

# Explain Plan


Mag. Thomas Griesmayer

# Definition

- The `EXPLAIN PLAN` statement displays execution plans chosen by the Oracle optimizer for `SELECT`, `UPDATE`, `INSERT`, and `DELETE` statements.
- A statement's execution plan is the sequence of operations Oracle performs to run the statement.

$$\begin{array}{r} ( \underbrace{(5-3)}_2 * \underbrace{(2+1)}_3 ) - 10 \\ \hline 6 \\ \hline -4 \end{array}$$

parallel  


sequential  


# Content

- The row source tree is the core of the execution plan. It shows the following information:
  - An ordering of the tables referenced by the statement
  - An access method for each table mentioned in the statement
  - A join method for tables affected by join operations in the statement
  - Data operations like filter, sort, or aggregation
- In addition to the row source tree, the plan table contains information about the following:
  - Optimization, such as the cost and cardinality of each operation
  - Partitioning, such as the set of accessed partitions
  - Parallel execution, such as the distribution method of join inputs

# Statistics

- Optimizer statistics are automatically gathered by automatic optimizer statistics collection, which gathers statistics on all objects in the database which have stale or missing statistics.
- Automatic optimizer statistics collection is enabled by default to run in all predefined maintenance windows.

```
ANALYZE TABLE CUSTOMER_INDEX COMPUTE STATISTICS;
```

# Reason

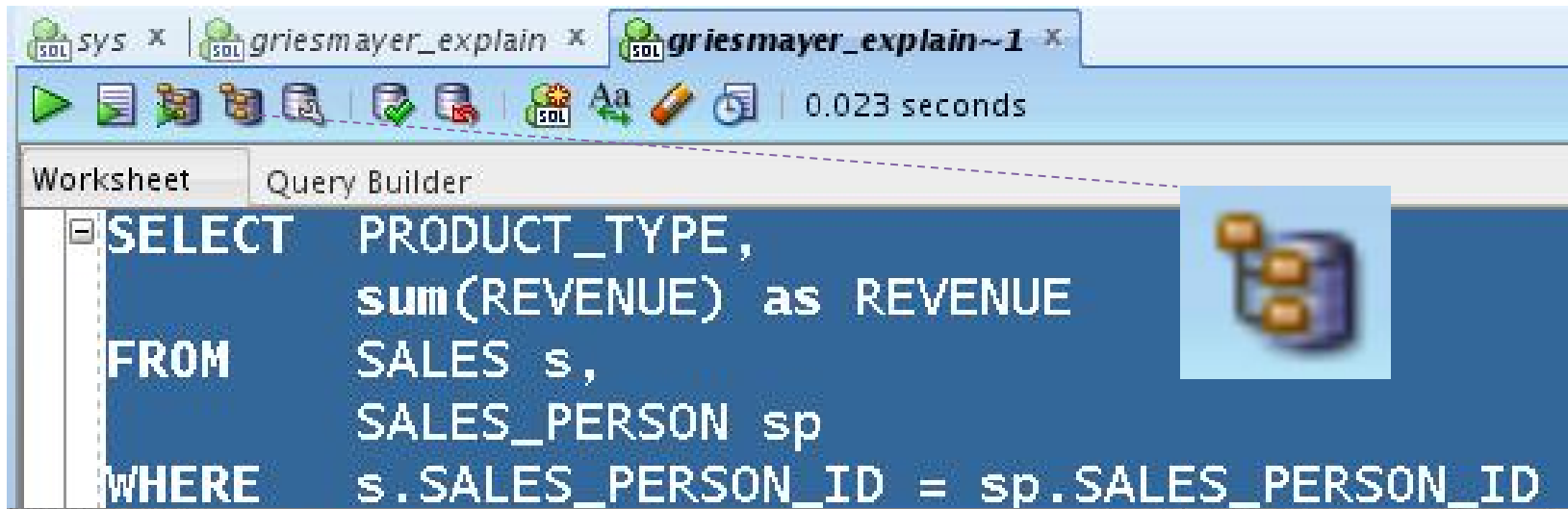
- The EXPLAIN PLAN results let you determine whether the optimizer selects a particular execution plan, such as, nested loops join.
- It also helps you to understand the performance of a query.
- With the cost-based optimizer, execution plans can and do change as the underlying costs change.

```
SELECT Date,  
       SUM(Revenue)  
WHERE Product_Type = 'B'
```

- 1
  - 2
  - 3
- ```
ORDER BY Date;
```

| RID | Date  | Type | Rev |
|-----|-------|------|-----|
| 14  | 1.1.  | B    | 5.0 |
| 23  | 12.1. | B    | 3.2 |
| A3  | 20.1. | B    | 6.3 |
| A7  | 1.1.  | A    | 2.9 |
| B3  | 12.1. | B    | 2.9 |

# SQL Developer



```
SELECT PRODUCT_TYPE,
       sum(REVENUE) as REVENUE
FROM   SALES s,
       SALES_PERSON sp
WHERE  s.SALES_PERSON_ID = sp.SALES_PERSON_ID
```

| OPERATION                            | OBJECT_NAME  | OPTIONS  | COST |
|--------------------------------------|--------------|----------|------|
| SELECT STATEMENT                     |              |          | 318  |
| HASH                                 |              | GROUP BY | 318  |
| HASH JOIN                            |              |          | 313  |
| Access Predicates                    |              |          |      |
| S.SALES_PERSON_ID=SP.SALES_PERSON_ID |              |          |      |
| TABLE ACCESS                         | SALES_PERSON | FULL     | 3    |
| TABLE ACCESS                         | SALES        | FULL     | 309  |

# FULL TABLE SCAN

- Reads all rows from a table and filters out those that do not meet the selection criteria (WHERE).
- When Oracle Database performs a full table scan, the blocks are read sequentially.
- The database reads each block only once.

