Tutorial (Query Hints)

1. Create Tables

As a case study, let's create the following FIVE tables by importing from dtaniar account as follows:

```
Create Table Customer1 As Select * From dtaniar.Customer1;
Create Table Item1 As Select * From dtaniar.Item1;
Create Table Inventory1 As Select * From dtaniar.Inventory1;
Create Table Order1 As Select * From dtaniar.Order1;
Create Table Order_Inv1 As Select * From dtaniar.Order_Inv1;
```

**Note that because we import the tables, only the tables and the records are imported (copied). PKs and FKs are not copied. Hence, the five tables above do not have PKs and FKs.

Look at the records in the tables that you have just imported.

```
Select * From Customer1;
Select * From Item1;
Select * From Inventory1;
Select * From Order1;
Select * From Order_Inv1;
```

2. Set up the Environment

If you are using **SQL Developer**, you could use either the following statement:

```
EXPLAIN PLAN FOR
SELECT <attribute(s)>,<fact measurement>
FROM <table name>
WHERE <Conditions>
GROUP BY <attributes>;

/*
Once you have run those queries to view the execution
plan run the following query
*/

SELECT * FROM table(dbms_xplan.display);
```

OR click the icon highlighted by the red box.

If you are using **SQL*Plus:**

1. Set up the line size (length of each line), using the following command:

```
SET linesize 150
```

2. Turn the Autotrace on, by either:

```
SET Autotrace on
```

OR

```
SET Autotrace Traceonly Explain
```

OR

```
Explain Plan
```

(Note that at the end of the experimentations, you need to turn off the Autotrace, by using `Set Autotrace Off`).

3. Join Query

The first query is a join query between Customer1 and Order1.

```
SQL> Explain Plan For
   2 Select *
   3 From Customer1 C, Order1 O
   4 Where C.CustID = O.CustID;

Explained.

SQL> Select * From Table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
Plan hash value: 3340424740

-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    10 |  1270 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN         |           |    10 |  1270 |     7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| CUSTOMER1 |     7 |   476 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| ORDER1    |    10 |   590 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("C"."CUSTID"="O"."CUSTID")

Note
-----
   - dynamic sampling used for this statement (level=2)

19 rows selected.
```

Questions:

(1) Draw the query tree based on the above execution plan

(2) Explain why the join operation used is a Hash Join

Answers: Write your answers here

4. Hints

/*+ HINTS */ is a hint provided by the user so that the system may follow the user's instruction. If you would like to instruct the system to use a SORT-MERGE JOIN operation, instead of a HASH JOIN for the join query above, then the USE_MERGE hint can be used, as follows:

```
SQL> Explain Plan For
   2 Select /*+ USE_MERGE (C O) */ *
   3 From Customer1 C, Order1 O
   4 Where C.CustID = O.CustID;

Explained.

SQL> Select * From Table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------
Plan hash value: 1292085652
------------------------------------------------------------------------------------
| Id  | Operation           | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |    10 |  1270 |     8  (25)| 00:00:01|
|   1 |  MERGE JOIN         |           |    10 |  1270 |     8  (25)| 00:00:01|
|   2 |   SORT JOIN         |           |     7 |   476 |     4  (25)| 00:00:01|
|   3 |    TABLE ACCESS FULL| CUSTOMER1 |     7 |   476 |     3   (0)| 00:00:01|
|*  4 |   SORT JOIN         |           |    10 |   590 |     4  (25)| 00:00:01|
|   5 |    TABLE ACCESS FULL| ORDER1    |    10 |   590 |     3   (0)| 00:00:01|
------------------------------------------------------------------------------------

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

   4 - access("C"."CUSTID"="O"."CUSTID")
       filter("C"."CUSTID"="O"."CUSTID")

Note
-----
   - dynamic sampling used for this statement (level=2)

22 rows selected.
```

Questions:

