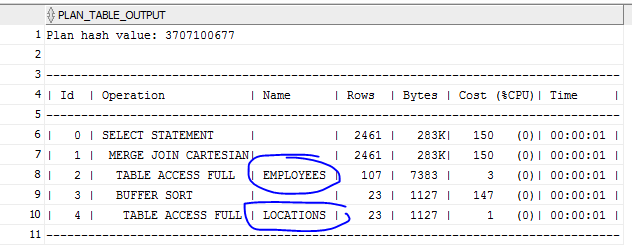
Here is used merge join, because both tables were sorted(employees – buffer sorting, locations via indexes). Location was chosen as an inner (ведущая) table because it’s smaller and therefor it will be **easier for database to do a join operation.**

### ORDERED hint and wrong predicate order.

select /\*+ ordered \*/ \*

from employees

cross join locations

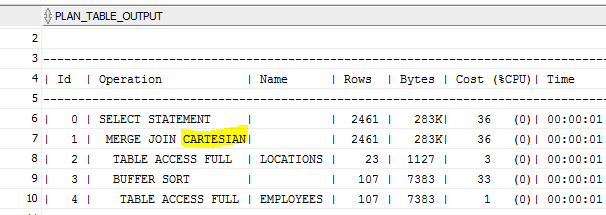


Now we see, that order chosen by Oracle was 3times **more optimal** than the second one.

### W/o join conditions

select \*

from employees e, locations l



## Task 6: Left/Right Outer Joins

**Left join** returns all rows from one table and only those rows from the joined table where the join condition is met. In the left join driven-to (к чему присоединяют) table is a table that is mentioned in the from condition. Driving (присоединяемая) is a table that is mentioned in the left join condition. For the **right join** is vice versa.

### Left join

**What happens here:** We have 27 departments and 27 rows, because all departments have locations.

**ANSI-syntax:**

select \* --27

from departments d

left join locations l

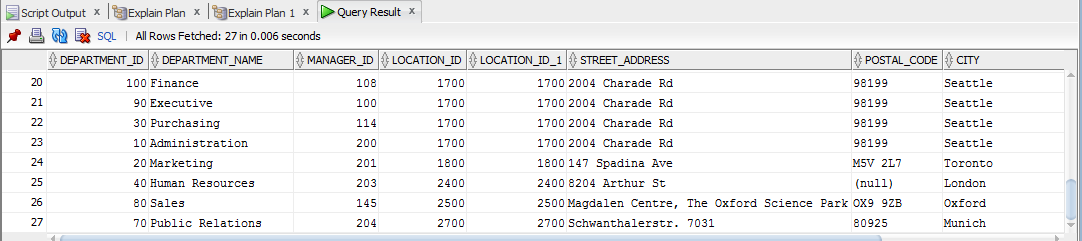
on l.location\_id=d.location\_id

**Oracle syntax:**

select \* --27

from departments d, locations l

where l.location\_id(+)=d.location\_id



### Right join

**What happens here:** We have 23 locations and 43 rows, because not all locations have departments, so they have null values in all columns connected with department.

**ANSI-syntax:**

select \* --43

from departments d

right join locations l

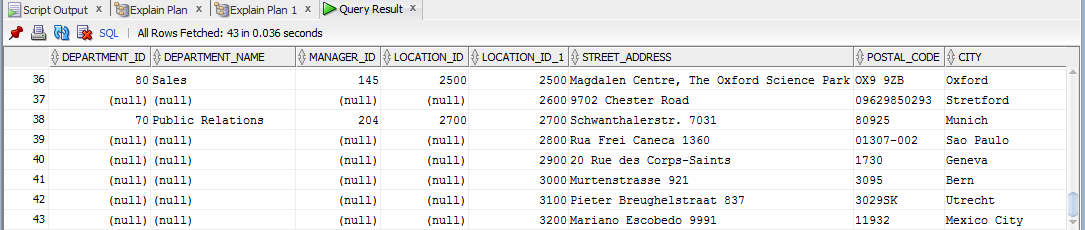
on l.location\_id=d.location\_id

**Oracle syntax:**

select \* --43

from departments d,locations l

where l.location\_id=d.location\_id(+)



## Task 7: Full Outer Join

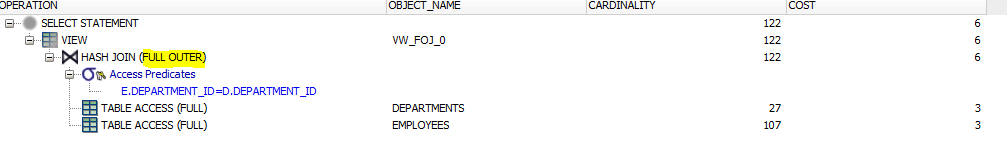
Full outer join is equal to left join union all right join. The full outer join will return all the rows from both tables that match plus the rows that are unique to each table.