The screenshot shows the Oracle SQL Developer interface. The top toolbar indicates a query execution time of 0.055 seconds. The main window displays the following SQL query:

```
SELECT count(*)  
FROM CUSTOMER_NOINDEX  
WHERE FIRST_NAME = 'Thomas';
```

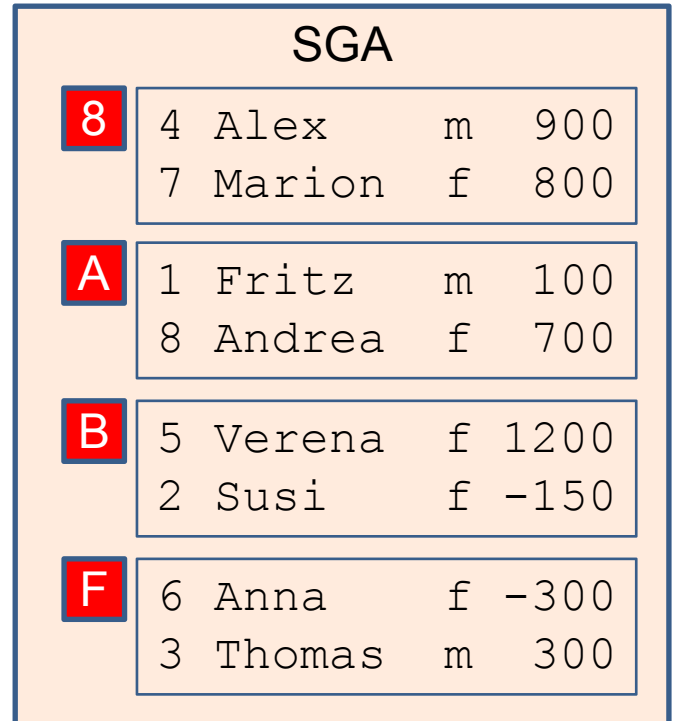
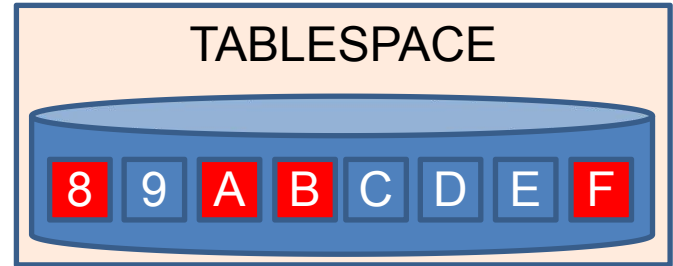
Below the query, the 'Script Output' and 'Explain Plan' tabs are visible. The 'Explain Plan' tab shows the execution plan for the query, with a total execution time of 0.055 seconds. The plan consists of the following operations:

| OPERATION           | OBJECT_NAME      | OPTIONS   | COST |
|---------------------|------------------|-----------|------|
| SELECT STATEMENT    |                  |           | 4401 |
| SORT                |                  | AGGREGATE |      |
| TABLE ACCESS        | CUSTOMER_NOINDEX | FULL      | 4401 |
| Filter Predicates   |                  |           |      |
| FIRST_NAME='Thomas' |                  |           |      |

In the execution plan, the 'FULL' option for the 'TABLE ACCESS' operation and the 'Filter Predicates' section are highlighted with red boxes.

# FULL TABLE SCAN

| ROWID | C_ID | NAME   | GENDER | BALANCE |
|-------|------|--------|--------|---------|
| A0    | 1    | Fritz  | m      | 100     |
| B1    | 2    | Susi   | f      | -150    |
| F1    | 3    | Thomas | m      | 300     |
| 80    | 4    | Alex   | m      | 900     |
| B0    | 5    | Verena | f      | 1200    |
| F0    | 6    | Anna   | f      | -300    |
| 81    | 7    | Marion | f      | 800     |
| A1    | 8    | Andrea | f      | 700     |





# ROW ID SCAN

- To access a table by ROWID, Oracle Database first obtains the ROWIDs of the selected rows, either from the statement's WHERE clause or through an index scan of one or more of the table's indexes.
- Oracle Database then locates each selected row in the table based on its ROWID.

Worksheet | Query Builder

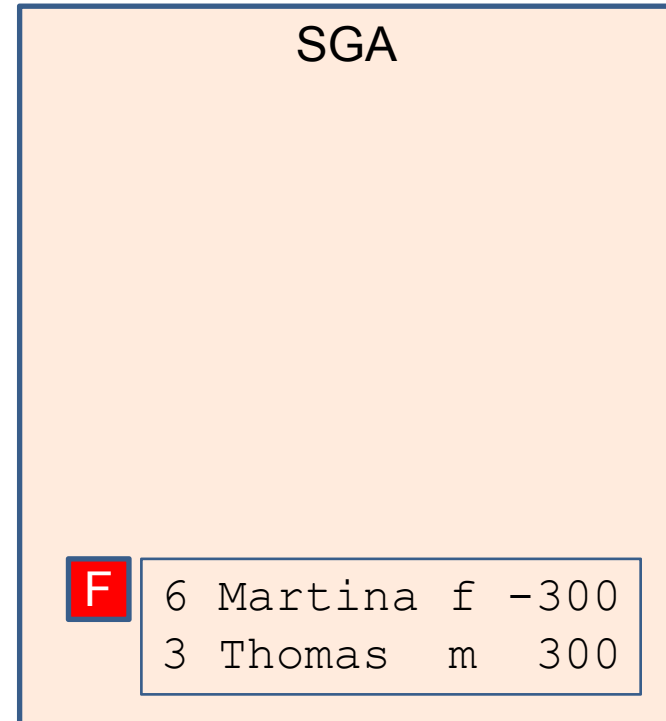
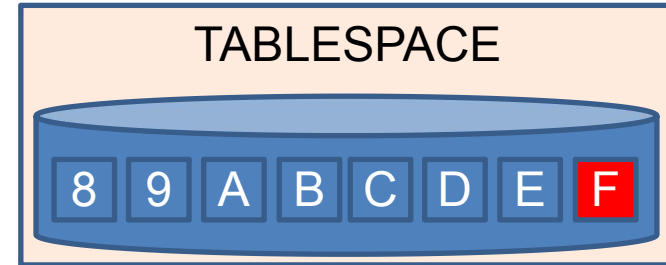
```
SELECT *  
FROM CUSTOMER_INDEX  
WHERE CUSTOMER_ID = 1;
```

