Oracle SQL tuning
- An Overview

3 P’s

- **Purpose :**
- To learn an overview of Performance Tuning
- **Product :**
- Learn to understand the basics of Tuning
  - Describe what attributes of a SQL statement can make it perform poorly
  - List the Oracle tools that can be used to tune SQL
- **Process:**
- Instructor led training with practical experience

Coverage

- Introduction to SQL tuning
- Describe why the SQL statements are performing poorly
- Introduction to Oracle Optimizer
- Discuss the need for Optimizer
- Explain the various phases of Optimization
- Gather Execution Plans
- Interpret Execution Plans
- Interpret the output of TKPROF
- Gather Optimizer statistics
- Use Hints appropriately

- Reasons for Inefficient SQL Performance
- SQL Tuning tasks
- Proactive tuning Methodology

Reasons for Inefficient SQL Performance

- Stale or missing optimizer statistics
- Missing access structures
- Suboptimal execution plan selection
- Poorly constructed SQL

Inefficient SQL Queries

```
SELECT COUNT(*) FROM products p   WHERE prod_list_price < 1.15
(SELECT avg(unit_cost) FROM costs c
WHERE c.prod_id = p.prod_id)

 SELECT * FROM job_history jh, employees e
 WHERE substr(to_char(e.employee_id),2) =
 substr(to_char(jh.employee_id),2)

 SELECT * FROM orders WHERE order_id_char = 1205

 SELECT * FROM employees
WHERE to_char(salary) = :sal

 SELECT * FROM parts_old
 UNION
 SELECT * FROM parts_new
```

SQL Tuning Tasks: Overview

- Identifying high-load SQL
- Gathering statistics
- Generating system statistics
- Rebuilding existing indexes
- Maintaining execution plans
- Creating new index strategies

Scalability with Application Design, Implementation, and Configuration

- Applications have a significant impact on scalability.
  - Poor schema design can cause expensive SQL that does not scale.
  - Poor transaction design can cause locking and serialization problems.
  - Poor connection management can cause unsatisfactory response times.

Common Mistakes on Customer Systems

1. Bad connection management
2. Bad use of cursors and the shared pool
3. Excess of resources consuming SQL statements
4. Use of nonstandard initialization parameters
5. Poor database disk configuration
6. Redo log setup problems
7. Excessive serialization
8. Inappropriate full table scans
9. Large number of recursive SQL statements related to space management or parsing activity
10. Deployment and migration errors

Proactive Tuning Methodology

- Simple design
- Data modeling
- Tables and indexes
- Using views
- Writing efficient SQL
- Cursor sharing
- Using bind variables

Simplicity in Application Design

- Simple tables
- Well-written SQL
- Indexing only as required
- Retrieving only required information

Table Design

- Compromise between flexibility and performance:
  - Principally normalize
  - Selectively denormalize
- Use Oracle performance and management features:
  - Default values
  - Constraints
  - Materialized views
  - Clusters
  - Partitioning
- Focus on business-critical tables

Index Design

- Create indexes on the following:
  - Primary key (can be automatically created)
  - Unique key (can be automatically created)
  - Foreign keys (good candidates)
- Index data that is frequently queried (select list).
- Use SQL as a guide to index design.

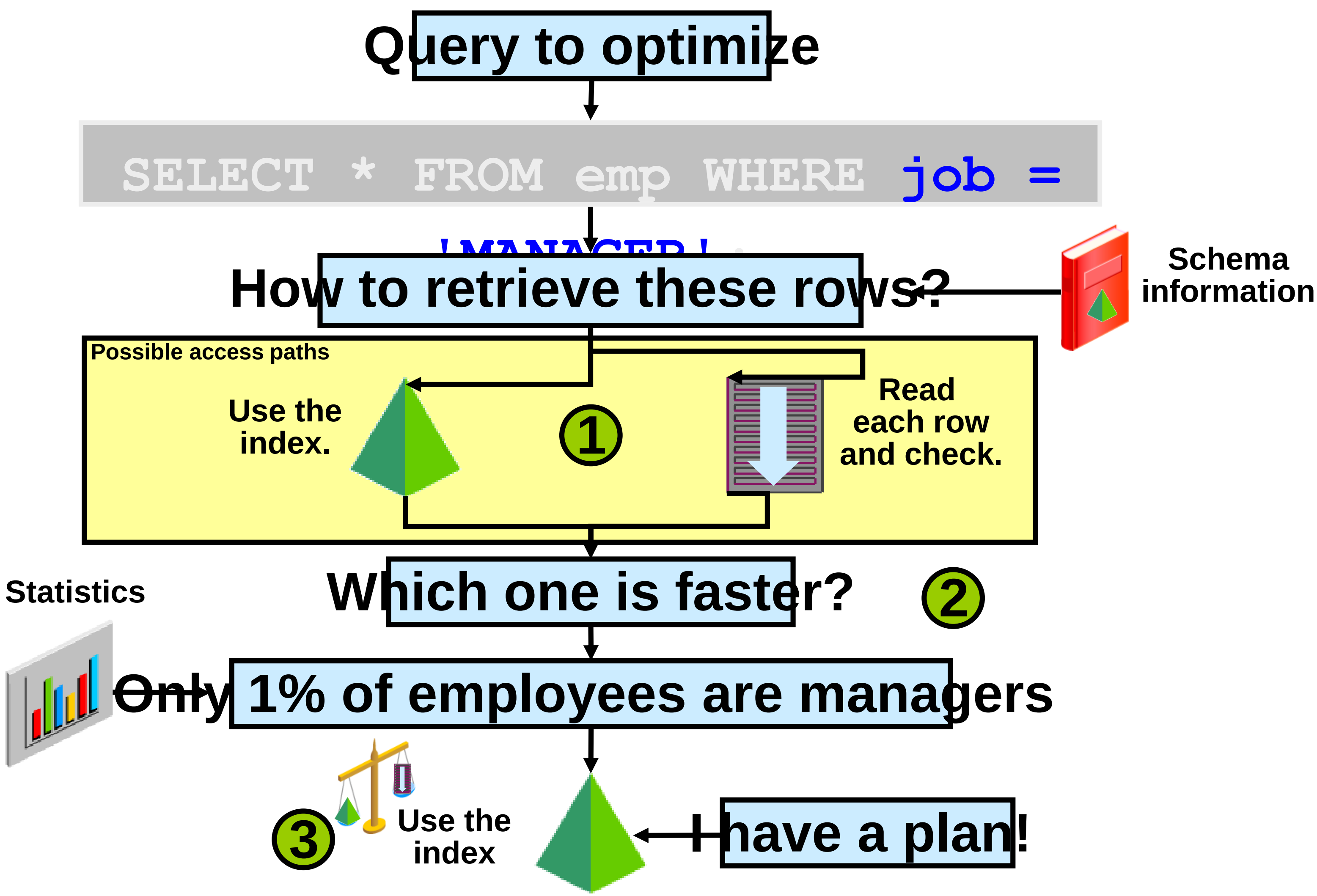
Writing SQL to Share Cursors

- Create generic code using the following:
  - Stored procedures and packages
  - Database triggers
  - Any other library routines and procedures
- Write to format standards (improves readability):
  - Case
  - White space
  - Comments
  - Object references
  - Bind variables

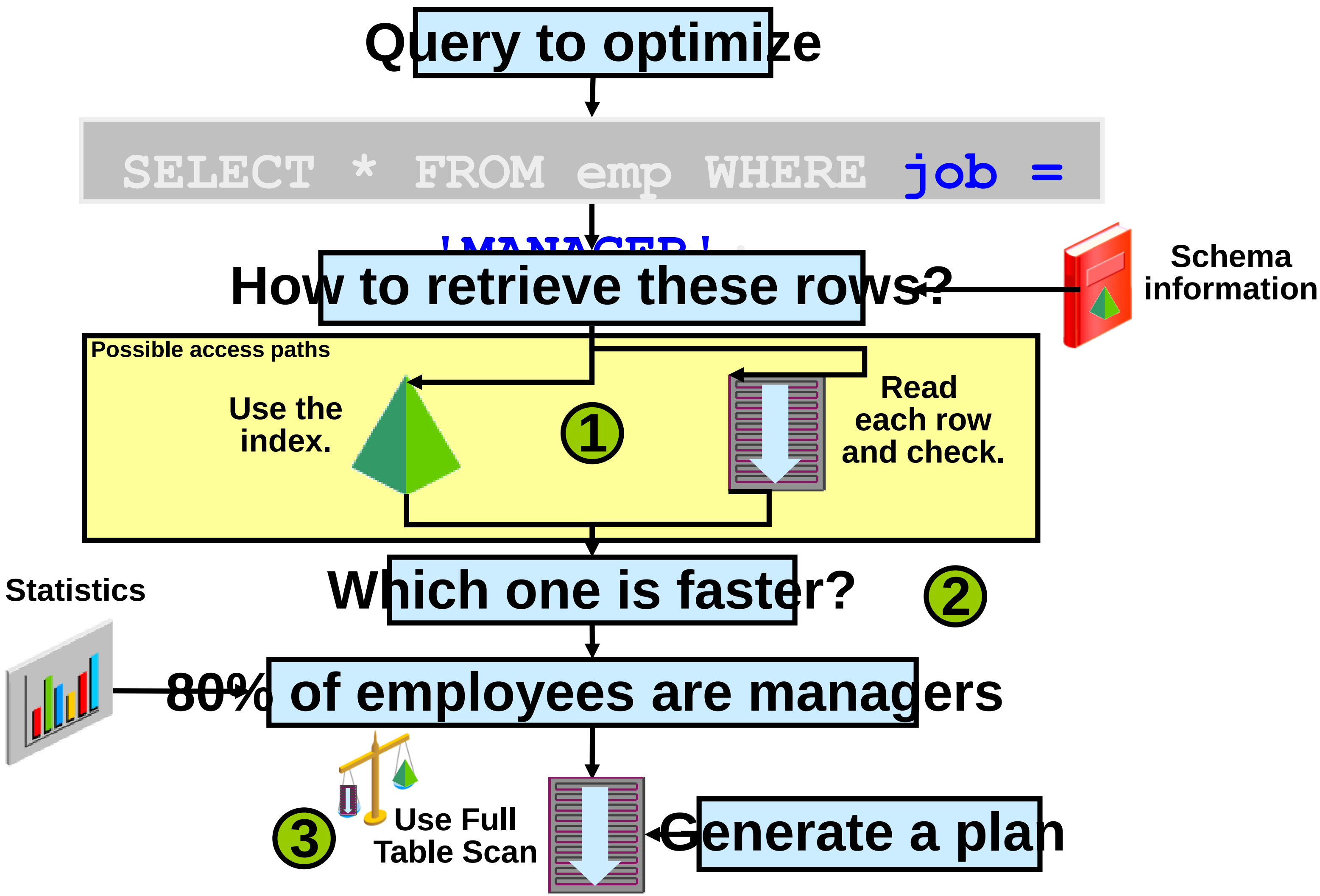
Performance Checklist

- Set initialization parameters and storage options.
- Verify resource usage of SQL statements.
- Validate connections by middleware.
- Verify cursor sharing.
- Validate migration of all required objects.
- Verify validity and availability of optimizer statistics.

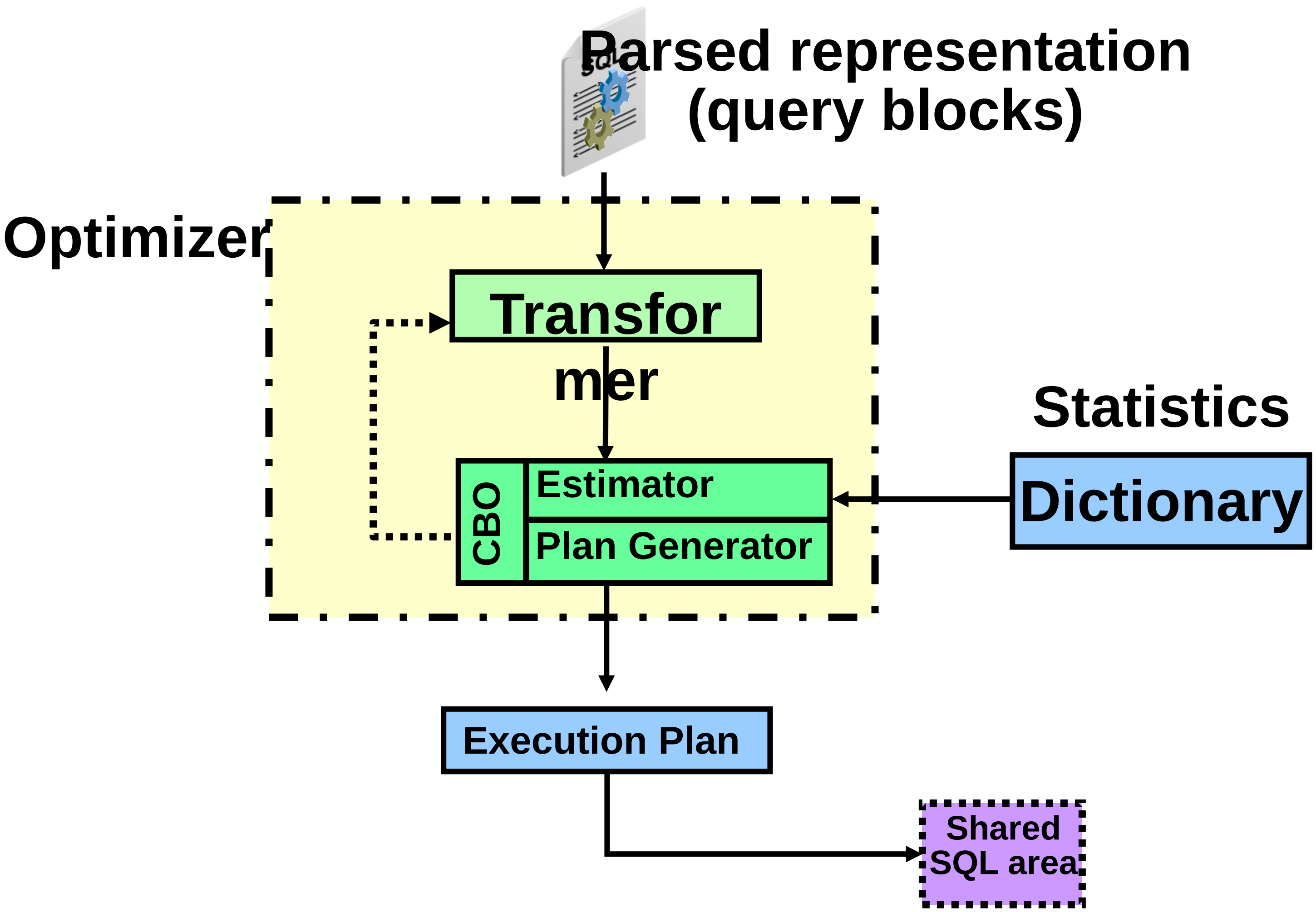
Why Do You Need an Optimizer?



Why Do You Need an Optimizer?



Optimization During Hard Parse Operation



Cost-Based Optimizer

- Piece of code:
  - Estimator
  - Plan generator
- Estimator determines cost of optimization suggestions made by the plan generator:
  - Cost: Optimizer's best estimate of the number of standardized I/Os made to execute a particular statement optimization
- Plan generator:
  - Tries out different statement optimization techniques
  - Uses the estimator to cost each optimization suggestion
  - Chooses the best optimization suggestion based on cost
  - Generates an execution plan for best optimization

Estimator: Selectivity

$$\text{Selectivity} = \frac{\text{Number of rows satisfying a condition}}{\text{Total number of rows}}$$

- Selectivity is the estimated proportion of a row set retrieved by a particular predicate or combination of predicates.
- It is expressed as a value between 0.0 and 1.0:
  - High selectivity: Small proportion of rows
  - Low selectivity: Big proportion of rows
- Selectivity computation:
  - If no statistics: Use dynamic sampling
  - If no histograms: Assume even distribution of rows
- Statistic information:
  - `DBA_TABLES` and `DBA_TAB_STATISTICS` (`NUM_ROWS`)
  - `DBA_TAB_COL_STATISTICS` (`NUM_DISTINCT`, `DENSITY`, `HIGH/LOW_VALUE`,…)

Estimator: Cardinality

Cardinality = Selectivity * Total number of rows

- Expected number of rows retrieved by a particular operation in the execution plan
- Vital figure to determine join, filters, and sort costs
- Simple example:
  - The number of distinct values in `DEV_NAME` is 203.
  - The number of rows in `COURSES` (original cardinality) is 1018.
  - Selectivity = 1/203 = 4.926*e-03
  - Cardinality = (1/203)*1018 = 5.01 (rounded off to 6)

```
SELECT days FROM courses WHERE dev_name
= 'ANGEL';
```

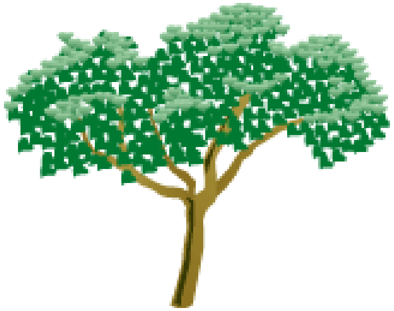
Plan Generator

```
select e.last_name, d.department_name
from   employees e, departments d
where  e.department_id = d.department_id;
```

```
Join order[1]:  DEPARTMENTS[D]#0  EMPLOYEES[E]#1
  NL Join:  Cost: 41.13  Resp: 41.13  Degree: 1
                SM cost: 8.01
                HA cost: 6.51
          Best:: JoinMethod: Hash
Cost: 6.51  Degree: 1  Resp: 6.51  Card: 106.00
Join order[2]:  EMPLOYEES[E]#1  DEPARTMENTS[D]#0
 NL Join:  Cost: 121.24  Resp: 121.24  Degree: 1
                SM cost: 8.01
                HA cost: 6.51
             Join order aborted
      Final cost for query block SEL$1 (#0)
               All Rows Plan:
             Best join order: 1
+-----------------------------------------------------------------+
| Id  | Operation             | Name        | Rows  | Bytes | Cost  |
+-----------------------------------------------------------------+
| 0   | SELECT STATEMENT      |             |       |       |     7 |
| 1   |  HASH JOIN            |             |   106 |  6042 |     7 |
| 2   |   TABLE ACCESS FULL | DEPARTMENTS|    27 |   810 |     3 |
| 3   |   TABLE ACCESS FULL | EMPLOYEES  |   107 |  2889 |     3 |
+-----------------------------------------------------------------+
```

What Is an Execution Plan?