(1) Draw the query tree based on the above execution plan

(2) Explain why the join operation used is a Merge Join

Answer: Write your answers here

5. Primary Key

Now, lets create a PK for the CUSTOMER table:

```
Alter Table Customer1
Add Constraint Cust_ID_PK PRIMARY KEY (CustID);
```

Try the same join query as above, and notice that now the query tree uses the INDEX. In this case, it uses a SORT-MERGE operation with INDEX. This proves that PK is implemented by an Index. This means that when an attribute is PK, by accessing the index of that attribute, the table is sorted based on that attribute.

```
SQL> Explain Plan For
   2 Select *
   3 From Customer1 C, Order1 O
   4 Where C.CustID = O.CustID;

Explained.

SQL> Select * From Table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
Plan hash value: 356031358
------------------------------------------------------------------------------------------
| Id  | Operation                     | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |    10 |  1270 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                   |             |    10 |  1270 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| CUSTOMER1   |     7 |   476 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | CUST_ID_PK  |     7 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |             |    10 |   590 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL          | ORDER1      |    10 |   590 |     3   (0)| 00:00:01 |
------------------------------------------------------------------------------------------

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

   4 - access("C"."CUSTID"="O"."CUSTID")
       filter("C"."CUSTID"="O"."CUSTID")

Note
-----
   - dynamic sampling used for this statement (level=2)

22 rows selected.
```

Table Order1 has to be sorted (line 4), but table Customer1 is not. Instead, the index (Cust_ID_PK), which is the PK of CustID attribute is read. CustID in the index is already sorted. So, line 3 reads the sorted CustID index entry one-by-one, and then access the corresponding record in table Customer1. Then a MERGE JOIN operation (line 1) is to join both tables Order1 and Customer1.

Question: Draw the query tree based on the above execution plan.

Answer: Write your answer here

The USE_NL hint instructs the system to use the Nested Loop join operation. The nested loop operation takes Customer1 and Order1 tables, and then performs a nested-loop join.

```
SQL> Explain Plan For
   2 Select /*+ USE_NL (C O) */ *
   3 From Customer1 C, Order1 O
   4 Where C.CustID = O.CustID;

Explained.

SQL> Select * From Table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 3381682769

------------------------------------------------------------------------------
| Id  | Operation          | Name       | Rows | Bytes| Cost(%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |            |   10 | 1270 |  12  (0)  | 00:00:01|
|   1 |  NESTED LOOPS      |            |   10 | 1270 |  12  (0)  | 00:00:01|
|   2 |   TABLE ACCESS FULL| CUSTOMER1  |    7 |  476 |   3  (0)  | 00:00:01|
|*  3 |   TABLE ACCESS FULL| ORDER1     |   10 |  590 |   1  (0)  | 00:00:01|
------------------------------------------------------------------------------

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

   3 - filter("C"."CUSTID"="O"."CUSTID")

Note
     - dynamic sampling used for this statement (level=2)

19 rows selected.
```

Question: Draw the query tree based on the above execution plan.

Answer: Write your answers here

6. Ignoring Primary Key

If you want to tell the system NOT TO use the PK, you can use the NO_INDEX hint. When the PK is ignored (is not used), this implies that the index is not used either. Therefore, the query will use the HASH JOIN method.

```
SQL> Explain Plan For
   2 Select /*+ NO_INDEX (C CUST_ID_PK) */ *
   3 From Customer1 C, Order1 O
   4 Where C.CustID = O.CustID;

Explained.

SQL> Select * From Table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 3340424740

------------------------------------------------------------------------------
```

```
| Id  | Operation           | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |    10 |  1270 |      7  (15)| 00:00:01 |
|*  1 |  HASH JOIN          |           |    10 |  1270 |      7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| CUSTOMER1 |     7 |   476 |      3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| ORDER1    |    10 |   590 |      3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("C"."CUSTID"="O"."CUSTID")

Note
-----
   - dynamic sampling used for this statement (level=2)

19 rows selected.
```