Script Output | Query Result | Explain Plan

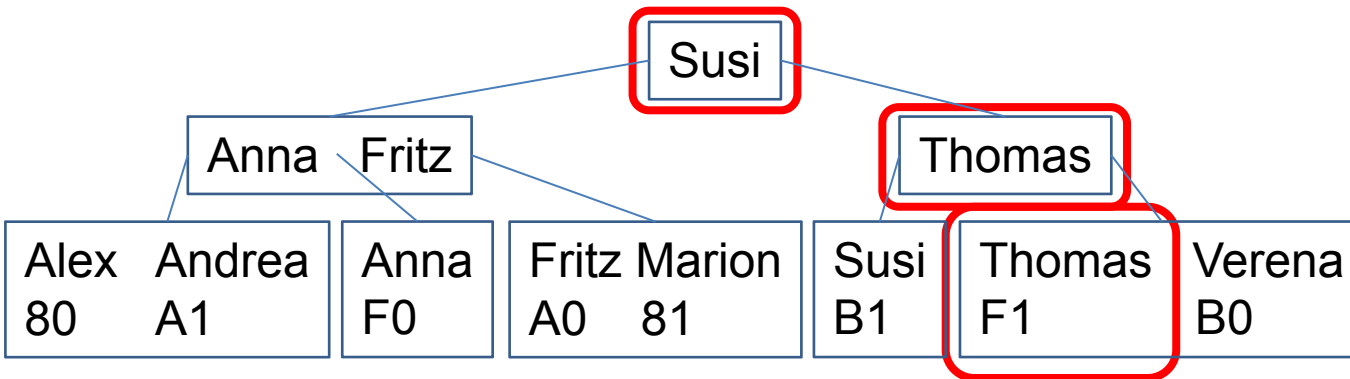
SQL | 0 seconds

| OPERATION         | OBJECT_NAME    | OPTIONS        | COST |
|-------------------|----------------|----------------|------|
| SELECT STATEMENT  |                |                | 2    |
| TABLE ACCESS      | CUSTOMER_INDEX | BY INDEX ROWID | 2    |
| INDEX             | SYS_C0033781   | UNIQUE SCAN    | 1    |
| Access Predicates |                |                |      |
| CUSTOMER_ID=1     |                |                |      |