- The execution plan of a SQL statement is composed of small building blocks called row sources for serial execution plans.
- The combination of row sources for a statement is called the execution plan.
- By using parent-child relationships, the execution plan can be displayed in a tree-like structure (text or graphical).



Where to Find Execution Plans?

- PLAN_TABLE **(SQL Developer or SQL*Plus)**
- V$SQL_PLAN **(Library Cache)**
- V$SQL_PLAN_MONITOR **(11*g*)**
- DBA_HIST_SQL_PLAN **(AWR)**
- STATS$SQL_PLAN **(Statspack)**
- **SQL management base (SQL plan baselines)**
- **SQL tuning set**
- **Trace files generated by** DBMS_MONITOR
- **Event 10053 trace file**
- **Process state dump trace file since 10*g*R2**

Viewing Execution Plans

- The `EXPLAIN PLAN` command followed by:
  - `SELECT from PLAN_TABLE`
  - `DBMS_XPLAN.DISPLAY()`
- SQL*Plus Autotrace: `SET AUTOTRACE ON`
- `DBMS_XPLAN.DISPLAY_CURSOR()`
- `DBMS_XPLAN.DISPLAY_AWR()`
- `DBMS_XPLAN.DISPLAY_SQLSET()`
- `DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE()`

The `EXPLAIN PLAN` Command

- Generates an optimizer execution plan
- Stores the plan in `PLAN_TABLE`
- Does not execute the statement itself

The `EXPLAIN PLAN` Command

EXPLAIN PLAN [**SET STATEMENT_ID = '***text***'**]

[**INTO** ***your plan table***]

FOR ***statement***

The EXPLAIN PLAN Command: Example

```
SQL> EXPLAIN PLAN
  2   SET STATEMENT_ID = 'demo01' FOR
  3   SELECT e.last_name, d.department_name
  4   FROM hr.employees e, hr.departments d
  5   WHERE e.department_id = d.department_id;
Explained.

SQL>
```

Note: The `EXPLAIN PLAN` command does not actually execute the statement.

PLAN_TABLE

- PLAN_TABLE:
  - Is automatically created to hold the EXPLAIN PLAN output.
  - You can create your own using utlxplan.sql.
  - Advantage: SQL is not executed
  - Disadvantage: May not be the actual execution plan
- PLAN_TABLE is hierarchical.
- Hierarchy is established with the ID and PARENT_ID columns.

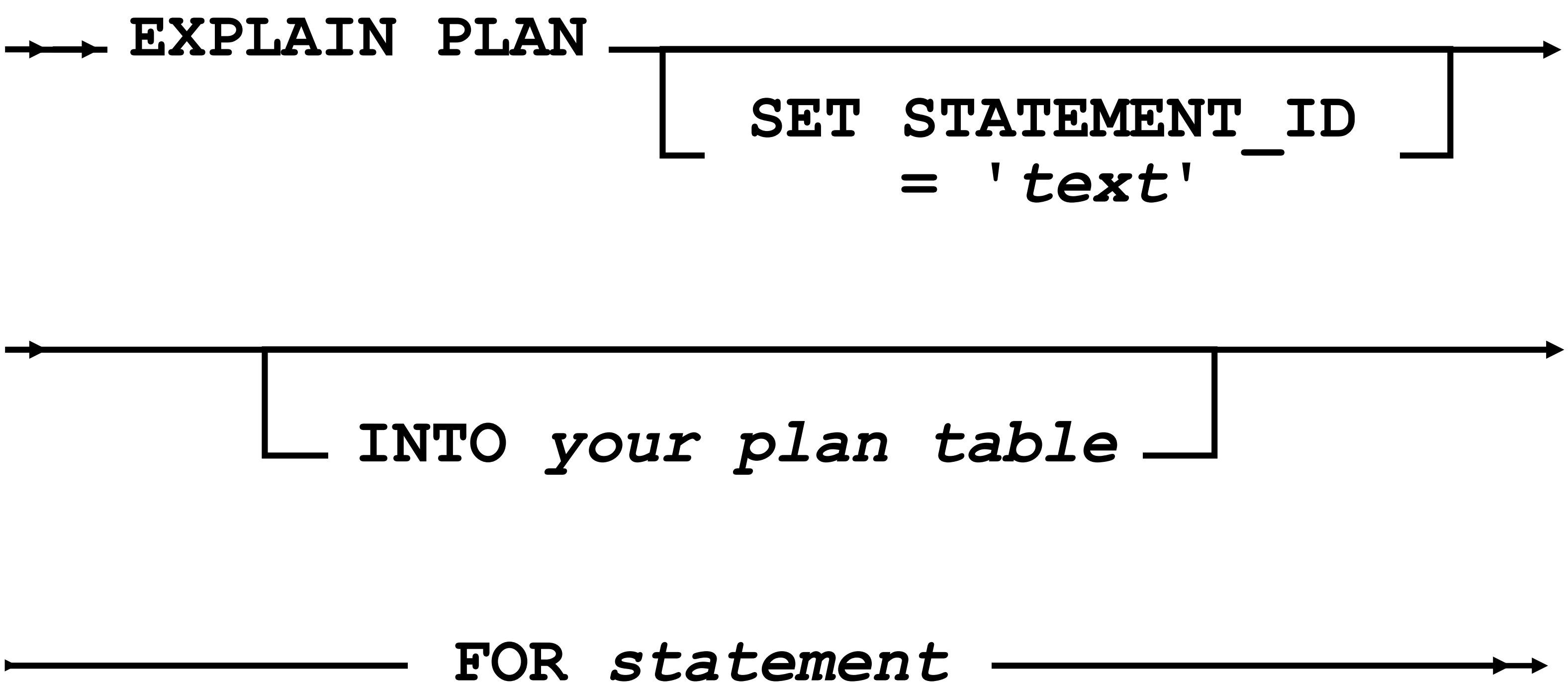
Displaying from `PLAN_TABLE`: Typical

```
SQL> EXPLAIN PLAN SET STATEMENT_ID = 'demo01' FOR SELECT * FROM emp
  2  WHERE ename = 'KING';

Explained.

SQL> SET LINESIZE 130
SQL> SET PAGESIZE 0
SQL> select * from table(DBMS_XPLAN.DISPLAY());

Plan hash value: 3956160932
---------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    37 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    37 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("ENAME"='KING')
```

Displaying from PLAN_TABLE: ALL

```
SQL> select * from table(DBMS_XPLAN.DISPLAY(null,null,'ALL'));

Plan hash value: 3956160932
--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    37 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    37 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
   1 - SEL$1 / EMP@SEL$1
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("ENAME"='KING')
Column Projection Information (identified by operation id):
-----------------------------------------------------------
 1 - "EMP"."EMPNO"[NUMBER,22], "ENAME"[VARCHAR2,10], "EMP"."JOB"[VARCHAR2,9],
     "EMP"."MGR"[NUMBER,22], "EMP"."HIREDATE"[DATE,7], "EMP"."SAL"[NUMBER,22],
     "EMP"."COMM"[NUMBER,22], "EMP"."DEPTNO"[NUMBER,22]
```

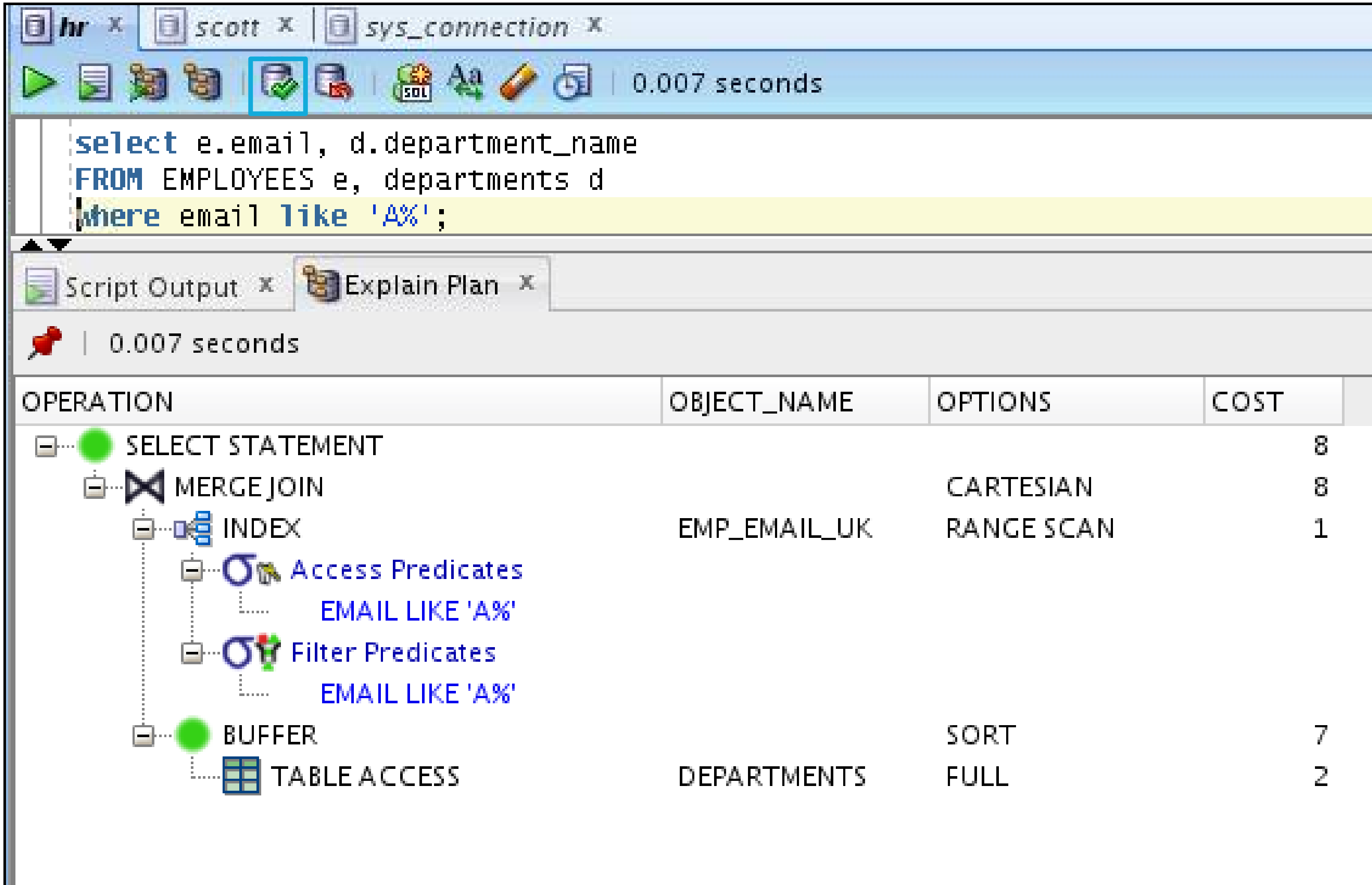
The EXPLAIN PLAN Command



Displaying from PLAN_TABLE: ADVANCED

```
select plan_table_output from table(DBMS_XPLAN.DISPLAY(null,null,'ADVANCED
 -PROJECTION -PREDICATE -ALIAS'));
Plan hash value: 3956160932
---------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    37 |     3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| EMP  |     1 |    37 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------
Outline Data
-------------
  /*+
      BEGIN_OUTLINE_DATA
      FULL(@"SEL$1" "EMP"@"SEL$1")
      OUTLINE_LEAF(@"SEL$1")
      ALL_ROWS
      DB_VERSION('11.1.0.6')
      OPTIMIZER_FEATURES_ENABLE('11.1.0.6')
      IGNORE_OPTIM_EMBEDDED_HINTS
      END_OUTLINE_DATA
  */
```

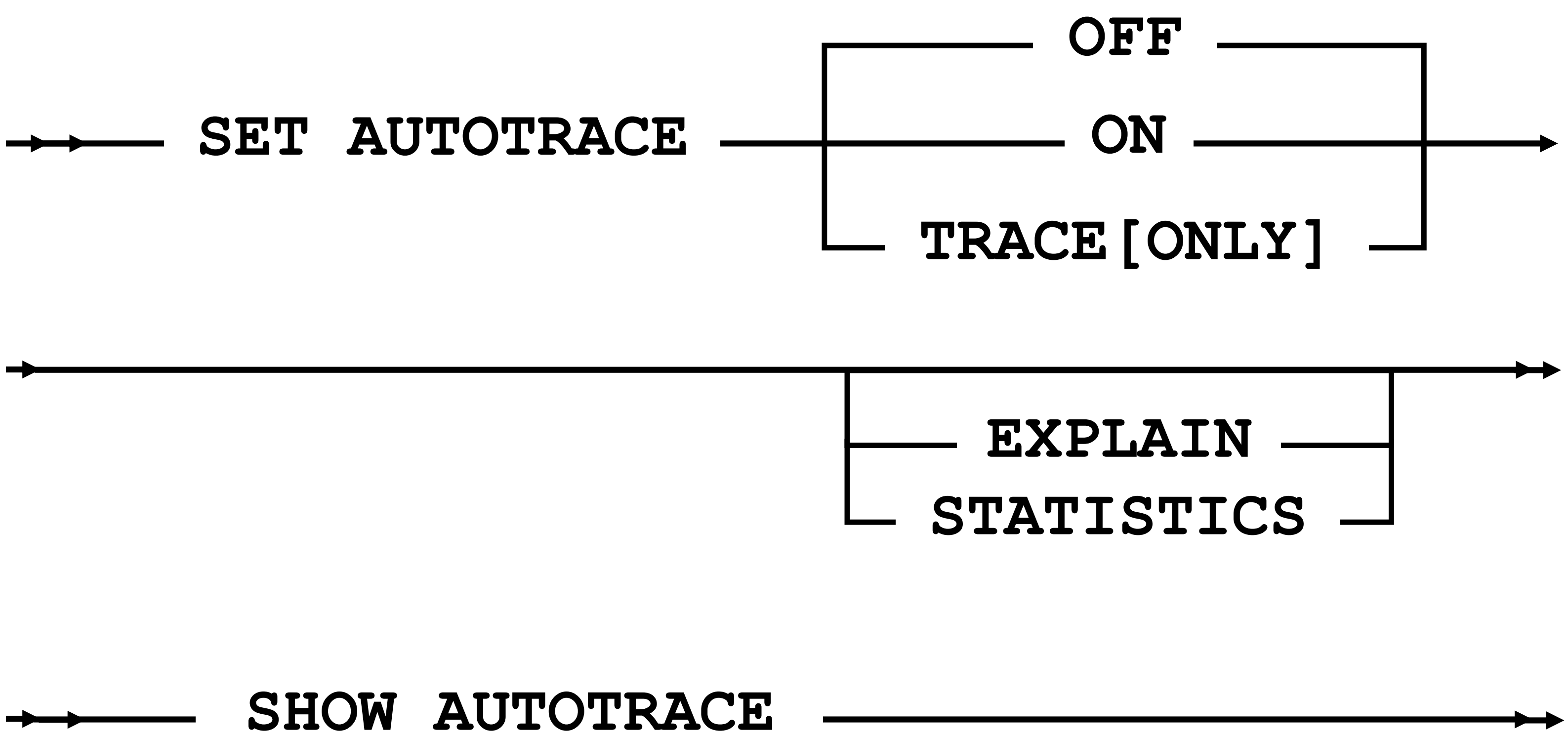
Explain Plan Using SQL Developer



AUTOTRACE

- Is a SQL*Plus and SQL Developer facility
- Was introduced with Oracle 7.3
- Needs a `PLAN_TABLE`
- Needs the `PLUSTRACE` role to retrieve statistics from some `V$` views
- By default, produces the execution plan and statistics after running the query
- May not be the execution plan used by the optimizer when using bind peeking (recursive `EXPLAIN PLAN`)

The `AUTOTRACE` Syntax



AUTOTRACE: Examples

- To start tracing statements using AUTOTRACE:
- To display

```
SQL> set autotrace on
```

- To display rows and statistics:
- To get the plan and the statistics only (suppress rows):

```
SQL> set autotrace traceonly explain
```