Alternatively, you can use the USE_HASH hint in order to enforce the system to use the HASH JOIN method. The result is the same as above.

```
SQL> Explain Plan For
   2 Select /*+ USE_HASH (C O) */ *
   3 From Customer1 C, Order1 O
   4 Where C.CustID = O.CustID;

Explained.

SQL> Select * From Table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
Plan hash value: 3340424740

-------------------------------------------------------------------------------
| Id  | Operation           | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |    10 |  1270 |      7  (15)| 00:00:01 |
|*  1 |  HASH JOIN          |           |    10 |  1270 |      7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| CUSTOMER1 |     7 |   476 |      3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| ORDER1    |    10 |   590 |      3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("C"."CUSTID"="O"."CUSTID")

Note
-----
   - dynamic sampling used for this statement (level=2)

19 rows selected.
```

7. Drop PK, Create Index

Now, let's drop the PK and create an index on the CUSTID attribute, instead:

```
Alter Table Customer1
Drop Constraint Cust_ID_PK;
```

```
CREATE INDEX Cust_ID_IDX
ON Customer1 (CustID);
```

We would like to proof whether having an Index will give the same impact as having a Primary Key. Now, let's do the same join query between Customer1 and Order1:

```
SQL> Explain Plan For
   2 Select *
   3 From Customer1 C, Order1 O
   4 Where C.CustID = O.CustID;

Explained.

SQL> Select * From Table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------
Plan hash value: 386756218

-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |              |    10 |  1270 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                   |              |    10 |  1270 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID | CUSTOMER1    |     7 |   476 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | CUST_ID_IDX  |     7 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |              |    10 |   590 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL          | ORDER1       |    10 |   590 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

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

   4 - access("C"."CUSTID"="O"."CUSTID")
       filter("C"."CUSTID"="O"."CUSTID")

Note
-----
   - dynamic sampling used for this statement (level=2)

22 rows selected.
```

The execution plan shows that the index is used, and therefore, a SORT-MERGE join operation is used. This execution plan is exactly the same as if it had PK. This proves that an Index on CustID gives the same impact as having CustID as PK.

Question: Draw the query tree based on the above execution plan.

Answer: Write your answer here

8. Drop PK, Ignore Index

We can ask the system to ignore the index, by using the NO_INDEX hint. Because the index is ignored, the execution plan uses a HASH JOIN instead.

```
SQL> Explain Plan For
   2 Select /*+ NO_INDEX (C CUST_ID_IDX) */ *
   3 From Customer1 C, Order1 O
   4 Where C.CustID = O.CustID;

Explained.

SQL> Select * From Table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 3340424740

------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    10 |  1270 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN         |           |    10 |  1270 |     7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| CUSTOMER1 |     7 |   476 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| ORDER1    |    10 |   590 |     3   (0)| 00:00:01 |
------------------------------------------------------------------------------

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

   1 - access("C"."CUSTID"="O"."CUSTID")

Note
-----
   - dynamic sampling used for this statement (level=2)

19 rows selected.
```

9. Table Ordering in the FROM Clause

Normally when we do a query SELECT FROM, the tables listed in the FROM clause has no importance. For example, SELECT * FROM TableA, TableB; does not mean that TableA will be executed first and then TableB. However, there is a hint called ORDERED which will enforce the system to use the order of the tables in the FROM clause. For example:

```
SQL> Explain Plan For
   2 Select /*+ ORDERED */ *
   3 From Order1 O, Customer1 C
   4 Where O.CustID = C.CustID;

Explained.

SQL> Select * From Table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 3408291328
```

```
-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    10 |   780 |     7  (15)| 00:00:01 |
|*  1 |  HASH JOIN         |           |    10 |   780 |     7  (15)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| ORDER1    |    10 |   290 |     3   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| CUSTOMER1 |     7 |   343 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - access("O"."CUSTID"="C"."CUSTID")

15 rows selected.
```

The ORDERED hint asks the system to obey the order of the tables in the FROM clause. In the above example, table Order1 is more important than table Customer1, because in the FROM clause, table Order1 is listed first. Because table Order1 has no index on CustID, the execution plan will use the HASH JOIN method, although table Customer1 has an index. The reason is because table Order1 is more important than table Customer1.