# ROW ID SCAN

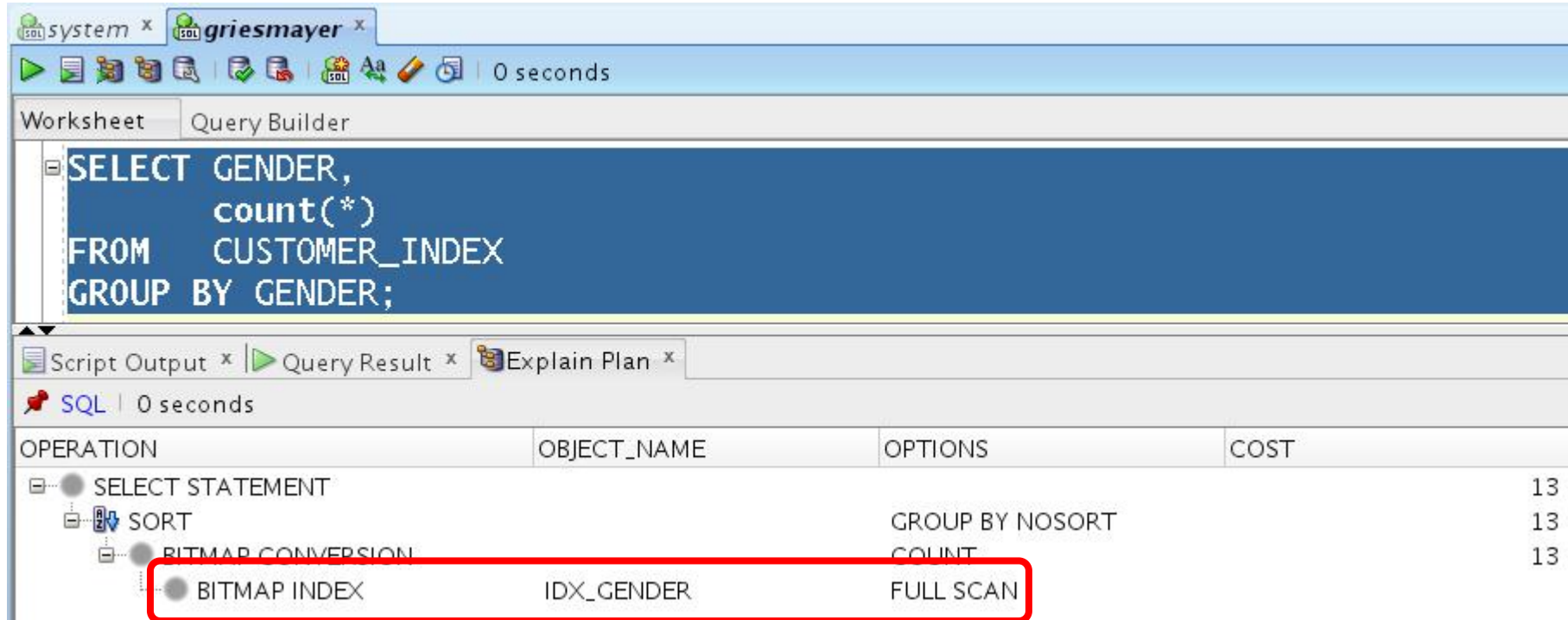


| ROWID | C_ID | NAME   | GENDER | BALANCE |
|-------|------|--------|--------|---------|
| A0    | 1    | Fritz  | m      | 100     |
| B1    | 2    | Susi   | f      | -150    |
| F1    | 3    | Thomas | m      | 300     |
| 80    | 4    | Alex   | m      | 900     |
| B0    | 5    | Verena | f      | 1200    |
| F0    | 6    | Anna   | f      | -300    |
| 81    | 7    | Marion | f      | 800     |
| A1    | 8    | Andrea | f      | 700     |



# INDEX SCAN

- If the statement accesses only columns of the index, then Oracle Database reads the indexed column values directly from the index, rather than from the table.



The screenshot shows the Oracle SQL Developer interface. The top window displays the following SQL query:

```
SELECT GENDER,  
       count(*)  
FROM   CUSTOMER_INDEX  
GROUP BY GENDER;
```

The bottom window shows the Explain Plan for the query, with the following columns: OPERATION, OBJECT\_NAME, OPTIONS, and COST. The plan is as follows:

| OPERATION         | OBJECT_NAME | OPTIONS         | COST |
|-------------------|-------------|-----------------|------|
| SELECT STATEMENT  |             |                 | 13   |
| SORT              |             | GROUP BY NOSORT | 13   |
| BITMAP CONVERSION |             | COUNT           | 13   |
| BITMAP INDEX      | IDX_GENDER  | FULL SCAN       |      |

The 'BITMAP INDEX' row is highlighted with a red rectangle, indicating that the query is using a full scan of the index.

```

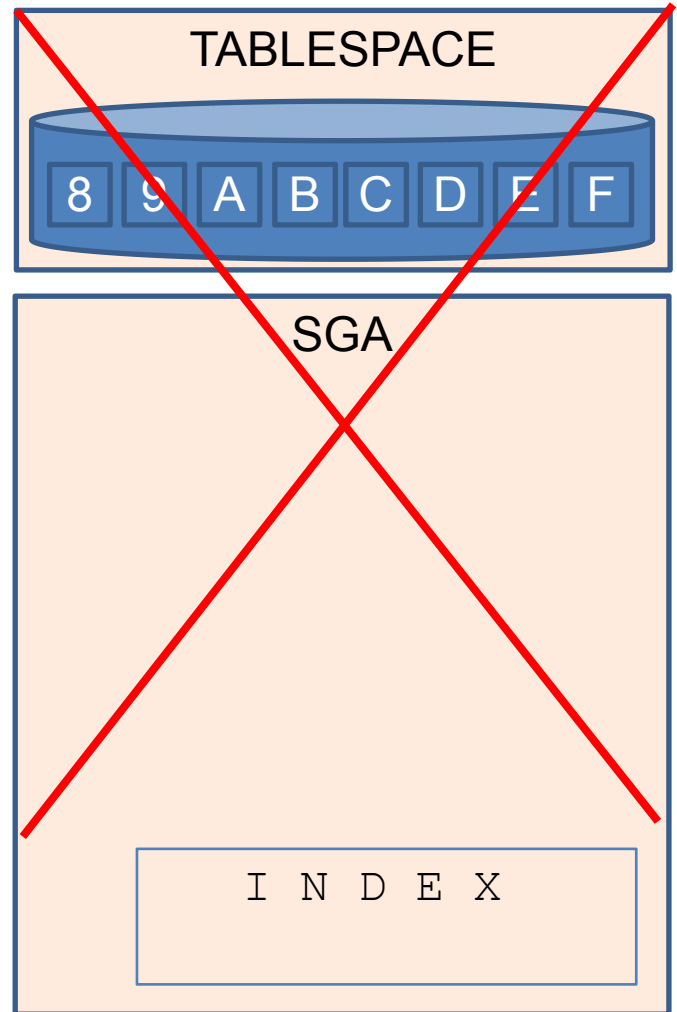
SELECT GENDER,
       COUNT (*)
FROM   CUSTOMER_INDEX
GROUP BY GENDER

```

|          | 80 | 81 | A0 | A1 | B0 | B1 | F0 | F1 |
|----------|----|----|----|----|----|----|----|----|
| <b>m</b> | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 1  |
| <b>f</b> | 0  | 1  | 0  | 1  | 1  | 1  | 1  | 0  |