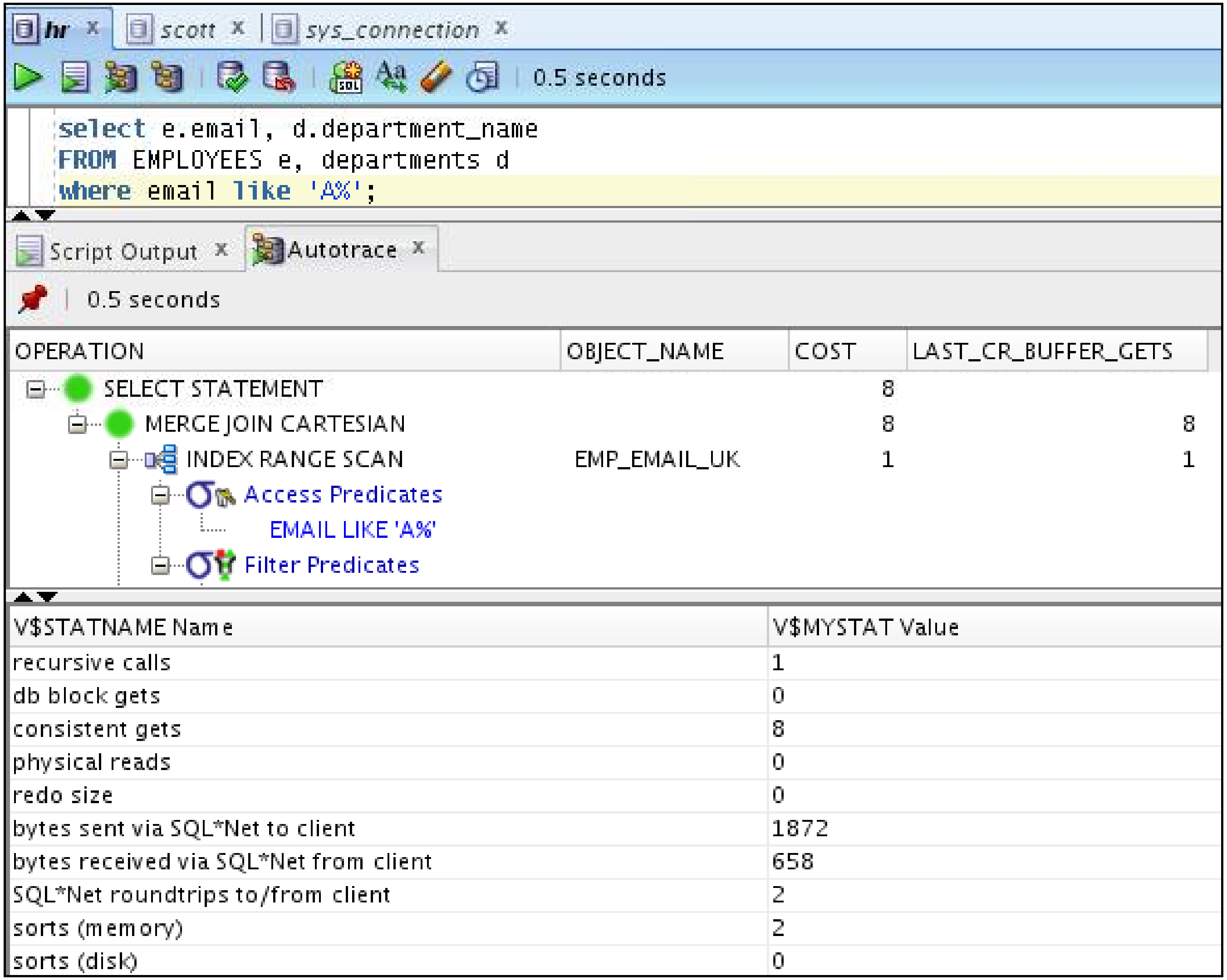
```
SQL> set autotrace on statistics
```

```
SQL> set autotrace traceonly
```

AUTOTRACE: Statistics

```
SQL> show autotrace
autotrace OFF
SQL> set autotrace traceonly statistics
SQL> SELECT * FROM oe.products;

288 rows selected.

Statistics
--------------------------------------------------------
       1334  recursive calls
          0  db block gets
        686  consistent gets
        394  physical reads
          0  redo size
     103919  bytes sent via SQL*Net to client
        629  bytes received via SQL*Net from client
         21  SQL*Net roundtrips to/from client
         22  sorts (memory)
          0  sorts (disk)
        288  rows processed
```

AUTOTRACE Using SQL Developer



Using the V$SQL_PLAN View

- V$SQL_PLAN provides a way of examining the execution plan for cursors that are still in the library cache.
- V$SQL_PLAN is very similar to PLAN_TABLE:
  - PLAN_TABLE shows a theoretical plan that can be used if this statement were to be executed.
  - V$SQL_PLAN contains the actual plan used.
- It contains the execution plan of every cursor in the library cache (including child).
- Link to V$SQL:
  - ADDRESS, HASH_VALUE, and CHILD_NUMBER

The V$SQL_PLAN Columns

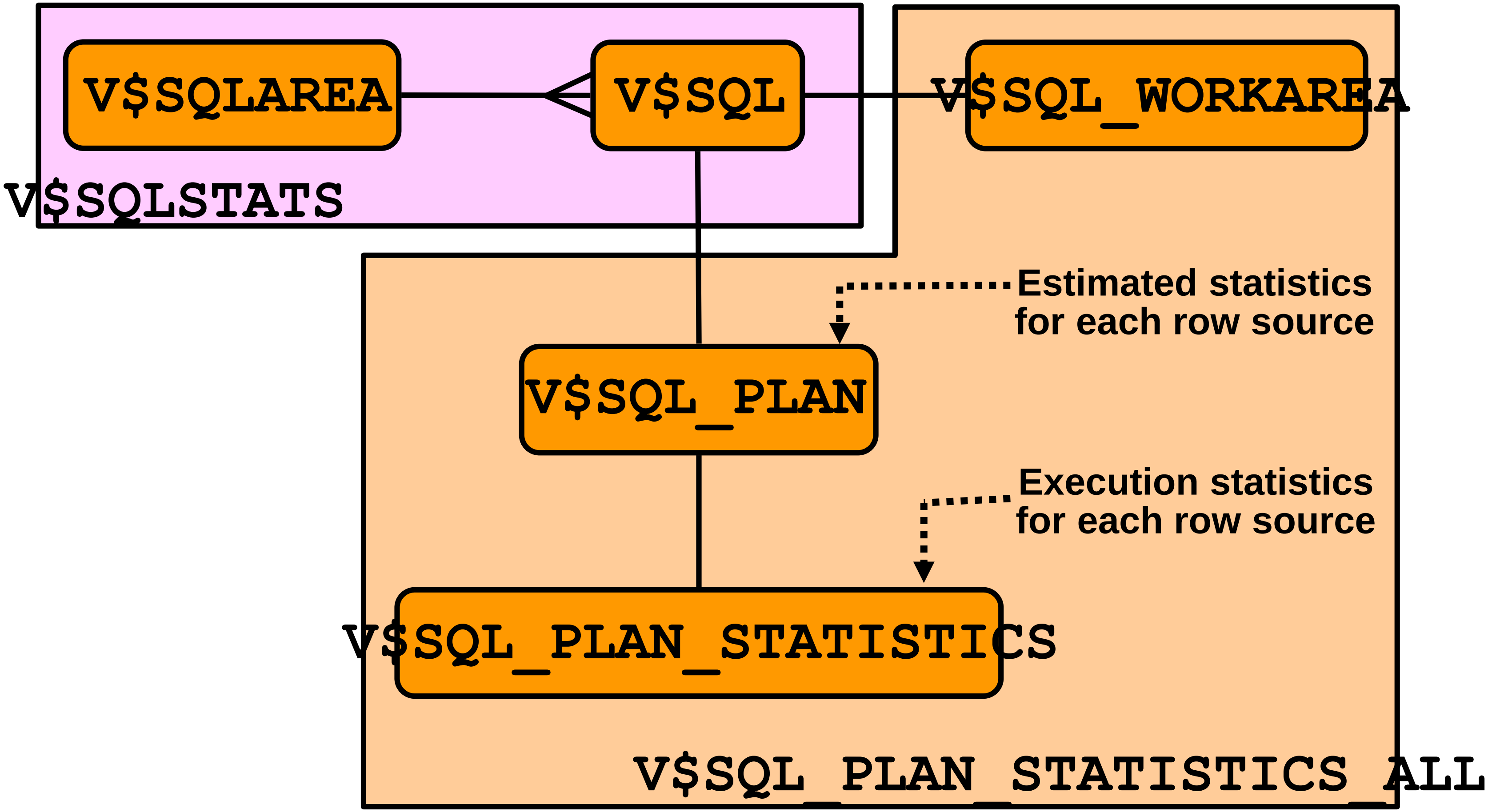
HASH_VALUE	Hash value of the parent statement in the library cache
ADDRESS	Address of the handle to the parent for this cursor
CHILD_NUMBER	Child cursor number using this execution plan
POSITION	Order of processing for all operations that have the same PARENT_ID
PARENT_ID	ID of the next execution step that operates on the output of the current step
ID	Number assigned to each step in the execution plan
PLAN_HASH_VALUE	Numerical representation of the SQL plan for the cursor

Note: This is only a partial listing of the columns.

The V$SQL_PLAN_STATISTICS View

- V$SQL_PLAN_STATISTICS provides actual execution statistics:
  - STATISTICS_LEVEL set to ALL
  - The GATHER_PLAN_STATISTICS hint
- V$SQL_PLAN_STATISTICS_ALL enables
  side-by-side comparisons of the optimizer estimates with the actual execution statistics.

Links Between Important Dynamic Performance Views



Querying V$SQL_PLAN

```
SELECT PLAN_TABLE_OUTPUT FROM
TABLE(DBMS_XPLAN.DISPLAY_CURSOR('47ju6102uvq5q'));
```

```
SQL_ID  47ju6102uvq5q, child number 0
-------------------------------------
SELECT e.last_name, d.department_name
FROM hr.employees e, hr.departments d WHERE
e.department_id =d.department_id

Plan hash value: 2933537672
--------------------------------------------------------------------------------
| Id  | Operation                    | Name          | Rows  | Bytes | Cost (%CPU|
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |               |       |       |     6 (100|
|   1 |  MERGE JOIN                  |               |   106 |  2862 |     6  (17|
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS   |    27 |   432 |     2   (0|
|   3 |    INDEX FULL SCAN           | DEPT_ID_PK    |    27 |       |     1   (0|
|*  4 |   SORT JOIN                  |               |   107 |  1177 |     4  (25|
|   5 |    TABLE ACCESS FULL         | EMPLOYEES     |   107 |  1177 |     3   (0|
--------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
       filter("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")

24 rows selected.
```

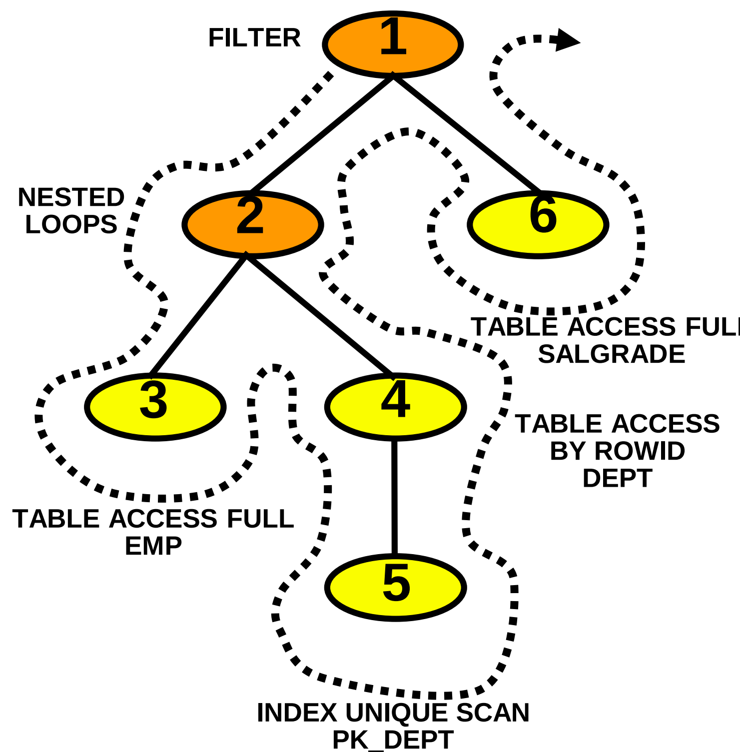
Execution Plan Interpretation: Example 1

```
SQL> alter session set statistics_level=ALL;

Session altered.

SQL> select /*+ RULE to make sure it reproduces 100% */ ename,job,sal,dname
from emp,dept where dept.deptno = emp.deptno and not exists (select * from salgrade
where emp.sal between losal and hisal);

no rows selected

SQL> select * from table(dbms_xplan.display_cursor(null,null,'TYPICAL IOSTATS
LAST'));

SQL_ID  274019myw3vuf, child number 0
-------------------------------------
…
Plan hash value: 1175760222
--------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Starts | A-Rows | Buffers |
--------------------------------------------------------------------------------
|*  1 |  FILTER                      |           |      1 |      0 |      61 |
|   2 |   NESTED LOOPS               |           |      1 |     14 |      25 |
|   3 |    TABLE ACCESS FULL         | EMP       |      1 |     14 |       7 |
|   4 |    TABLE ACCESS BY INDEX ROWID| DEPT     |     14 |     14 |      18 |
|*  5 |     INDEX UNIQUE SCAN        | PK_DEPT   |     14 |     14 |       4 |
|*  6 |   TABLE ACCESS FULL          | SALGRADE  |     12 |     12 |      36 |
--------------------------------------------------------------------------------
…
```

Execution Plan Interpretation: Example 1

```
SELECT /*+ RULE */ ename,job,sal,dname
FROM emp,dept
WHERE dept.deptno=emp.deptno and not exists(SELECT *
                                            FROM salgrade
                                            WHERE emp.sal between losal and hisal);
```

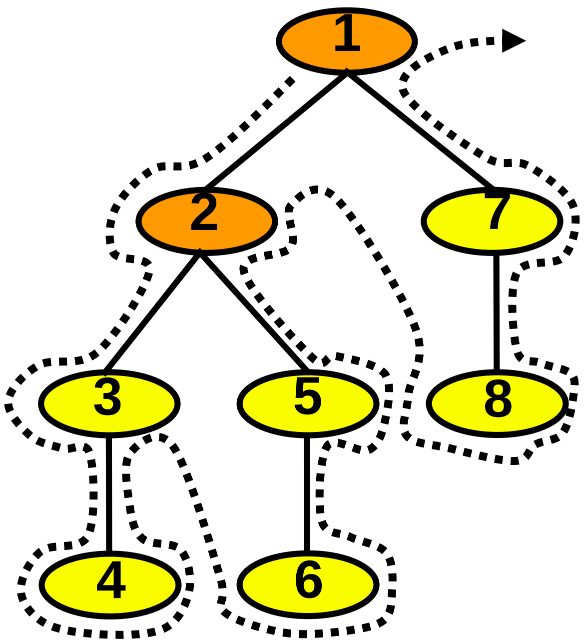
```
--------------------------------------------------
| Id  | Operation                      | Name     |
--------------------------------------------------
|   0 | SELECT STATEMENT               |          |
|*  1 |  FILTER                        |          |
|   2 |   NESTED LOOPS                 |          |
|   3 |    TABLE ACCESS FULL           | EMP      |
|   4 |    TABLE ACCESS BY INDEX ROWID| DEPT     |
|*  5 |     INDEX UNIQUE SCAN          | PK_DEPT  |
|*  6 |   TABLE ACCESS FULL            | SALGRADE |
--------------------------------------------------

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

   1 - filter( NOT EXISTS
     (SELECT 0 FROM "SALGRADE" "SALGRADE" WHERE
      "HISAL">=:B1 AND "LOSAL"<=:B2))
   5 - access("DEPT"."DEPTNO"="EMP"."DEPTNO")
   6 - filter("HISAL">=:B1 AND "LOSAL"<=:B2)
```



Execution Plan Interpretation: Example 2

```
SQL> select /*+ USE_NL(d) use_nl(m) */ m.last_name as dept_manager
  2  ,      d.department_name
  3  ,      l.street_address
  4  from   hr.employees m   join
  5         hr.departments d on (d.manager_id = m.employee_id)
  6         natural join
  7         hr.locations l
  8  where  l.city = 'Seattle';
```

```
0    SELECT STATEMENT
1 0    NESTED LOOPS
2 1     NESTED LOOPS
3 2      TABLE ACCESS BY INDEX ROWID LOCATIONS
4 3       INDEX RANGE SCAN           LOC_CITY_IX
5 2      TABLE ACCESS BY INDEX ROWID DEPARTMENTS
6 5       INDEX RANGE SCAN           DEPT_LOCATION_IX
7 1     TABLE ACCESS BY INDEX ROWID  EMPLOYEES
8 7      INDEX UNIQUE SCAN           EMP_EMP_ID_PK
```



Reviewing the Execution Plan

- Drive from the table that has most selective filter.
- Look for the following:
  - Driving table has the best filter
  - Fewest number of rows are returned to the next step
  - The join method is appropriate for the number of rows returned
  - Views are correctly used
  - Unintentional Cartesian products
  - Tables accessed efficiently

Querying the AWR

- Retrieve all execution plans stored for a particular `SQL_ID`.

```
SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE (DBMS_XPLAN.DISPLAY_AWR('454rug2yva18w'));


PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------
SQL_ID 454rug2yva18w
--------------------
select /* example */ * from hr.employees natural join hr.departments

Plan hash value: 4179021502

---------------------------------------------------------------------------------
| Id  | Operation           | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |              |       |       |     6 (100)|          |
|   1 |  HASH JOIN          |              |    11 |   968 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPARTMENTS  |    11 |   220 |     2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMPLOYEES    |   107 |  7276 |     3   (0)| 00:00:01 |
---------------------------------------------------------------------------------
```

```
SELECT tf.* FROM DBA_HIST_SQLTEXT ht, table
    (DBMS_XPLAN.DISPLAY_AWR(ht.sql_id,null, null,  'ALL' )) tf
 WHERE ht.sql_text like '%JF%';
```

Generating SQL Reports from AWR Data

```
SQL> @$ORACLE_HOME/rdbms/admin/awrsqrpt

Specify the Report Type …
Would you like an HTML report, or a plain text report?
Specify the number of days of snapshots to choose from
Specify the Begin and End Snapshot Ids …
Specify the SQL Id …
Enter value for sql_id: dvza55c7zu0yv
Specify the Report Name …
```

WORKLOAD REPOSITORY SQL Report

Snapshot Period Summary

DB Name	DB Id	Instance	Inst num	Startup Time	Release	RAC
ORCL	1249102530	orcl	1	14-Jun-10 02:06	11.2.0.1.0	NO

	Snap Id	Snap Time	Sessions	Cursors/Session
Begin Snap:	218	17-Jun-10 22:00:47	43	6.3
End Snap:	226	18-Jun-10 04:21:15	40	6.4
Elapsed:		380.47 (mins)		
DB Time:		5.54 (mins)		

SQL ID: dvza55c7zu0yv

- 1st Capture and Last Capture Snap IDs refer to Snapshot IDs witin the snapshot range
- SELECT sql_id, sql_text from DBA_HIST_SQLTEXT where sql_text like '%sa...