select \* --123

from departments d

full join employees e

on e.department\_id=d.department\_id



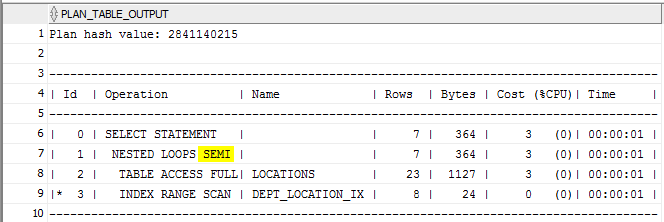
Returns 123 rows (matched ones + unique for each table)

### Semi join

select \*

from locations l

where l.location\_id in (select location\_id from departments d)



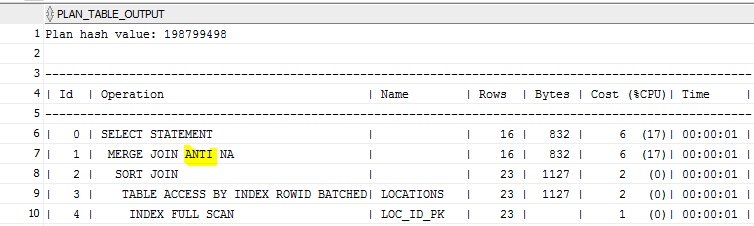
A semi join is chosen because of **IN.**

### Anti join

select \*

from locations l

where l.location\_id not in (select location\_id from departments d)

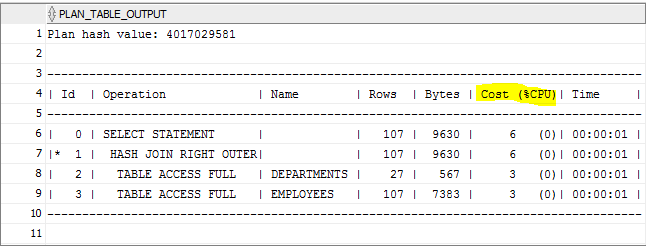


An anti join is chosen because of **NOT IN.**

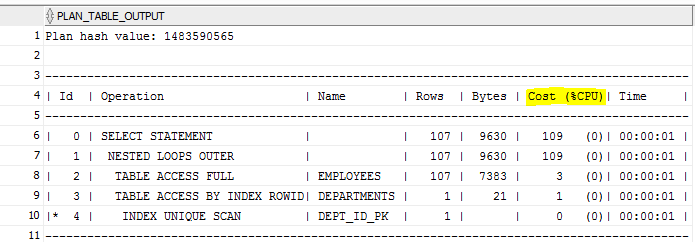
## Task 8: Results

|  |  |  |  |
| --- | --- | --- | --- |
| Table “A” | Table “B” |  | Join type description |
| Small table with index on join field | IOT | select \* from regions r  left join countries c  on r.region\_id= c.region\_id | MERGE JOIN was selected by optimizer because both tables have indexes and therefore they are already sorted (furthermore the second table is IOT table) |
| The same table A from the previous w/o index | The same table B from the previous w/o index | create table reg\_1 as  select \* from regions;  create table coun\_1 as  select \* from countries;  select \* from reg\_1 r  left join coun\_1 c  on r.region\_id= c.region\_id | It’s interesting, but HASH JOIN was used (i assumed it’ll be nested).  Maybe because 2 full scans are required, so it will be faster to use hash join. |
| Small table with index  on join field | Small table w/o index on join field | select \* from merge\_emp f  inner join merge\_emp\_1 s  on f.empno=s.empno | MERGE JOIN, because inner(ведущая) table is sorted. |
| Small table w/o index on join field | Small table w index on join field | select \* from employees e  left join departments d  on e.department\_id=d.department\_id  where d.department\_id=30 | Because of WHERE predicate NESTED LOOP was used. |
| Small table w/o index on join field | Small table w index on join field | select \* from employees e  left join departments d  on e.department\_id=d.department\_id | Here both table has the same type as in the previous situation,but without WHERE condition.and therefore HASH JOIN was used. In this case it was faster to use this type (proof below)\*. |
| Small table w index on join field | Medium table w/o index on join field | select \* from departments d  left join employees e  on D.department\_id=E.department\_id | MERGE JOIN, bcs rows in department (ведущая таблица) are sorted. |
| Medium table w/o index on join field | Small table w index on join field | select \* from employees e  left join departments d  on E.department\_id=D.department\_id | HASH JOIN, bcs in employees rows aren’t sorted. |
| Big table with index on join field | Big table w/o index on join field | select \*  from t2  inner join t1  on t1.id=t2.id | HASH JOIN, because it’s equijoin with a large amount of data (deapite of index). |
| Big table with index on join field | Big table w/o index on join field | select \* from merge\_emp f  full join merge\_emp\_1 s  on f.empno>s.empno | MERGE JOIN, bcs it’s not equijoin and MERGE\_EMP has index. |
| Small table w index on join field | Small table w index on join field | select \* from job\_history h  left join jobs j  on h.job\_id=h.job\_id | NESTED LOOP was used, because 2 tables are small and it faster to use NESTED LOOP, then sort the second table (because it has more than 1 index) |

### Proof:



With a hint to use NESTED LOOP:



### Overall conclusion: There are a lot of factors which are influence on the selection of an optimal execution plan (size, indexes, conditions). And even though two cases may seem the same they can be treated in the different ways.