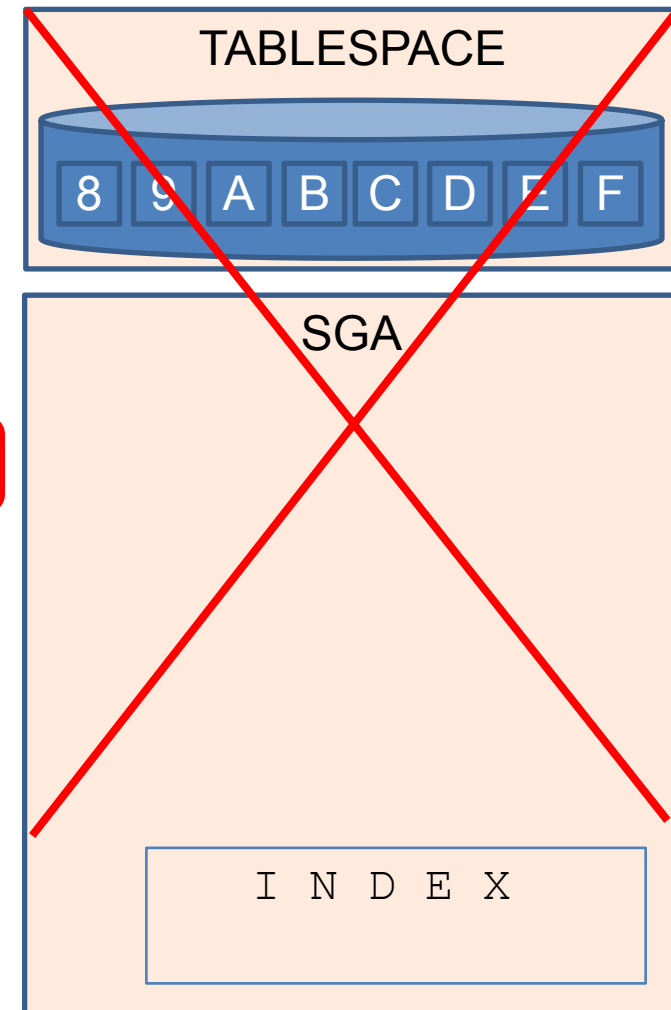
|   |
|---|
| 3 |
| 5 |

# INDEX SCAN



```
SELECT COUNT(DISTINCT NAME)
FROM CUSTOMER_INDEX
```

# INDEX SCAN



Susi

Anna Fritz

Thomas

|      |        |      |       |        |      |        |        |
|------|--------|------|-------|--------|------|--------|--------|
| Alex | Andrea | Anna | Fritz | Marion | Susi | Thomas | Verena |
| 80   | A1     | F0   | A0    | 81     | B1   | F1     | B0     |

# INDEX RANGE SCAN

- Data is returned in the ascending order of index columns.
- Multiple rows with identical values are sorted in ascending order by ROWID. If an index can satisfy an ORDER BY clause, then the optimizer uses this option and avoids a sort.

# INDEX RANGE SCAN

The screenshot shows a SQL IDE window with the following components:

- Worksheet:** Contains the SQL query:

```
SELECT *  
FROM CUSTOMER_INDEX  
WHERE FIRST_NAME BETWEEN 'Anna' and 'Anja'  
ORDER BY FIRST_NAME;
```
- Query Result / Explain Plan:** Shows the execution plan for the query. The plan is as follows:

| OPERATION            | OBJECT_NAME    | OPTIONS        | COST |
|----------------------|----------------|----------------|------|
| SELECT STATEMENT     |                |                | 0    |
| FILTER               |                |                |      |
| Filter Predicates    |                |                |      |
| NULL IS NOT NULL     |                |                |      |
| TABLE ACCESS         | CUSTOMER_INDEX | BY INDEX ROWID | 0    |
| INDEX                | IDX_FIRST_NAME | RANGE SCAN     | 0    |
| Access Predicates    |                |                |      |
| AND                  |                |                |      |
| FIRST_NAME >= 'Anna' |                |                |      |
| FIRST_NAME <= 'Anja' |                |                |      |