#	Plan Hash Value	Total Elapsed Time(ms)	Executions	1st Capture Snap ID	Last Capture Snap ID
1	1258587641	429	1	226	226

Back to Top

Plan 1(PHV: 1258587641)

- Plan Statistics
- Execution Plan

tkprof Output with Index: Example

```
...
select max(cust_credit_limit) from customers where cust_city ='Paris'

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      0.00       0.00          1         77          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.00       0.00          1         77          0           1

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 88

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=77 pr=1 pw=0 time=0 us)
     77   TABLE ACCESS BY INDEX ROWID CUSTOMERS (cr=77 pr=1 pw=0 time=760 us
                                                 cost=85 size=1260 card=90)
     77    INDEX RANGE SCAN CUST_CUST_CITY_IDX (cr=2 pr=1 pw=0 time=152 us
                                         cost=1 size=0 card=90)(object id 78183)
```

Invoking the `tkprof` Utility

```
tkprof inputfile outputfile [waits=yes|no]
                            [sort=option]
                            [print=n]
                            [aggregate=yes|no]
                            [insert=sqlscriptfile]
                            [sys=yes|no]
                            [table=schema.table]
                            [explain=user/password]
                            [record=statementfile]
                            [width=n]
```

Invoking the `tkprof` Utility

```
tkprof inputfile outputfile [waits=yes|no]
                            [sort=option]
                            [print=n]
                            [aggregate=yes|no]
                            [insert=sqlscriptfile]
                            [sys=yes|no]
                            [table=schema.table]
                            [explain=user/password]
                            [record=statementfile]
                            [width=n]
```

Output of the `tkprof` Command

- There are seven categories of trace statistics:

`Count`	Number of times the procedure was executed
`CPU`	Number of seconds to process
`Elapsed`	Total number of seconds to execute
`Disk`	Number of physical blocks read
`Query`	Number of logical buffers read for consistent read
`Current`	Number of logical buffers read in current mode
`Rows`	Number of rows processed by the fetch or execute

Output of the `tkprof` Command

➢The `tkprof` output also includes the following:

- Recursive SQL statements
- Library cache misses
- Parsing user ID
- Execution plan
- Optimizer mode or hint
- Row source operation

```
...
Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 85

Rows     Row Source Operation
-------  ---------------------------------------------------
      5  TABLE ACCESS BY INDEX ROWID EMPLOYEES (cr=4 pr=1 pw=0 time=0 us …
      5   INDEX RANGE SCAN EMP_NAME_IX (cr=2 pr=1 pw=0 time=80 us cost=1 …
...
```

tkprof Output with No Index: Example

```
...
select max(cust_credit_limit) from customers where cust_city ='Paris'

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      0.02       0.10         72       1459          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.02       0.10         72       1459          0           1

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 88

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=1459 pr=72 pw=0 time=0 us)
     77   TABLE ACCESS FULL CUSTOMERS (cr=1459 pr=72 pw=0 time=4104 us
          cost=405 size=1260 card=90)
...
```

tkprof Output with Index: Example

```
...
select max(cust_credit_limit) from customers where cust_city ='Paris'

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      0.00       0.00          1         77          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.00       0.00          1         77          0           1

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 88

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=77 pr=1 pw=0 time=0 us)
     77   TABLE ACCESS BY INDEX ROWID CUSTOMERS (cr=77 pr=1 pw=0 time=760 us
           cost=85 size=1260 card=90)
     77    INDEX RANGE SCAN CUST_CUST_CITY_IDX (cr=2 pr=1 pw=0 time=152 us
           cost=1 size=0 card=90)(object id 78183)
```

`tkprof` Output with Index: Example

```
...
select max(cust_credit_limit) from customers where cust_city ='Paris'

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      0.00       0.00          1         77          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.00       0.00          1         77          0           1

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 88

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE (cr=77 pr=1 pw=0 time=0 us)
     77   TABLE ACCESS BY INDEX ROWID CUSTOMERS (cr=77 pr=1 pw=0 time=760 us
                                       cost=85 size=1260 card=90)
     77    INDEX RANGE SCAN CUST_CUST_CITY_IDX (cr=2 pr=1 pw=0 time=152 us
                                 cost=1 size=0 card=90)(object id 78183)
```

Session Level Tracing: Example

- For all sessions in the database:

```
EXEC dbms_monitor.DATABASE_TRACE_ENABLE(TRUE,TRUE);
```

```
EXEC dbms_monitor.DATABASE_TRACE_DISABLE();
```

```
EXEC dbms_monitor.SESSION_TRACE_ENABLE(session_id=>
27, serial_num=>60, waits=>TRUE, binds=>FALSE);
```

```
EXEC dbms_monitor.SESSION_TRACE_DISABLE(session_id
=>27, serial_num=>60);
```

Trace Your Own Session

- Enabling trace:

```
EXEC DBMS_SESSION.SESSION_TRACE_ENABLE(waits =>
TRUE, binds => FALSE);
```

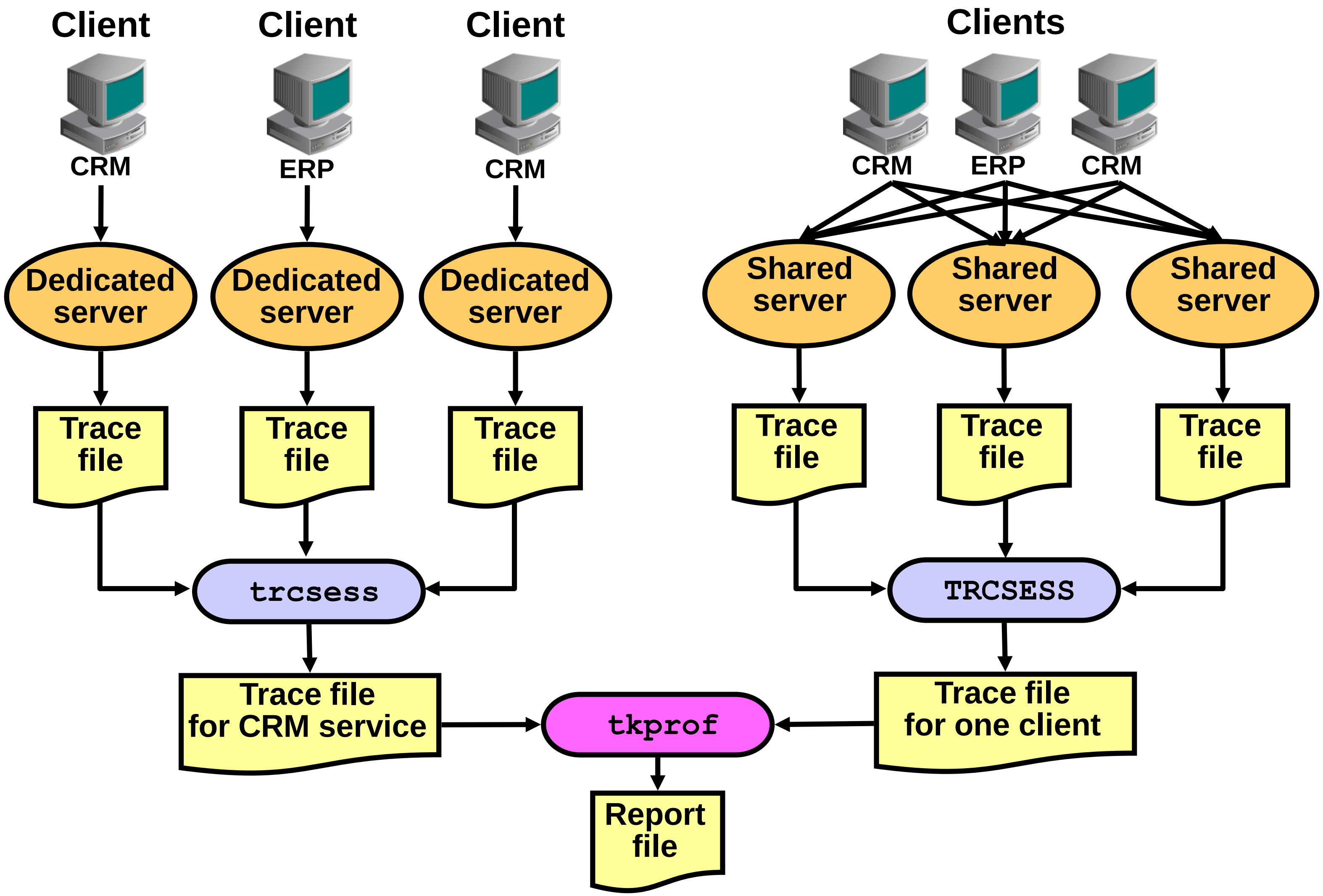
- Disabling trace:

```
EXEC DBMS_SESSION.SESSION_TRACE_DISABLE();
```

- Easily identifying your trace files:

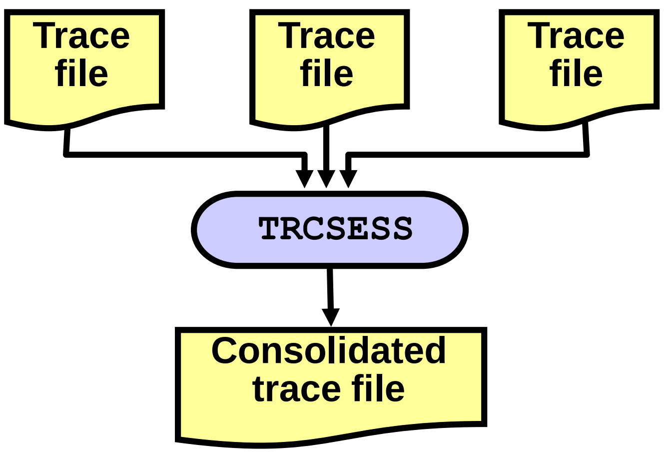
```
alter session set
tracefile_identifier='mytraceid';
```

The `trcsess` Utility



Invoking the `trcsess` Utility

```
trcsess  [output=output_file_name]
           [session=session_id]
         [clientid=client_identifier]
           [service=service_name]
           [action=action_name]
           [module=module_name]
           [<trace file names>]
```



The `trcsess` Utility: Example

```
exec dbms_session.set_identifier('HR session');
```

First session

Second session

```
exec dbms_session.set_identifier('HR session');
```

```
exec DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE( -
client_id=>'HR session', waits => FALSE, -
binds => FALSE);
```

Third session

```
select * from employees;
...
```

```
select * from departments;
...
```

```
exec DBMS_MONITOR.CLIENT_ID_TRACE_DISABLE( -
client_id => 'HR session');
```

```
trcsess output=mytrace.trc clientid='HR session'
$ORACLE_BASE/diag/rdbms/orcl/orcl/trace/*.trc
```

SQL Trace File Contents

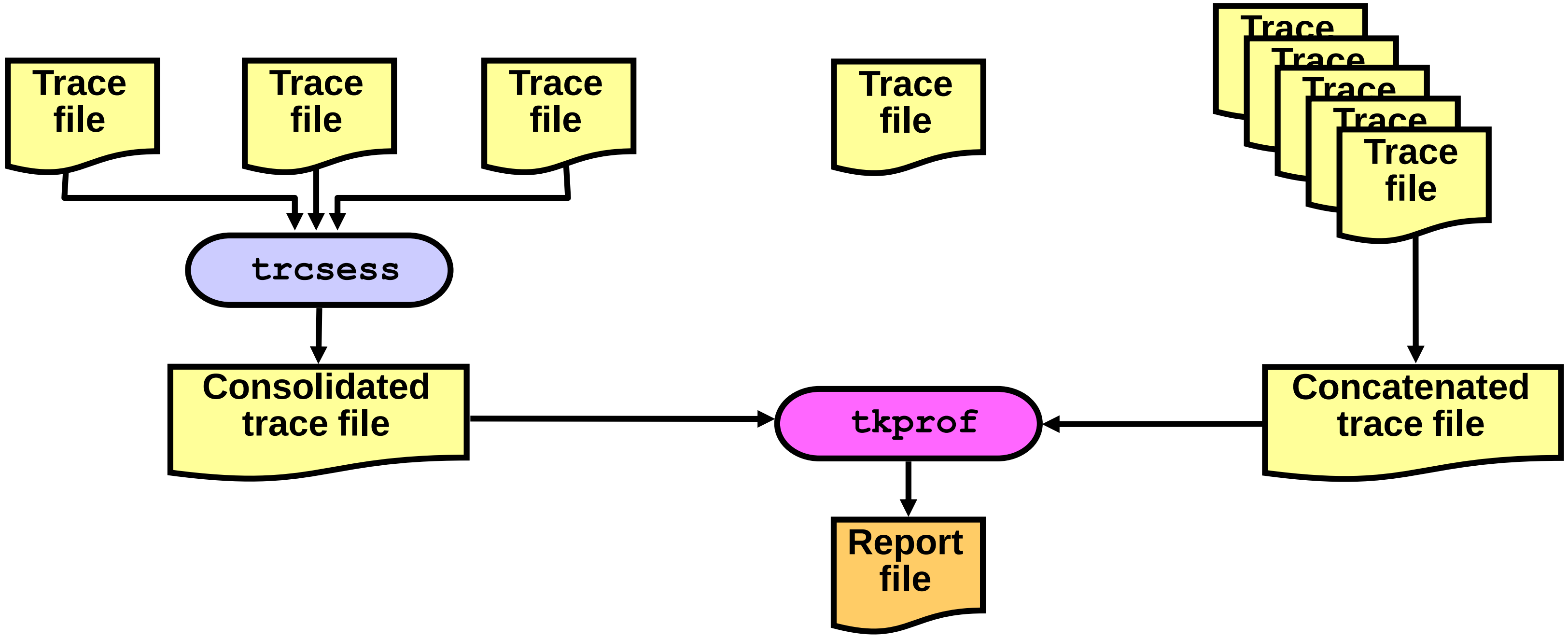
- Parse, execute, and fetch counts
- CPU and elapsed times
- Physical reads and logical reads
- Number of rows processed
- Misses on the library cache
- Username under which each parse occurred
- Each commit and rollback
- Wait event and bind data for each SQL statement
- Row operations showing the actual execution plan of each SQL statement
- Number of consistent reads, physical reads, physical writes, and time elapsed for each operation on a row

SQL Trace File Contents: Example

```
*** [ Unix process pid: 15911 ]
*** 2010-07-29 13:43:11.327
*** 2010-07-29 13:43:11.327
*** 2010-07-29 13:43:11.327
*** 2010-07-29 13:43:11.327
…
=====================
PARSING IN CURSOR #2 len=23 dep=0 uid=85 oct=3 lid=85 tim=1280410994003145 hv=40
69246757 ad='4cd57ac0' sqlid='f34thrbt8rjt5'
select * from employees
END OF STMT
PARSE #2:c=3000,e=2264,p=0,cr=0,cu=0,mis=1,r=0,dep=0,og=1,plh=1445457117,
tim=1280410994003139
EXEC #2:c=0,e=36,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=1,plh=1445457117,
tim=1280410994003312
FETCH #2:c=0,e=215,p=0,cr=3,cu=0,mis=0,r=1,dep=0,og=1,plh=1445457117,
tim=1280410994003628
FETCH #2:c=0,e=89,p=0,cr=5,cu=0,mis=0,r=15,dep=0,og=1,plh=1445457117,
tim=1280410994004232
…
FETCH #2:c=0,e=60,p=0,cr=1,cu=0,mis=0,r=1,dep=0,og=1,plh=1445457117,
tim=1280410994107857
STAT #2 id=1 cnt=107 pid=0 pos=1 obj=73933 op='TABLE ACCESS FULL EMPLOYEES (cr=15
pr=0 pw=0 time=0 us cost=3 size=7383 card=107)'
XCTEND rlbk=0, rd_only=1, tim=1280410994108875
=====================
```

Formatting SQL Trace Files: Overview

- Use the `tkprof` utility to format your SQL trace files:
  - Sort raw trace file to exhibit top SQL statements
  - Filter dictionary statements