10. Foreign Key

If you try the following:

```
Select *
From Customer1 C, Order1 O
Where C.CustID = O.CustID;
```

You will get a SORT-MERGE with INDEX, as CustID in Customer1 is indexed (or PK). But CustID in Order1 is not set as a FOREIGN KEY.

Now let's set CustID in both tables properly; that is CustID in Customer1 is PK, and CustID in Order is FK.

```
Alter Table Customer1
Add Constraint Cust_ID_PK PRIMARY KEY (CustID);
```

```
Alter Table Order1
Add Constraint Cust_ID_FK FOREIGN KEY (CustID)
REFERENCES Customer1(CustID);
```

And then try:

```
SQL> Explain Plan For
   2 Select *
   3 From Customer1 C, Order1 O
   4 Where C.CustID = O.CustID;

Explained.
```

```
SQL> Select * From Table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 386756218

-----------------------------------------------------------------------------------------------
| Id  | Operation                     | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |             |    10 |   780 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                   |             |    10 |   780 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| CUSTOMER1   |     7 |   343 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | CUST_ID_IDX |     7 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |             |    10 |   290 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL          | ORDER1      |    10 |   290 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------

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

   4 - access("C"."CUSTID"="O"."CUSTID")
       filter("C"."CUSTID"="O"."CUSTID")

18 rows selected.
```

The execution plan shows that the FK does not give any impact at all. The execution plan still uses a SORT-MERGE JOIN, but table Order1 still needs to be sorted. This shows that FK is not implemented by an index, whereas PK is.

Lets replace the FK with an index, and then try the same join query. Now, we have two indices: one is the PK of Customer1 which is CustID, and two is the index of CustID of Order1. Therefore, the join query is based on two indexed attributes: CustID of Customer1 and of Order1.

```
Alter Table Order1
Drop Constraint Cust_ID_FK;
```

```
Create Index Cust_ID_FK_IDX on Order1(CustID);
```

```
SQL> Explain Plan For
   2 Select *
   3 From Customer1 C, Order1 O
   4 Where C.CustID = O.CustID;

Explained.

SQL> Select * From Table(dbms_xplan.display);
```

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------
Plan hash value: 194852167
-------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name           | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                |    10 |   780 |     6  (17)| 00:00:01 |
|   1 |  MERGE JOIN                   |                |    10 |   780 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| ORDER1         |    10 |   290 |     2   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN            | CUST_ID_FK_IDX |    10 |       |     1   (0)| 00:00:01 |
|*  4 |   SORT JOIN                   |                |     7 |   343 |     4  (25)| 00:00:01 |
|   5 |    TABLE ACCESS FULL          | CUSTOMER1      |     7 |   343 |     3   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

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

   4 - access("C"."CUSTID"="O"."CUSTID")
       filter("C"."CUSTID"="O"."CUSTID")

18 rows selected.
```

The execution plan above shows that even when the query has two indices, only one index will be used in the query processing. In the above example, only the index on CustID of table Order1 is used, whereas the PK of Customer1 is not used at all. So, only one index is used at a time. The decision on which index is used is based on the larger table, which in this case is table Order1.

Question: Draw the query tree based on the above execution plan.

Answer: Write your answer here

The following queries will produce the same execution plan as the above, where the FK index is used, and the PK index is not used.

```
Select /*+ ORDERED */ *
From Order1 O, Customer1 C
Where O.CustID = C.CustID;
```

```
Select /*+ NO_INDEX (C(CustID)) */ *
From Customer1 C, Order1 O
Where C.CustID = O.CustID;
```

END OF TUTORIAL