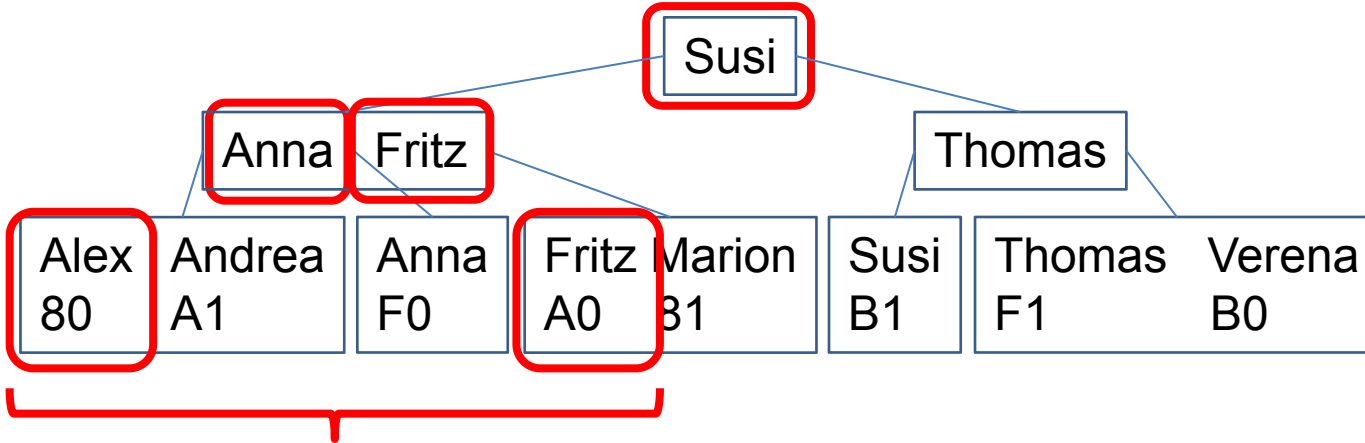
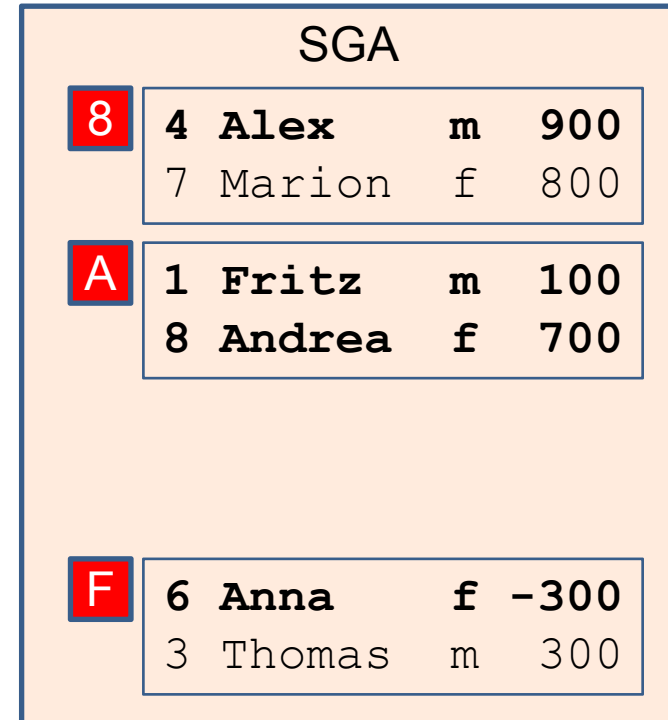
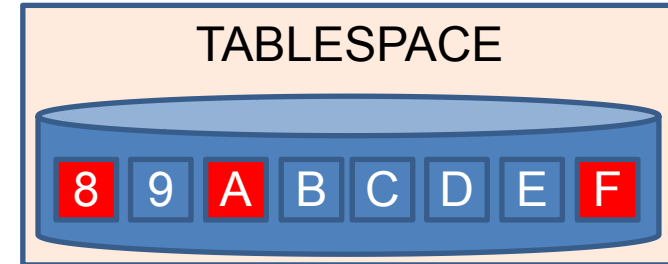
The execution plan is partially highlighted with a red box, encompassing the TABLE ACCESS, INDEX, and Access Predicates sections.

```

SELECT *
FROM   CUSTOMER_INDEX
WHERE  FIRST_NAME BETWEEN
      'Alex' and 'Fritz'
ORDER BY FIRST_NAME;

```

# INDEX RANGE SCAN





# NESTED LOOP JOIN

- The optimizer determines the driving table and designates it as the outer table.
- The other table is designated as the inner table.
- For every row in the outer table, Oracle Database accesses all the rows in the inner table. The outer loop is for every row in the outer table and the inner loop is for every row in the inner table.

# NESTED LOOP JOIN

system x griesmayer x

0.02 seconds

Worksheet Query Builder

```
SELECT *  
FROM CUSTOMER_INDEX cus  
INNER JOIN  
FIRSTNAME nam  
ON cus.FIRST_NAME = nam.FIRST_NAME
```

Script Output x Query Result x Explain Plan x

SQL | 0.02 seconds

| OPERATION         | OBJECT_NAME    | OPTIONS                       | COST |
|-------------------|----------------|-------------------------------|------|
| SELECT STATEMENT  |                |                               | 3    |
| NESTED LOOPS      |                |                               | 3    |
| NESTED LOOPS      |                |                               | 3    |
| TABLE ACCESS      | FIRSTNAME      | FULL                          | 3    |
| INDEX             | IDX_FIRST_NAME | RANGE SCAN                    | 0    |
| Access Predicates |                | CUS.FIRST_NAME=NAM.FIRST_NAME |      |
| TABLE ACCESS      | CUSTOMER_INDEX | BY INDEX ROWID                | 0    |

# NESTED LOOP JOIN

```
FOR EACH CUSTOMER
  FOR EACH FIRSTNAME
    if CUSTOMER.NAME = FIRSTNAME.NAME
      add to result
```

| ROWID | C_ID | NAME   | BALANCE |
|-------|------|--------|---------|
| A0    | 1    | Fritz  | 100     |
| B1    | 2    | Susi   | -150    |
| F1    | 3    | Thomas | 300     |
| B0    | 5    | Verena | 1200    |
| F0    | 6    | Anna   | -300    |
| 81    | 7    | Marion | 800     |
| A1    | 8    | Andrea | 700     |

| ROWID | NAME   | GENDER |
|-------|--------|--------|
| 01    | Alex   | male   |
| 13    | Clara  | female |
| 21    | Marion | female |
| 00    | Andrea | female |
| 20    | Makrus | male   |
| 03    | Susi   | female |
| 02    | Thomas | male   |
| 10    | Anna   | female |
| 11    | Verena | female |
| 22    | Maria  | female |
| 12    | Fritz  | male   |

# CARTESIAN JOIN

- The database uses a Cartesian join when one or more of the tables does not have any join conditions to any other tables in the statement.
- The optimizer joins every row from one data source with every row from the other data source, creating the Cartesian product of the two sets.