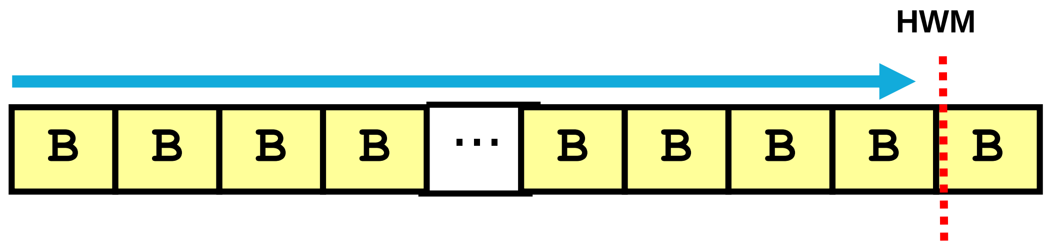


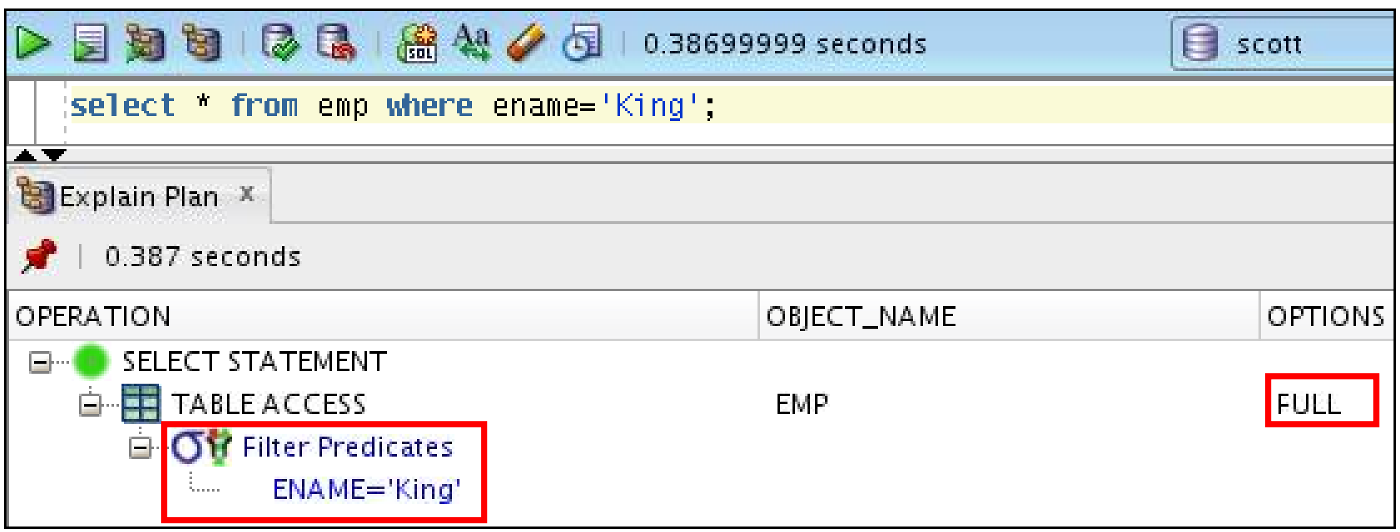
Main Structures and Access Paths

Structures	Access Paths
Tables	Full Table Scan Rowid Scan Sample Table Scan
Indexes	Index Scan (Unique) Index Scan (Range) Index Scan (Full) Index Scan (Fast Full) Index Scan (Skip) Index Scan (Index Join) Using Bitmap Indexes Combining Bitmap Indexes

Full Table Scan

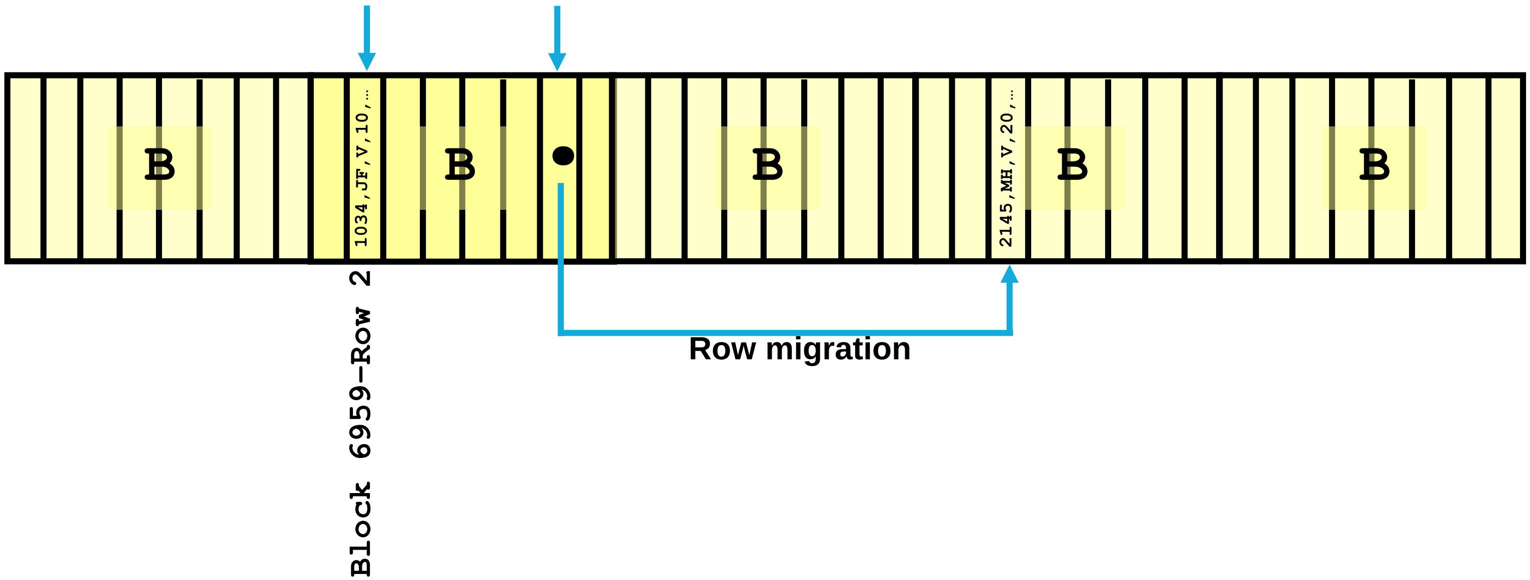
- Performs multiblock reads
  (here `DB_FILE_MULTIBLOCK_READ_COUNT = 4`)
- Reads all formatted blocks below the high-water mark
- May filter rows
- Is faster than index
  range scans for large amount of data





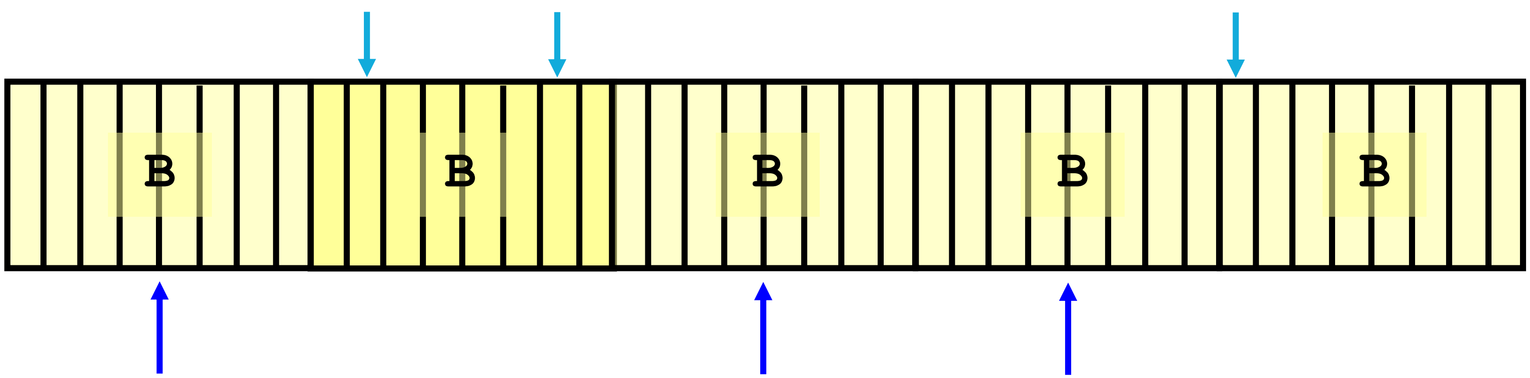
ROWID Scan

```
select * from scott.emp where rowid='AAAQ+LAAEAAAAAfAAJ';

-----------------------------------------------------------------
| Id  | Operation                 | Name | Rows  | Bytes | Cost |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT          |      |     1 |    37 |     1|
|   1 |  TABLE ACCESS BY USER ROWID| EMP  |     1 |    37 |     1|
-----------------------------------------------------------------
```



Sample Table Scans

```
SELECT * FROM emp SAMPLE BLOCK (10) SEED (1);

----------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost  (%CPU)|
----------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |     4 |    99 |     2    (0)|
|   1 |  TABLE ACCESS SAMPLE| EMP  |     4 |    99 |     2    (0)|
----------------------------------------------------------------------
```



Automatic Workload Repository (AWR)

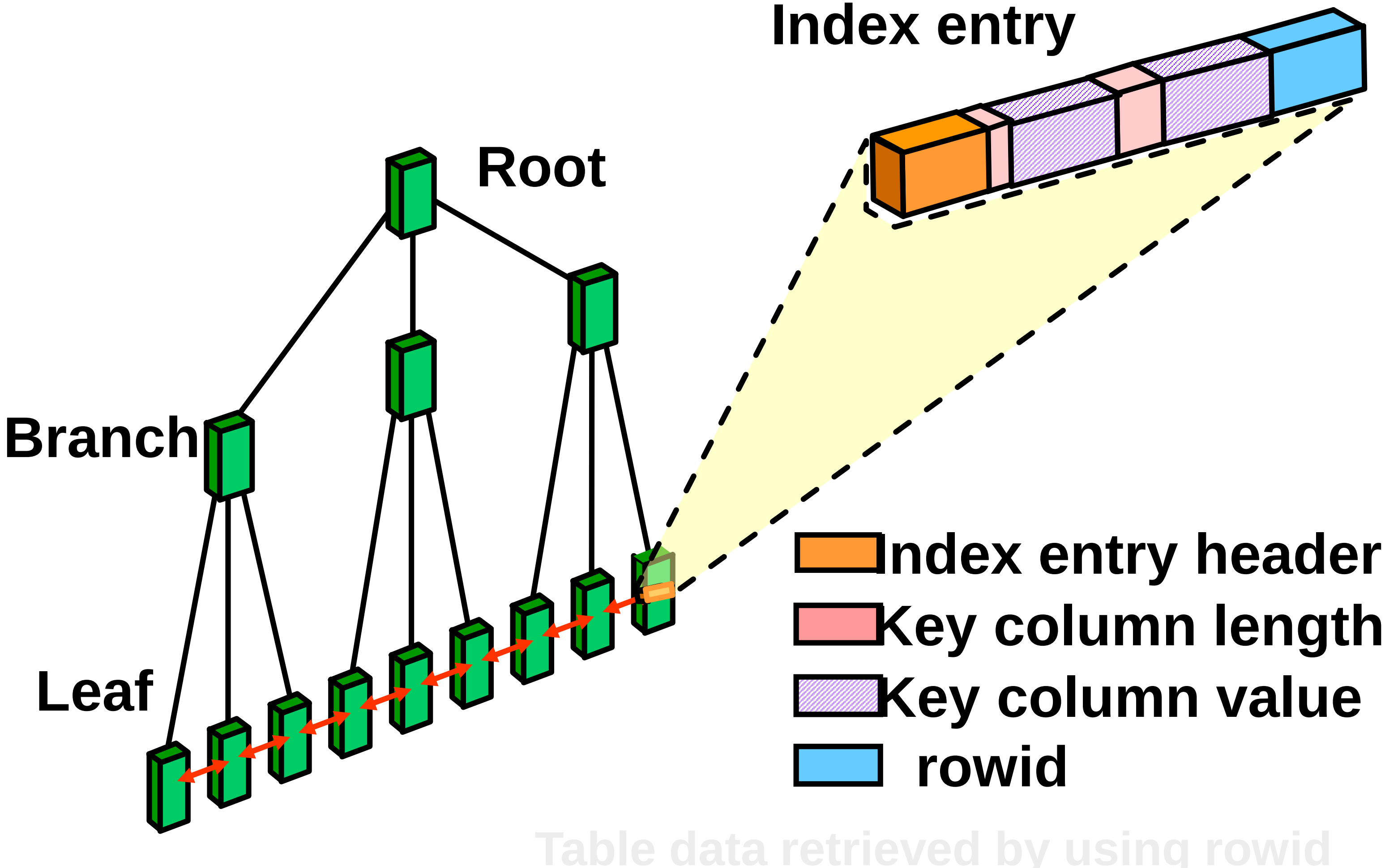
- Collects, processes, and maintains performance statistics for problem-detection and self-tuning purposes
- Statistics include:
  - Object statistics
  - Time-model statistics
  - Some system and session statistics
  - Active Session History (ASH) statistics
- Automatically generates snapshots of the performance data

Indexes: Overview

➢ Indexes

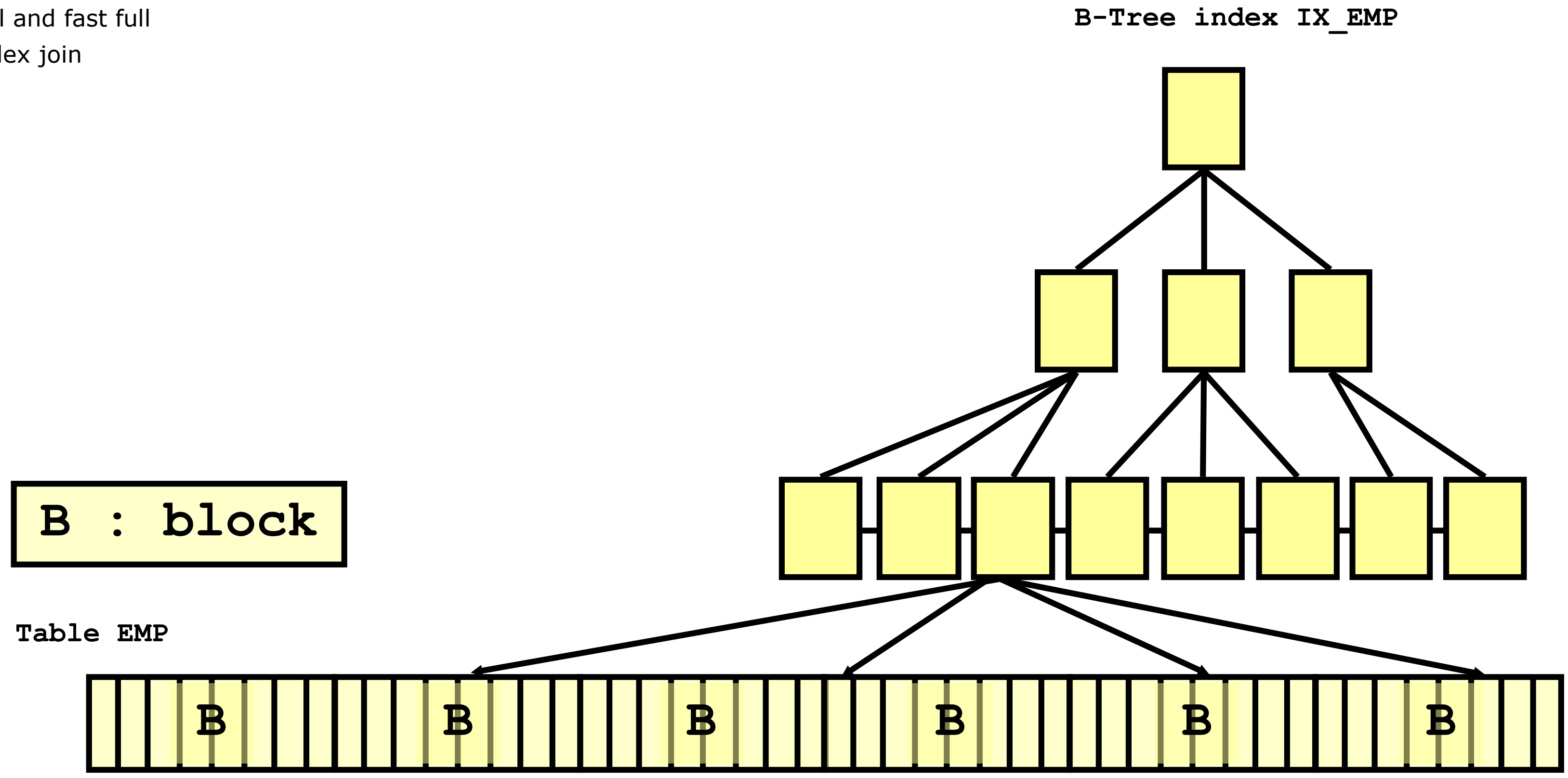
- Storage techniques:
  - B*-tree indexes: The default and the most common
    - Normal
    - Function based: Precomputed value of a function or expression
    - Index-organized table (IOT)
  - Bitmap indexes
  - Cluster indexes: Defined specifically for cluster
- Index attributes:
  - Key compression
  - Reverse key
  - Ascending, descending
- Domain indexes: Specific to an application or cartridge

Normal B*-tree Indexes



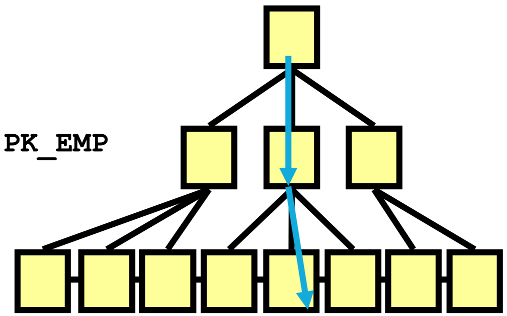
Index Scans

- Types of index scans:
  - Unique
  - Min/Max
  - Range (Descending)
  - Skip
  - Full and fast full
  - Index join