# CARTESIAN JOIN

The screenshot shows a database query tool interface. The top part displays the SQL query:

```
SELECT *  
FROM CUSTOMER_INDEX cus  
INNER JOIN  
FIRSTNAME nam  
ON 1=1
```

The bottom part shows the execution plan for the query. The plan is as follows:

| OPERATION        | OBJECT_NAME    | OPTIONS   | COST   |
|------------------|----------------|-----------|--------|
| SELECT STATEMENT |                |           | 419764 |
| MERGE JOIN       |                | CARTESIAN | 419764 |
| TABLE ACCESS     | CUSTOMER_INDEX | FULL      | 4400   |
| BUFFER           |                | SORT      | 415364 |
| TABLE ACCESS     | FIRSTNAME      | FULL      | 1      |

The 'MERGE JOIN' row is highlighted with a red box, indicating the Cartesian join operation.

```

FOR EACH CUSTOMER
  FOR EACH FIRSTNAME
    add to temp
FOR EACH temp
  if (temp.NAME1 = temp.NAME2)
    add to result

```

| ROWID | C_ID | NAME   | BALANCE |
|-------|------|--------|---------|
| A0    | 1    | Fritz  | 100     |
| B1    | 2    | Susi   | -150    |
| F1    | 3    | Thomas | 300     |

| ROWID | NAME   | GENDER |
|-------|--------|--------|
| 00    | Andrea | female |
| 20    | Makrus | male   |
| 03    | Susi   | female |
| 12    | Fritz  | male   |

## CARTESIAN JOIN

| C_ID | NAME1  | BAL  | NAME2  | GEN    |
|------|--------|------|--------|--------|
| 1    | Fritz  | 100  | Andrea | female |
| 1    | Fritz  | 100  | Markus | male   |
| 1    | Fritz  | 100  | Susi   | female |
| 1    | Fritz  | 100  | Fritz  | male   |
| 2    | Susi   | -150 | Andrea | female |
| 2    | Susi   | -150 | Markus | male   |
| 2    | Susi   | -150 | Susi   | female |
| 2    | Susi   | -150 | Fritz  | male   |
| 3    | Thomas | 300  | Andrea | female |
| 3    | Thomas | 300  | Markus | male   |
| 3    | Thomas | 300  | Susi   | female |
| 3    | Thomas | 300  | Fritz  | male   |

# INDEX JOIN

- An index join is a hash join of several indexes that together contain all the table columns referenced in the query.
- If the database uses an index join, then table access is not needed because the database can retrieve all the relevant column values from the indexes.

# INDEX JOIN

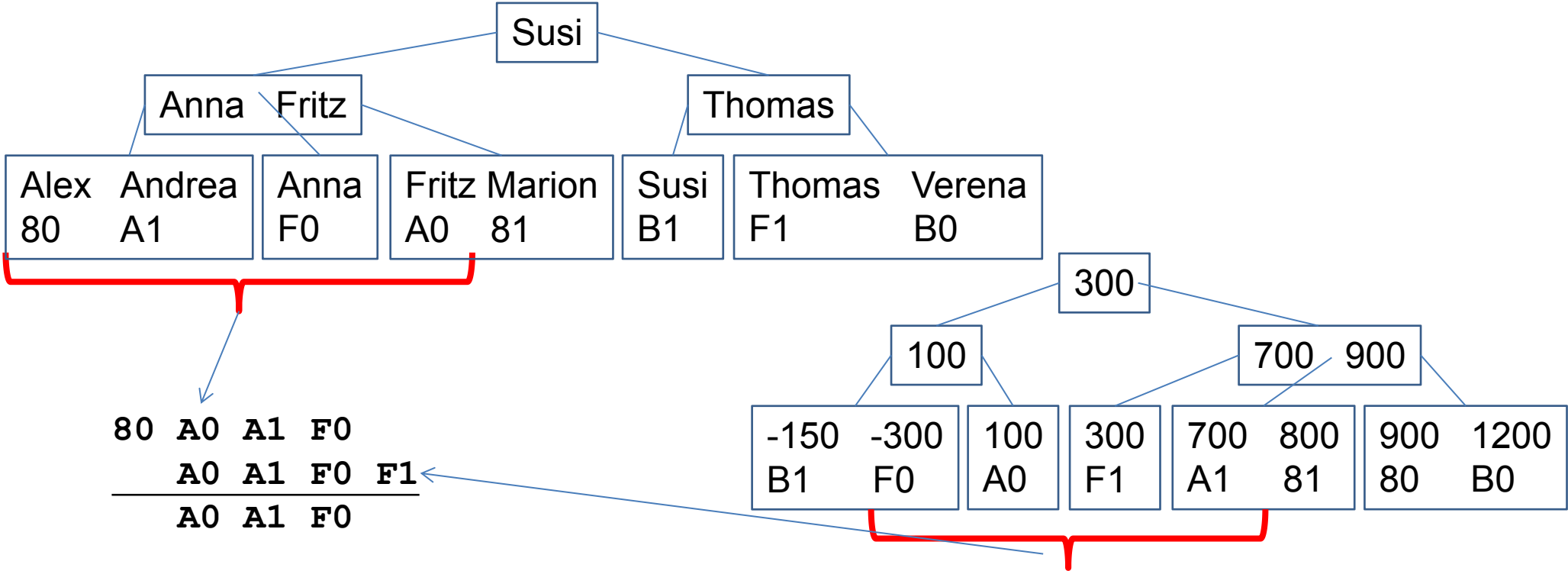
```
SELECT FIRST_NAME,
       BALANCE
FROM   CUSTOMER_INDEX cus
WHERE  cus.BALANCE between 100.00 and 110.00 and
       cus.FIRST_NAME between 'Anna' and 'Anja'
```

| OPERATION         | OBJECT_NAME        | OPTIONS    | COST |
|-------------------|--------------------|------------|------|
| SELECT STATEMENT  |                    |            | 0    |
| FILTER            |                    |            |      |
| Filter Predicates |                    |            |      |
| VIEW              | index\$_join\$_001 |            | 726  |
| Filter Predicates |                    |            |      |
| HASH JOIN         |                    |            |      |
| Access Predicates |                    |            |      |
| ROWD=ROWD         |                    |            |      |
| INDEX             | IDX_BALANCE        | RANGE SCAN | 5    |
| INDEX             | IDX_FIRST_NAME     | RANGE SCAN | 2489 |



# INDEX JOIN

```
SELECT FIRST_NAME,  
       BALANCE  
FROM   CUSTOMER_INDEX  
WHERE  FIRST_NAME BETWEEN 'Alex' and 'Fritz' and  
       BALANCE between -300.00 and 750.00
```



# HASH JOIN

- The database uses hash joins to join large data sets.
- The optimizer uses the smaller of two tables or data sources to build a hash table on the join key in memory. It then scans the larger table, probing the hash table to find the joined rows.
- This method is best when the smaller table fits in available memory.

# HASH JOIN

```
SELECT *  
FROM CUSTOMER cus  
INNER JOIN  
FIRSTNAME nam  
ON cus.FIRST_NAME = nam.FIRST_NAME
```

| OPERATION                     | OBJECT_NAME | OPTIONS | COST |
|-------------------------------|-------------|---------|------|
| SELECT STATEMENT              |             |         | 4404 |
| HASH JOIN                     |             |         | 4404 |
| Access Predicates             |             |         |      |
| CUS.FIRST_NAME=NAM.FIRST_NAME |             |         |      |
| TABLE ACCESS                  | FIRSTNAME   | FULL    | 3    |
| TABLE ACCESS                  | CUSTOMER    | FULL    | 4399 |

| RID | C_ID | NAME   | BAL  |
|-----|------|--------|------|
| A0  | 1    | Fritz  | 100  |
| B1  | 2    | Susi   | -150 |
| F1  | 3    | Thomas | 300  |
| B0  | 5    | Verena | 1200 |
| F0  | 6    | Anna   | -300 |
| 81  | 7    | Marion | 800  |
| A1  | 8    | Andrea | 700  |

| RID | NAME   | GENDER |
|-----|--------|--------|
| 01  | Alex   | male   |
| 20  | Makrus | male   |
| 03  | Susi   | female |
| 02  | Thomas | male   |
| 10  | Anna   | female |
| 11  | Verena | female |
| 12  | Fritz  | male   |

## HASH JOIN

|           |           |          |               |               |
|-----------|-----------|----------|---------------|---------------|
| <b>F1</b> | <b>02</b> | <b>3</b> | <b>Thomas</b> | <b>male</b>   |
| A0        | 01        |          |               |               |
| A0        | 10        |          |               |               |
| <b>A0</b> | <b>12</b> | <b>1</b> | <b>Fritz</b>  | <b>male</b>   |
| A1        | 01        |          |               |               |
| A1        | 10        |          |               |               |
| A1        | 12        |          |               |               |
| F0        | 01        |          |               |               |
| <b>F0</b> | <b>10</b> | <b>6</b> | <b>Anna</b>   | <b>female</b> |
| F0        | 12        |          |               |               |
| <b>B0</b> | <b>11</b> | <b>5</b> | <b>Verena</b> | <b>female</b> |
| 81        | 20        |          |               |               |
| <b>B1</b> | <b>03</b> | <b>2</b> | <b>Susi</b>   | <b>female</b> |

| HASH | RID | RID | RID |
|------|-----|-----|-----|
| 0    | F1  |     |     |
| 1    | A0  | A1  | F0  |
| 2    | B0  |     |     |
| 3    | 81  |     |     |
| 4    | B1  |     |     |

| HASH | RID | RID | RID |
|------|-----|-----|-----|
| 0    | 02  |     |     |
| 1    | 01  | 10  | 12  |
| 2    | 11  |     |     |
| 3    | 20  |     |     |
| 4    | 03  |     |     |

# SORT MERGE JOIN

- In a merge join, there is no concept of a driving table.
- The join consists of two steps:
  - Sort join operation - both the inputs are sorted on the join key.
  - Merge join operation - the sorted lists are merged together.

# SORT MERGE JOIN

```
SELECT *
FROM   CUSTOMER_INDEX cus
       INNER JOIN
       FIRSTNAME_INDEX nam
       ON  cus.FIRST_NAME = nam.FIRST_NAME
ORDER BY cus.FIRST_NAME;
```

| OPERATION                     | OBJECT_NAME       | OPTIONS        | COST  |
|-------------------------------|-------------------|----------------|-------|
| SELECT STATEMENT              |                   |                | 27182 |
| MERGE JOIN                    |                   |                | 27182 |
| TABLE ACCESS                  | FIRSTNAME_INDEX   | BY INDEX ROWID | 2     |
| INDEX                         | IDX_FIRSTNAME_REF | FULL SCAN      | 1     |
| SORT                          |                   | JOIN           | 27180 |
| Access Predicates             |                   |                |       |
| CUS.FIRST_NAME=NAM.FIRST_NAME |                   |                |       |
| Filter Predicates             |                   |                |       |
| CUS.FIRST_NAME=NAM.FIRST_NAME |                   |                |       |
| TABLE ACCESS                  | CUSTOMER_INDEX    | FULL           | 4399  |

| RID | C_ID | NAME   | BAL  |
|-----|------|--------|------|
| A0  | 1    | Fritz  | 100  |
| B1  | 2    | Susi   | -150 |
| F1  | 3    | Thomas | 300  |
| B0  | 5    | Verena | 1200 |
| F0  | 6    | Anna   | -300 |
| 81  | 7    | Marion | 800  |
| A1  | 8    | Andrea | 700  |

# SORT MERGE JOIN

| RID | NAME   | GENDER |
|-----|--------|--------|
| 01  | Alex   | male   |
| 20  | Markus | male   |
| 03  | Susi   | female |
| 02  | Thomas | male   |
| 10  | Anna   | female |
| 11  | Verena | female |
| 12  | Fritz  | male   |