Index Unique Scan

index UNIQUE Scan PK_EMP



```
create unique index PK_EMP on EMP(empno)

select * from emp where empno = 9999;
```

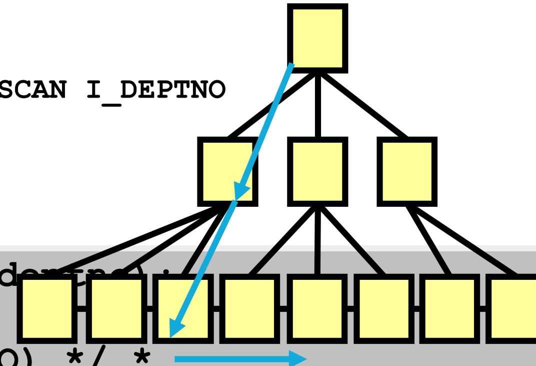
Explain Plan

0.024 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			1	1
TABLE ACCESS	EMP	BY INDEX ROWID	1	1
INDEX	PK_EMP	UNIQUE SCAN	0	1
Access Predicat				
EMPNO=7839				

Index Range Scan

Index Range SCAN I_DEPTNO



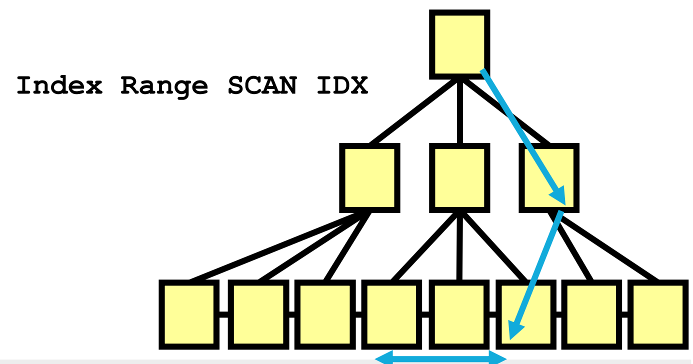
```
create index I_DEPTNO on EMP(deptno);

select /*+ INDEX(EMP I_DEPTNO) */ *
from emp where deptno = 10 and sal > 1000;
```

Script Output | Explain Plan

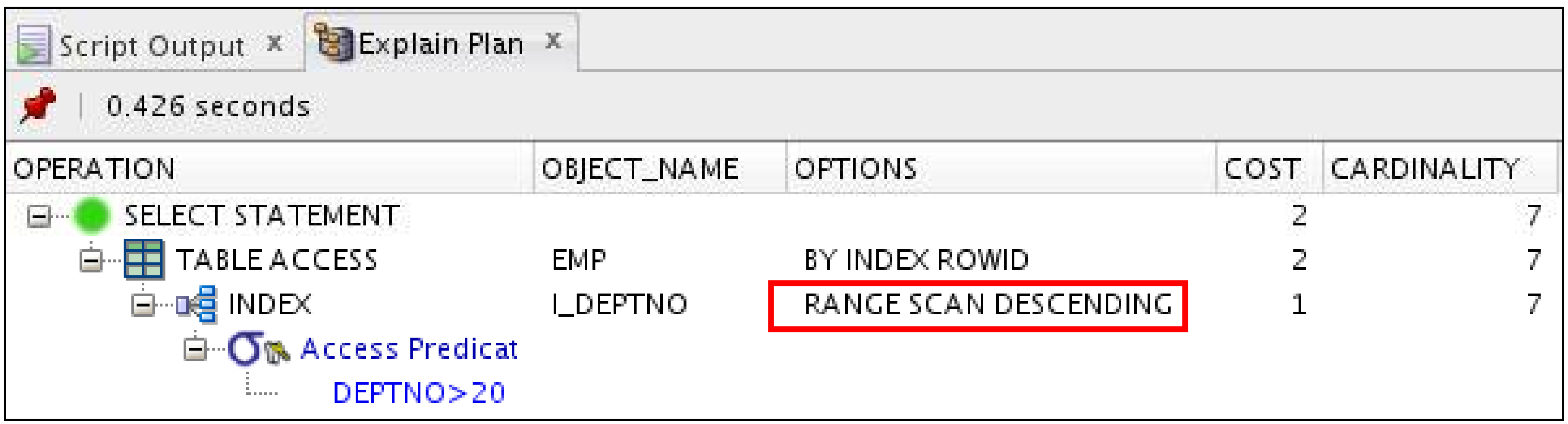
0.012 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	4
TABLE ACCESS	EMP	BY INDEX ROWID	2	4
Filter Predicates				
SAL>1000				
INDEX	I_DEPTNO	RANGE SCAN	1	5
Access Predicat				
DEPTNO=10				

Index Range Scan: Descending



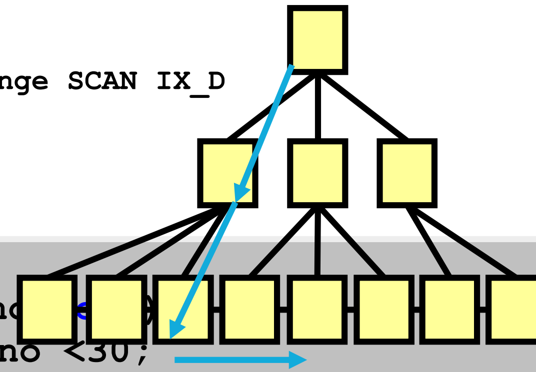

```
select * from emp where deptno>20 order by deptno desc;
```



OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	7
TABLE ACCESS	EMP	BY INDEX ROWID	2	7
INDEX	I_DEPTNO	RANGE SCAN DESCENDING	1	7
Access Predicat				
DEPTNO>20				

Descending Index Range Scan

Index Range SCAN IX_D



```
drop index I_Deptno;
create index IX_D on EMP(deptno desc);
select * from emp where deptno <30;
```

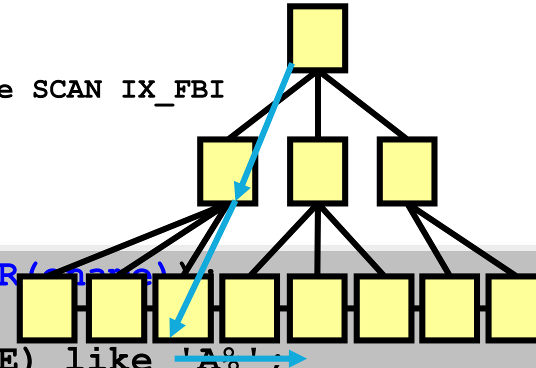
Script Output | Explain Plan

0.011 seconds

OPERATION	OBJECT_NA...	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	9
TABLE ACCESS	EMP	BY INDEX ROWID	2	9
INDEX	IX_D	RANGE SCAN	1	1
Access Predicates				
SYS_OP_DESCEND(DEPTNO)>HEXTORAW('3EE0FF')				
Filter Predicates				
SYS_OP_UNDESCEND(SYS_OP_DESCEND(DEPTNO))<30				

Index Range Scan: Function-Based

Index Range SCAN IX_FBI

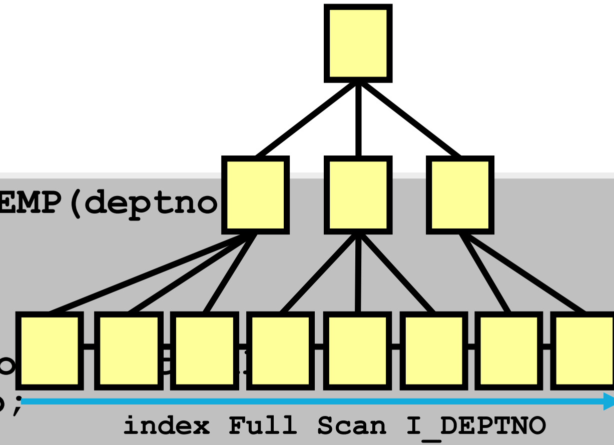


```
create index IX_FBI on EMP(UPPER(ename));

select * from emp where upper(ENAME) like 'A%';
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	1
TABLE ACCESS	EMP	BY INDEX RO...	2	1
INDEX	IX_FBI	RANGE SCAN	1	1
Access Predicates				
UPPER(ENAME) LIKE 'A%'				
Filter Predicates				
UPPER(ENAME) LIKE 'A%'				

Index Full Scan

```
create index I_DEPTNO on EMP(deptno

select *
from emp
where sal > 1000 and deptno
order by deptno;
```

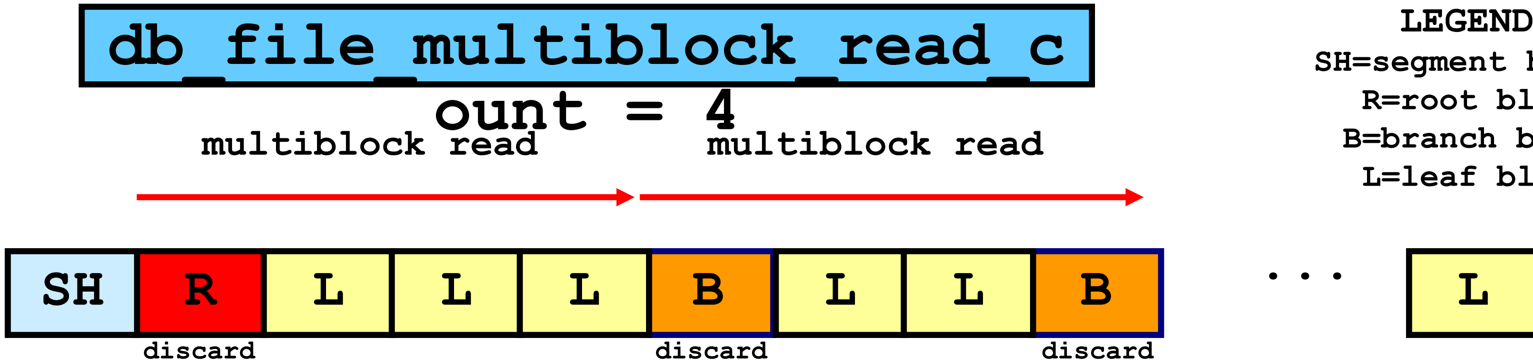


Script Output × | Explain Plan ×

0.012 seconds

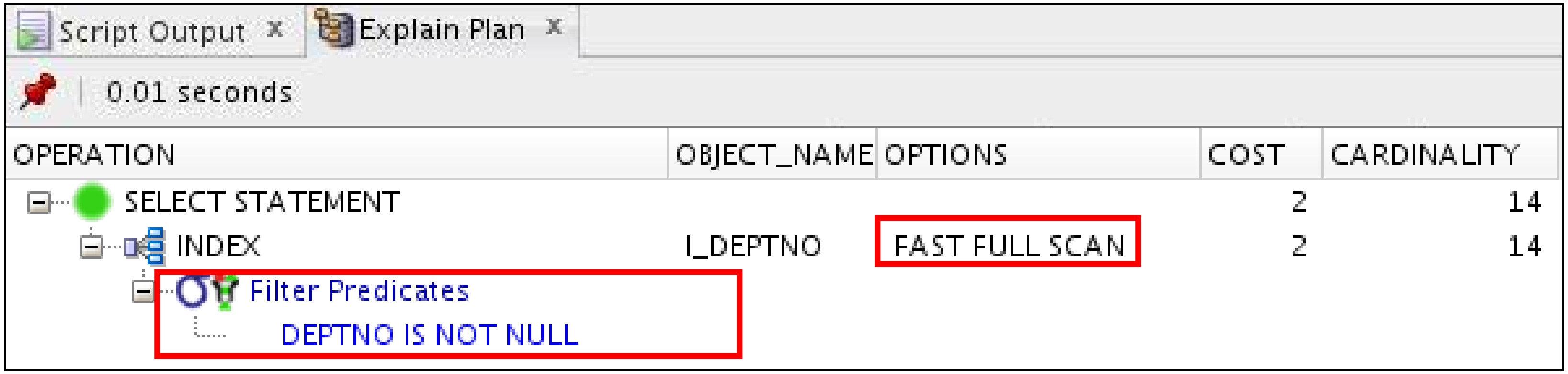
OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	13
TABLE ACCESS	EMP	BY INDEX RO...	2	13
Filter Predicates				
SAL>1000				
INDEX	I_DEPTNO	FULL SCAN	1	14
Filter Predicates				
DEPTNO IS NOT NULL				

Index Fast Full Scan



LEGEND:
SH=segment header
R=root block
B=branch block
L=leaf block

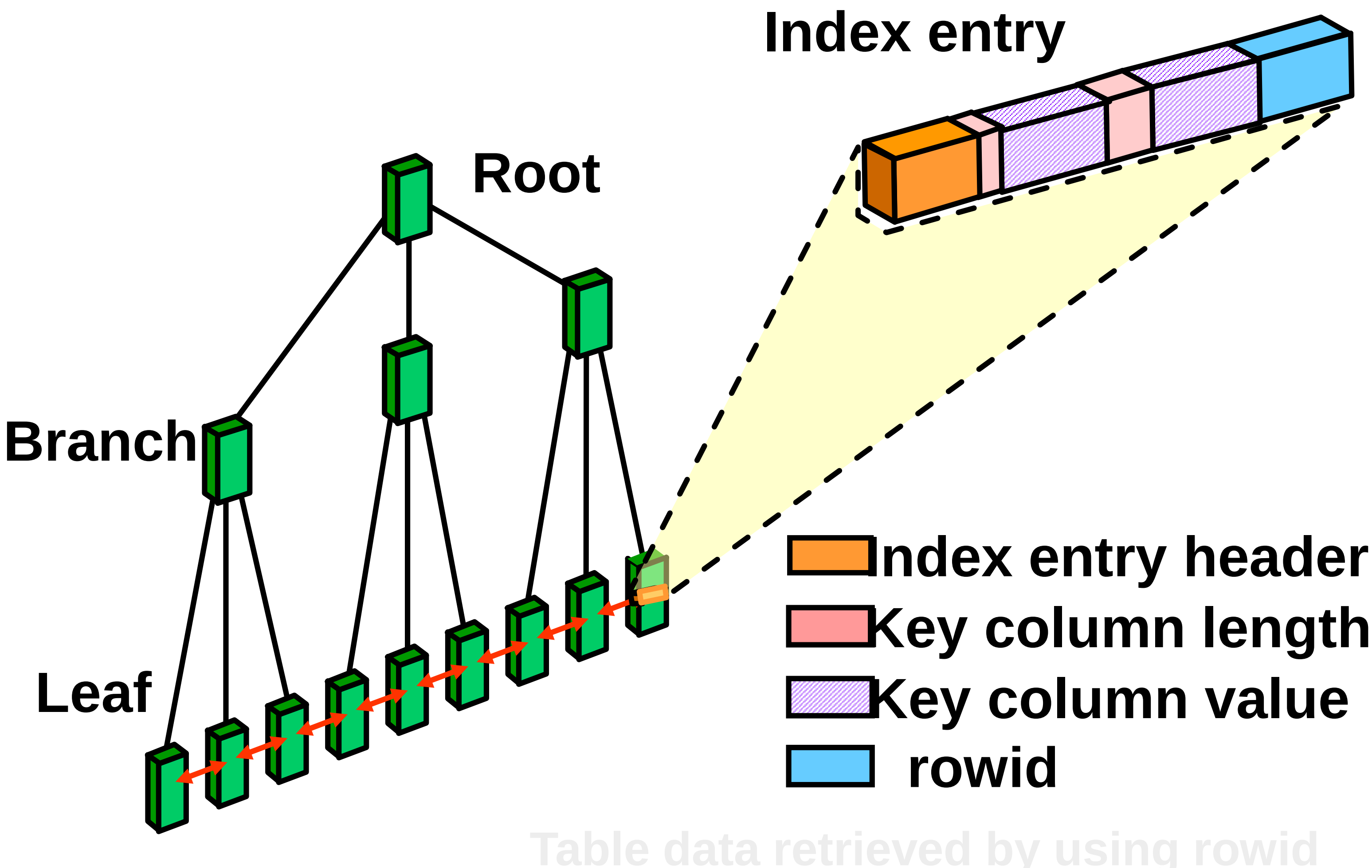
```
select /*+ INDEX_FFS(EMP I_DEPTNO)  */ deptno from emp
where deptno is not null;
```



Script Output | Explain Plan

0.01 seconds

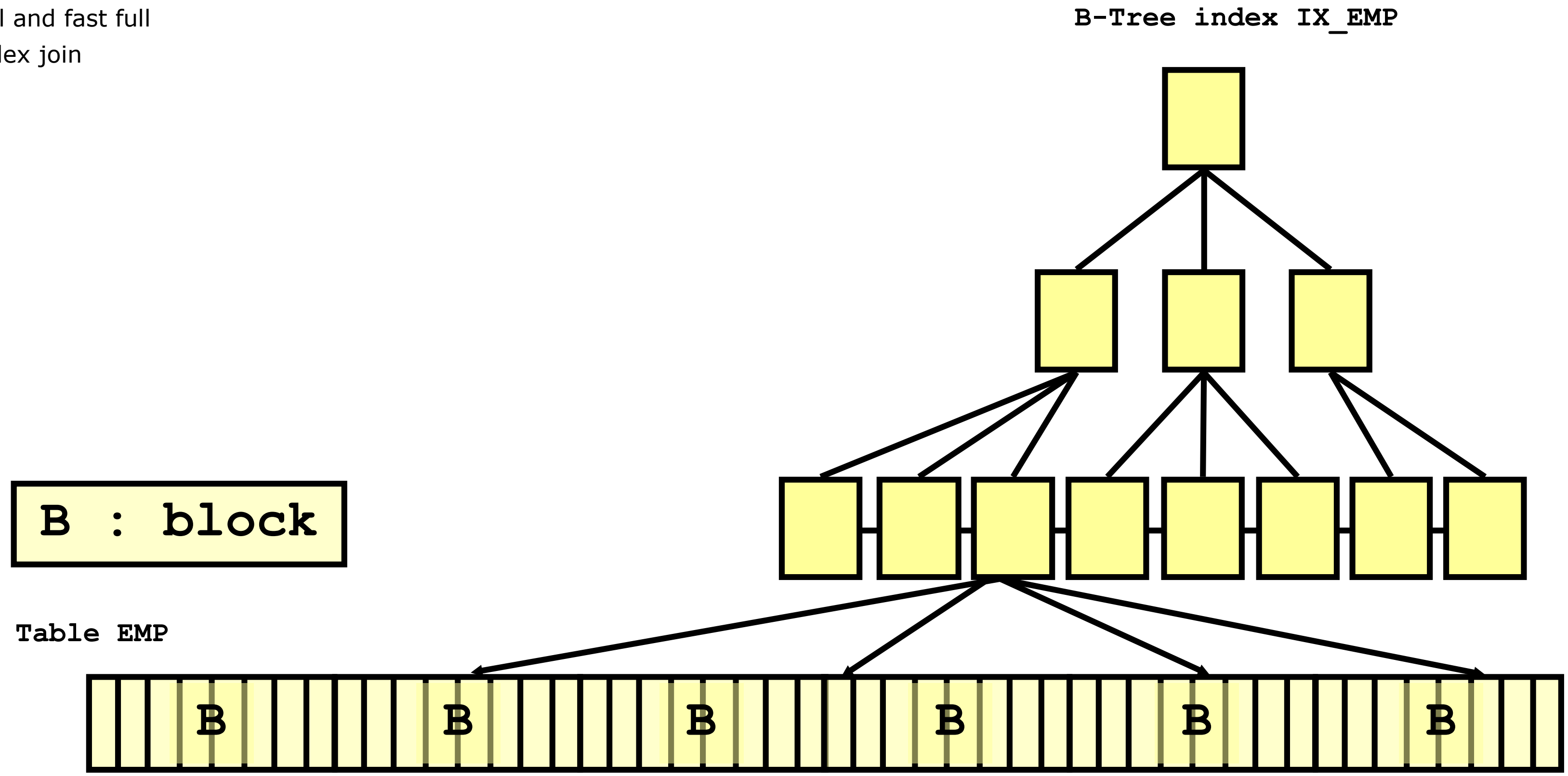
OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	14
INDEX	I_DEPTNO	FAST FULL SCAN	2	14
Filter Predicates				
DEPTNO IS NOT NULL				


Index entry
Root
Branch
Leaf
Index entry header
Key column length
Key column value
rowid
Table data retrieved by using rowid

Index Scans

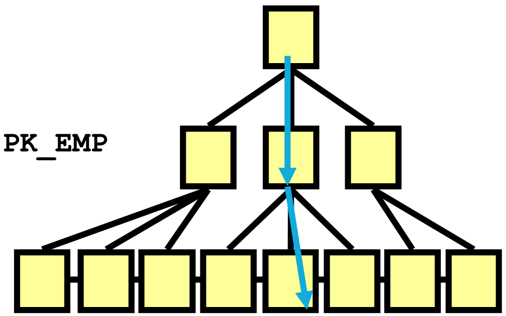
➢Types of index scans:

- Unique
- Min/Max
- Range (Descending)
- Skip
- Full and fast full
- Index join

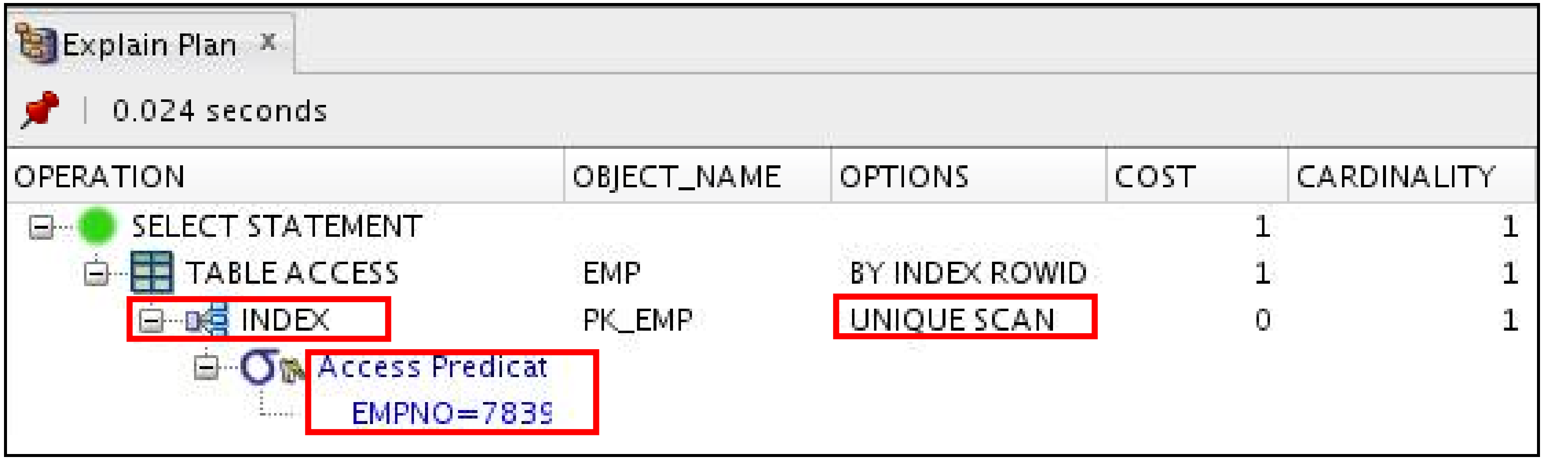


Index Unique Scan

index UNIQUE Scan PK_EMP



```
create unique index PK_EMP on EMP(empno)

select * from emp where empno = 9999;
```



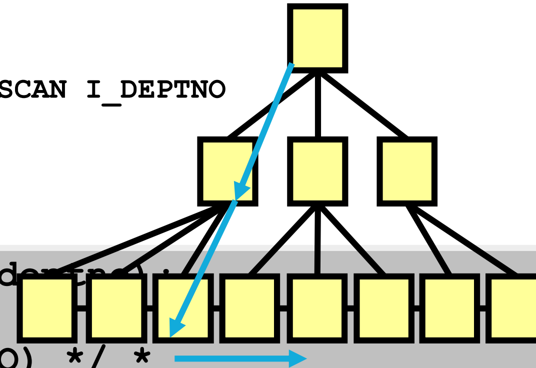
Explain Plan

0.024 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			1	1
TABLE ACCESS	EMP	BY INDEX ROWID	1	1
INDEX	PK_EMP	UNIQUE SCAN	0	1
Access Predicat				
EMPNO=7839				

Index Range Scan

Index Range SCAN I_DEPTNO



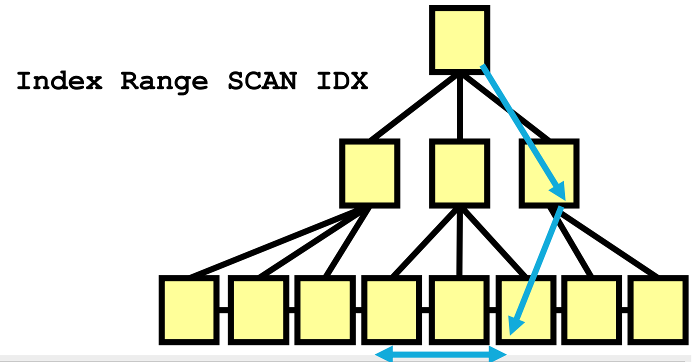
```
create index I_DEPTNO on EMP(deptno);

select /*+ INDEX(EMP I_DEPTNO) */ *
from emp where deptno = 10 and sal > 1000;
```

Script Output × | Explain Plan ×

0.012 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	4
TABLE ACCESS	EMP	BY INDEX ROWID	2	4
Filter Predicates				
SAL>1000				
INDEX	I_DEPTNO	RANGE SCAN	1	5
Access Predicat				
DEPTNO=10				

Index Range Scan: Descending



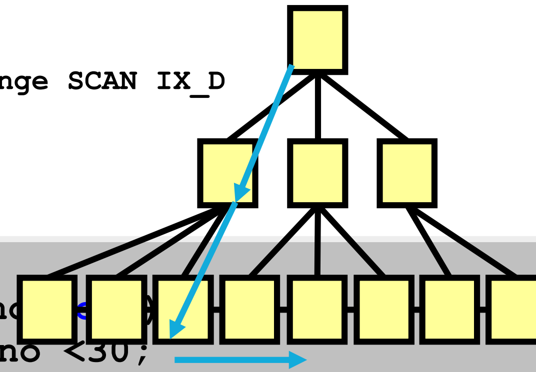

```
select * from emp where deptno>20 order by deptno desc;
```

Script Output × | Explain Plan ×

0.426 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	7
TABLE ACCESS	EMP	BY INDEX ROWID	2	7
INDEX	I_DEPTNO	RANGE SCAN DESCENDING	1	7
Access Predicat				
DEPTNO>20				

Descending Index Range Scan




```
drop index I_Deptno;
create index IX_D on EMP(deptno desc);
select * from emp where deptno <30;
```

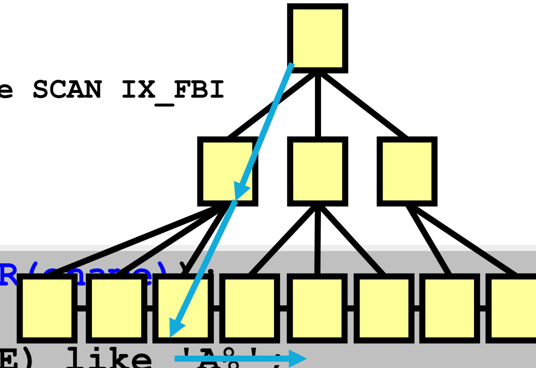
Script Output × | Explain Plan ×

0.011 seconds

OPERATION	OBJECT_NA...	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	9
TABLE ACCESS	EMP	BY INDEX ROWID	2	9
INDEX	IX_D	RANGE SCAN	1	1
Access Predicates				
SYS_OP_DESCEND(DEPTNO)>HEXTORAW('3EE0FF')				
Filter Predicates				
SYS_OP_UNDESCEND(SYS_OP_DESCEND(DEPTNO))<30				

Index Range Scan: Function-Based

Index Range SCAN IX_FBI

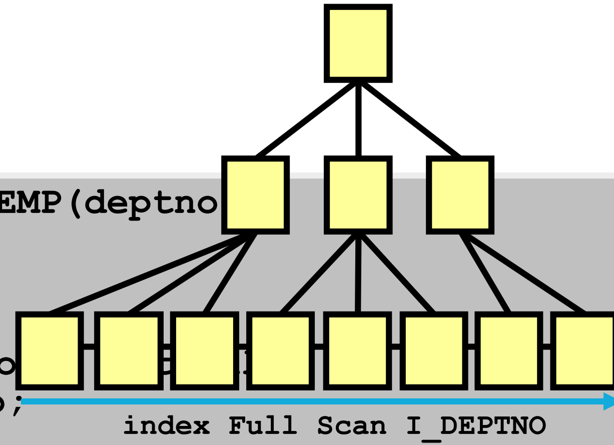


```
create index IX_FBI on EMP(UPPER(ename));

select * from emp where upper(ENAME) like 'A%';
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	1
TABLE ACCESS	EMP	BY INDEX RO...	2	1
INDEX	IX_FBI	RANGE SCAN	1	1
Access Predicates				
UPPER(ENAME) LIKE 'A%'				
Filter Predicates				
UPPER(ENAME) LIKE 'A%'				

Index Full Scan

```
create index I_DEPTNO on EMP(deptno

select *
from emp
where sal > 1000 and deptno
order by deptno;
```

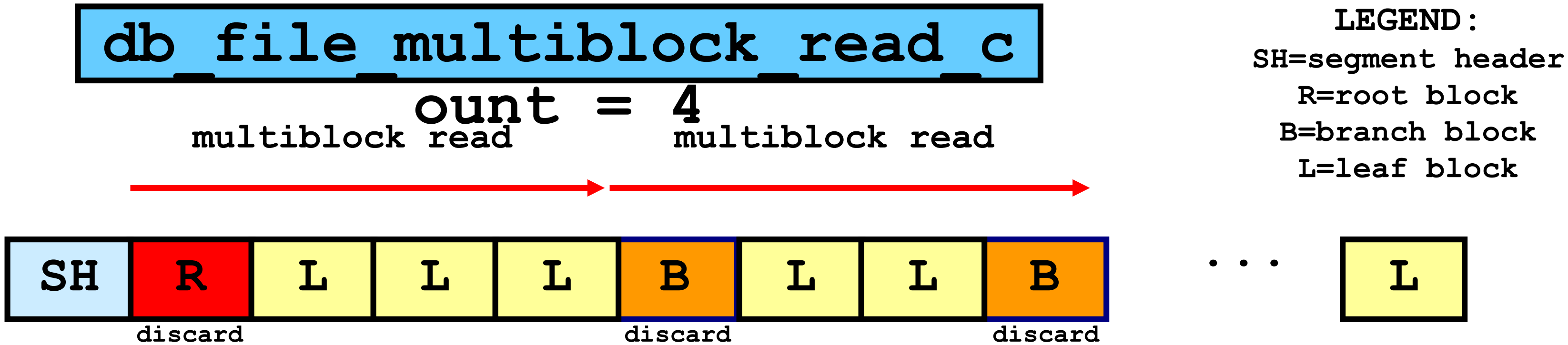


Script Output | Explain Plan

0.012 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	13
TABLE ACCESS	EMP	BY INDEX RO...	2	13
Filter Predicates				
SAL>1000				
INDEX	I_DEPTNO	FULL SCAN	1	14
Filter Predicates				
DEPTNO IS NOT NULL				

Index Fast Full Scan



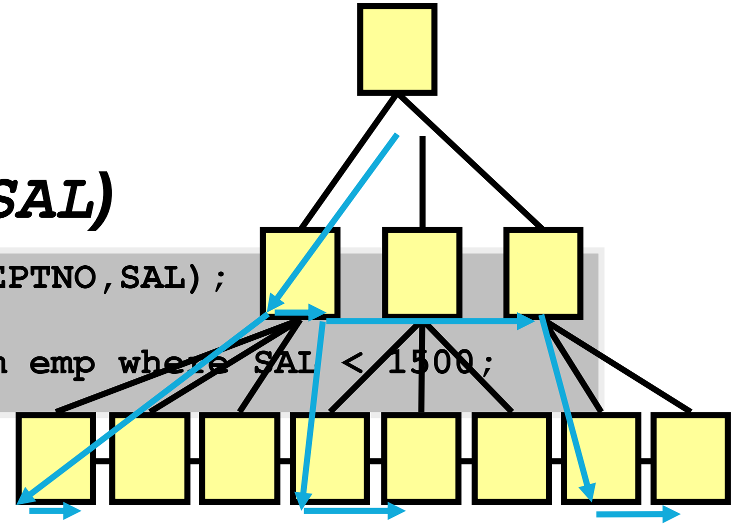

```
select /*+ INDEX_FFS(EMP I_DEPTNO)  */ deptno from emp
where deptno is not null;
```

Script Output | Explain Plan

0.01 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	14
INDEX	I_DEPTNO	FAST FULL SCAN	2	14
Filter Predicates				
DEPTNO IS NOT NULL				

Index Skip Scan: Example



Index on (DEPTNO, SAL)

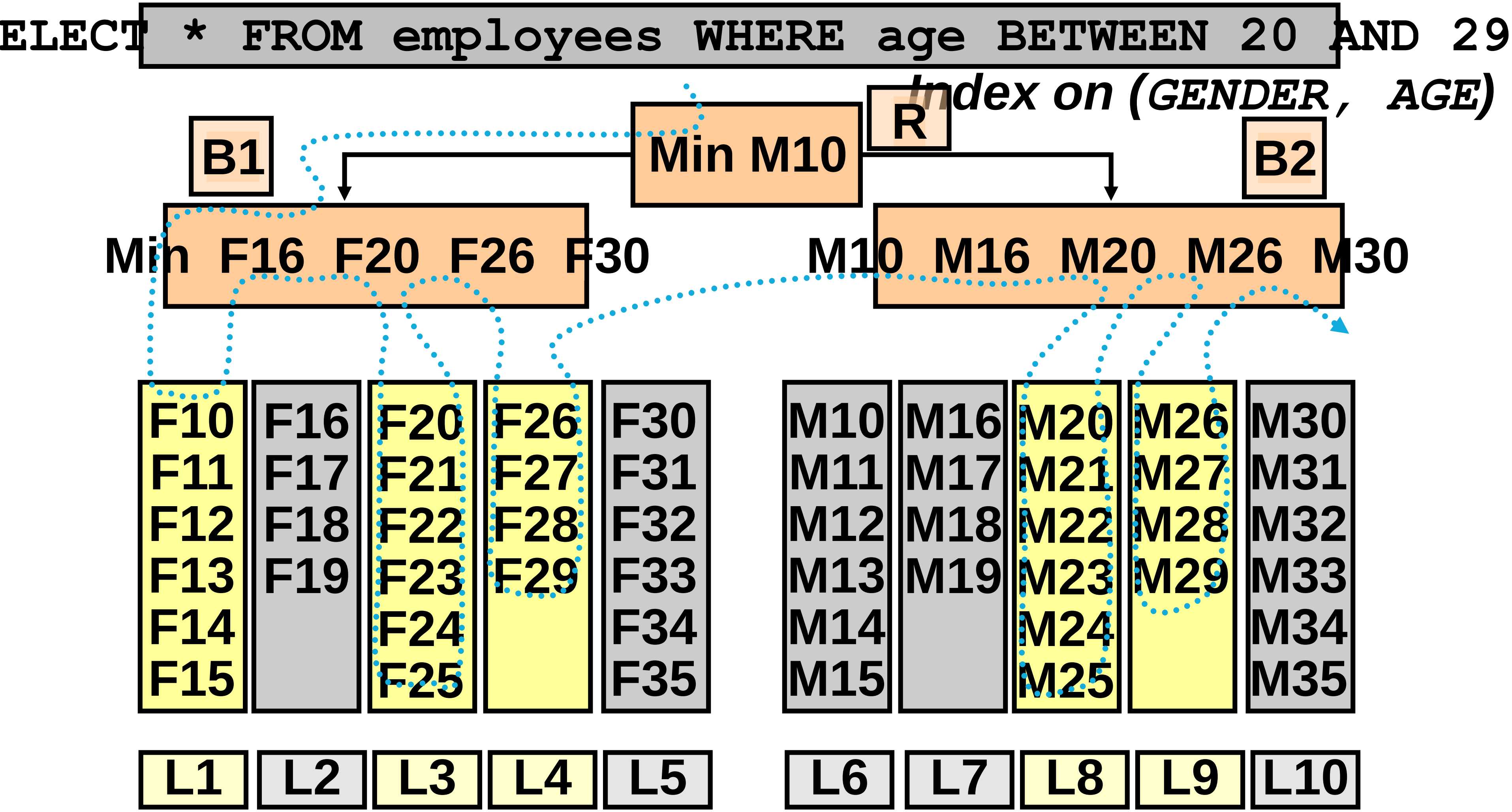
```
create index IX_SS on EMP(DEPTNO,SAL);

select /*+ index_ss(EMP IX_SS) */ * from emp where SAL < 1500;
```

Script Output | Explain Plan

0.01 seconds

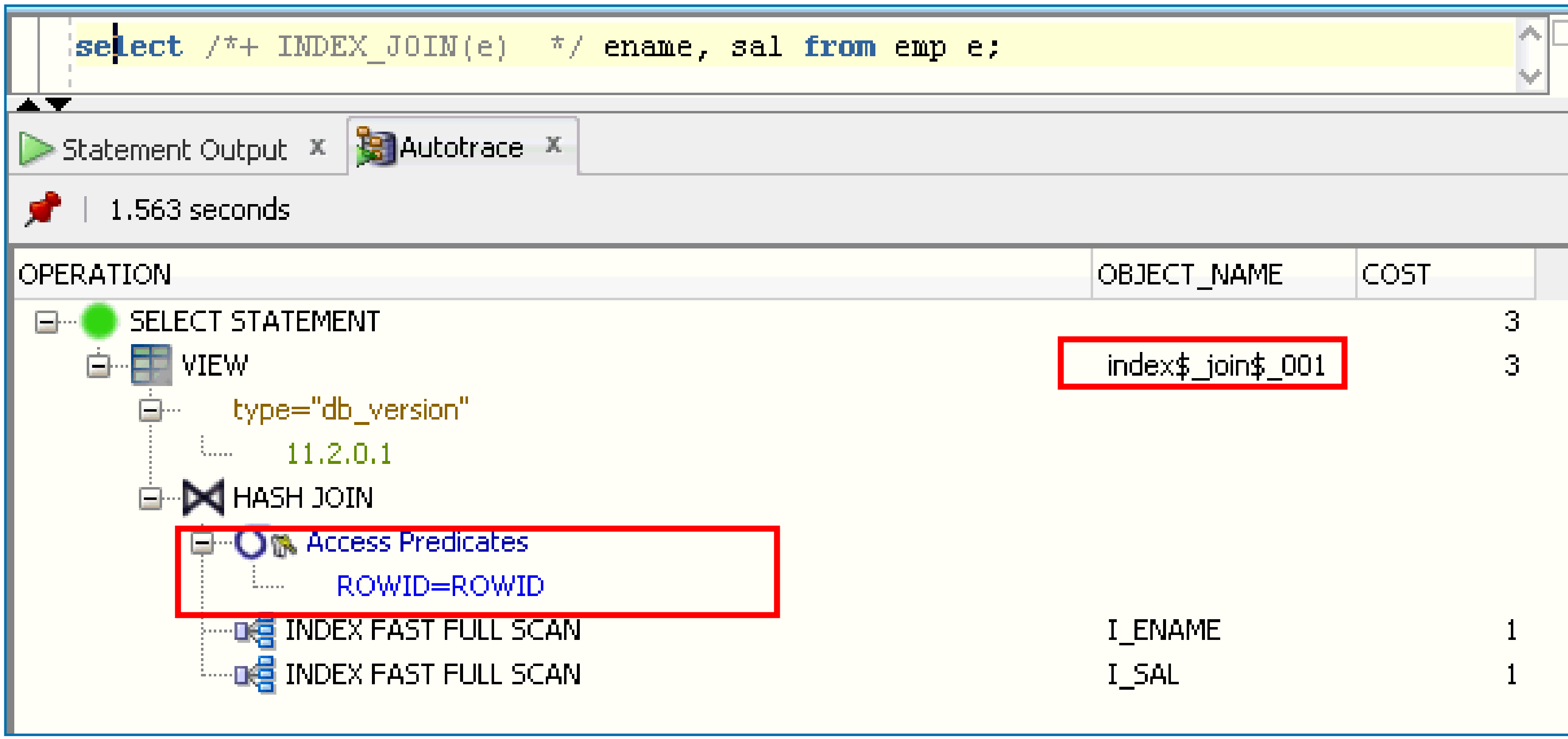
OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			4	2
TABLE ACCESS	EMP	BY INDEX ROWID	4	2
INDEX	IX_SS	SKIP SCAN	3	2
Access Predicates				
SAL<1500				
Filter Predicates				
SAL<1500				

Index Skip Scan



Index Join Scan

```
alter table emp modify (SAL not null, ENAME not null);
create index I_ENAME on EMP(ename);
create index I_SAL on EMP(sal);
```



B*-tree Indexes and Nulls

```
create table nulltest ( col1 number, col2 number not null);
create index nullind1 on nulltest (col1);
create index notnullind2 on nulltest (col2);
```

```
select /*+ index(t nullind1) */ col1 from nulltest t;
```

Script Output × | Explain Plan ×

0.864 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	1
TABLE ACCESS	NULLTEST	FULL	2	1

```
select col1 from nulltest t where col1=10;
```

Script Output × | Explain Plan ×

1.105 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			1	1
INDEX	NULLIND1	RANGE SCAN	1	1
Access Predicates				
COL1=10				

```
select /*+ index(t notnullind2) */ col2 from nulltest t;
```

Script Output × | Explain Plan ×

0.034 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			1	1
INDEX	NOTNULLIND2	FULL SCAN	1	1

Using Indexes: Considering Nullable Columns

Column	Null?
SSN	Y
FNAME	Y
LNAME	N
⋮	

PERSON

```
CREATE UNIQUE INDEX person_ssn_ix
ON person(ssn);
```

```
SELECT COUNT(*) FROM person;

SELECT STATEMENT    |
 SORT AGGREGATE     |
TABLE ACCESS FULL| PERSON
```

```
DROP INDEX person_ssn_ix;
```

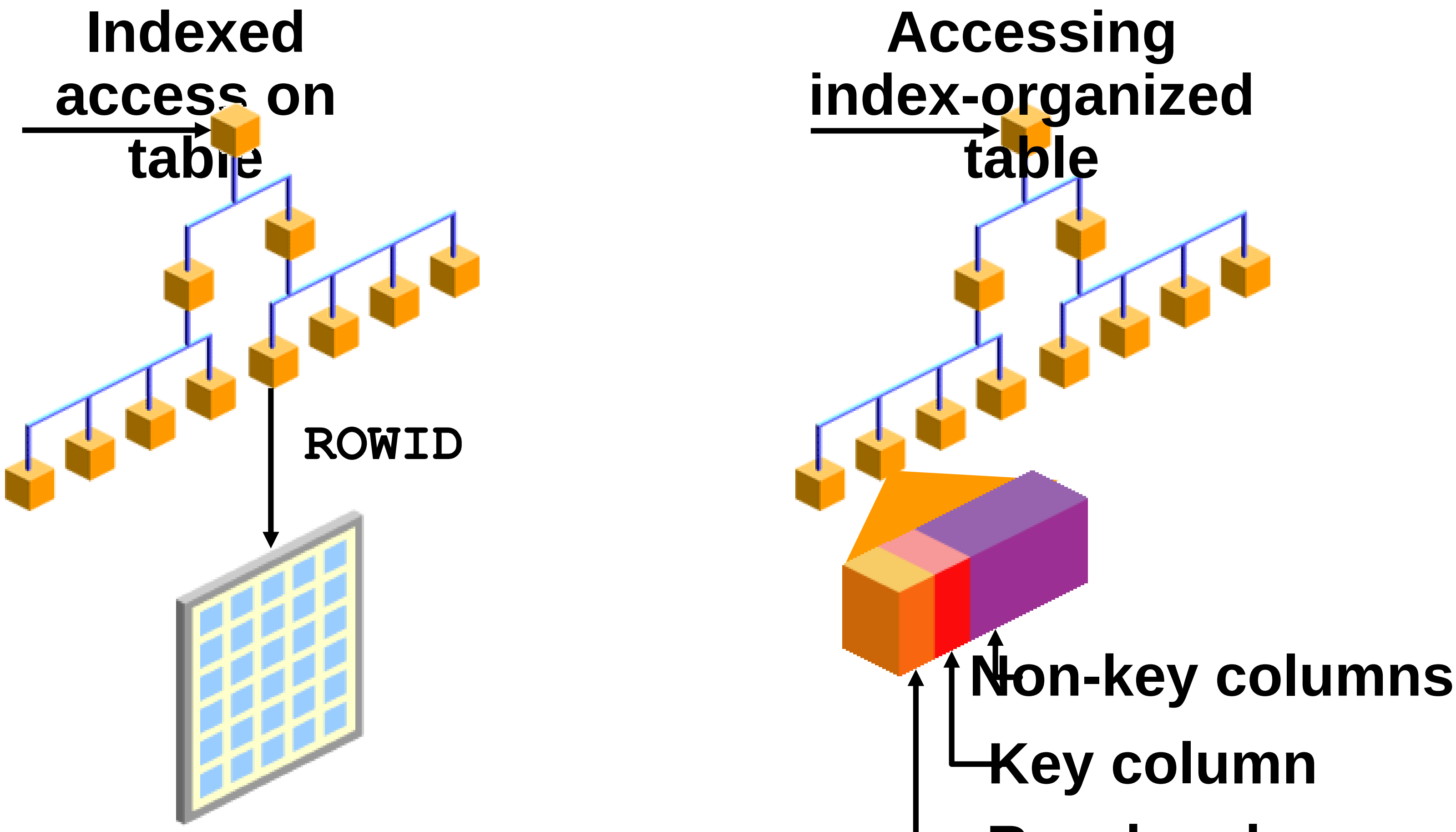
Column	Null?
SSN	N
FNAME	Y
LNAME	N
⋮	

PERSON

```
ALTER TABLE person ADD CONSTRAINT pk_ssn
PRIMARY KEY (ssn);
```

```
SELECT /*+ INDEX(person) */ COUNT(*) FROM
person;

SELECT STATEMENT       |
 SORT AGGREGATE        |
INDEX FAST FULL SCAN| PK_SSN
```

Index-Organized Tables



Index-Organized Table Scans

```
create table iotemp
( empno number(4) primary key, ename varchar2(10) not null,
job varchar2(9), mgr number(4), hiredate date,
sal number(7,2) not null,  comm number(7,2), deptno number(2))
organization index;
```

```
select * from iotemp where empno=9999;
```

Script Output × | Explain Plan ×

0.734 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			1	1
INDEX	SYS_IOT_TOP_7...	UNIQUE SCAN	1	1
Access Predicates				
EMPNO=9999				

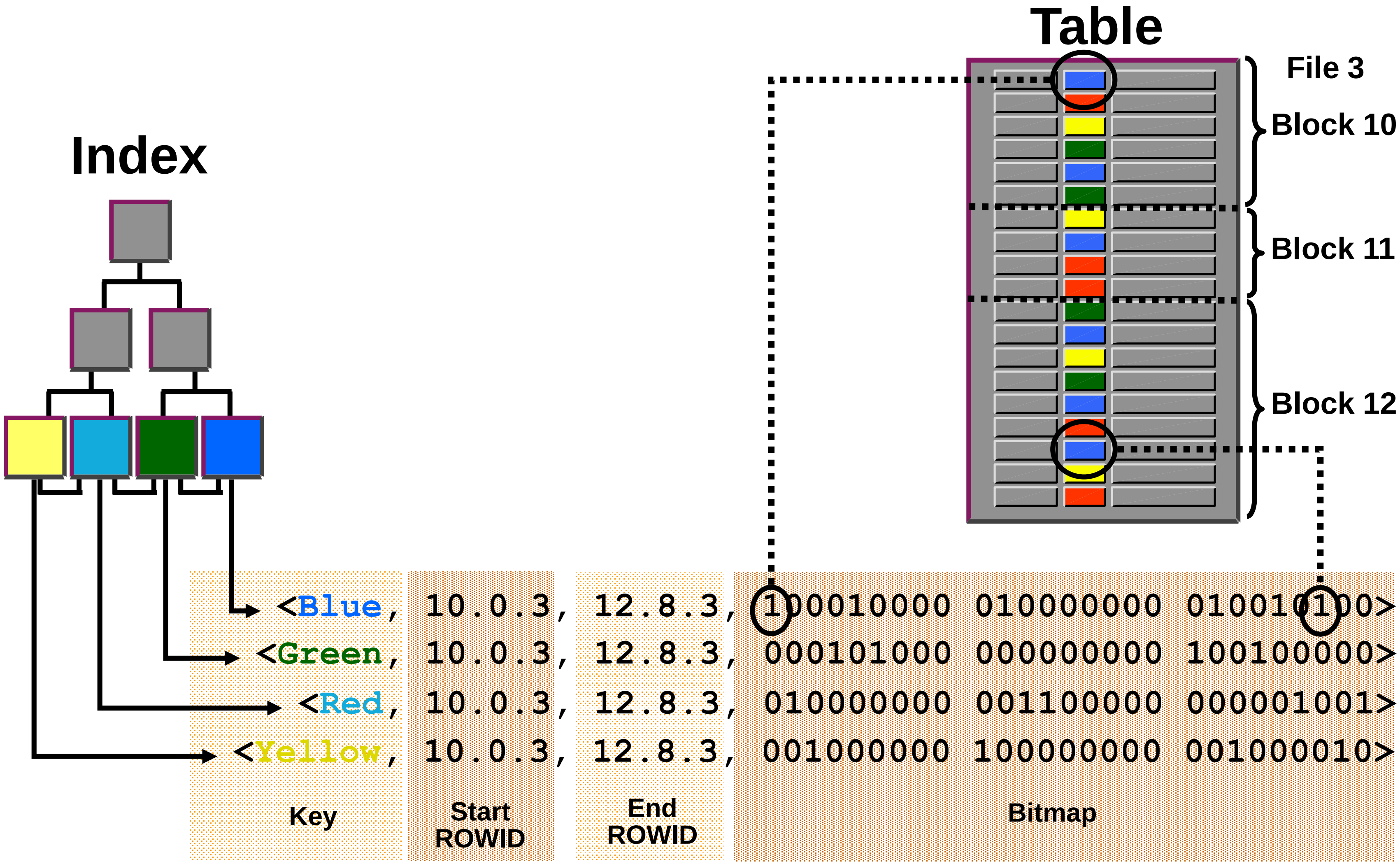
```
select * from iotemp where sal>1000;
```

Script Output × | Explain Plan ×

0.007 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALITY
SELECT STATEMENT			2	1
INDEX	SYS_IOT_TOP_7...	FAST FULL SCAN	2	1
Filter Predicates				
SAL>1000				

Bitmap Indexes



Bitmap Index Access: Examples

```
SELECT * FROM PERF_TEAM WHERE country='FR';
---------------------------------------------------------------------
| Id | Operation                     | Name      | Rows | Bytes |
---------------------------------------------------------------------
|  0 | SELECT STATEMENT              |           |    1 |    45 |
|  1 |  TABLE ACCESS BY INDEX ROWID  | PERF_TEAM |    1 |    45 |
|  2 |   BITMAP CONVERSION TO ROWIDS |           |      |       |
|  3 |    BITMAP INDEX SINGLE VALUE  | IX_B2     |      |       |
---------------------------------------------------------------------
Predicate:  3 - access("COUNTRY"='FR')
```

```
SELECT * FROM PERF_TEAM WHERE country>'FR';
---------------------------------------------------------------------
| Id | Operation                     | Name      | Rows | Bytes |
---------------------------------------------------------------------
|  0 | SELECT STATEMENT              |           |    1 |    45 |
|  1 |  TABLE ACCESS BY INDEX ROWID  | PERF_TEAM |    1 |    45 |
|  2 |   BITMAP CONVERSION TO ROWIDS |           |      |       |
|  3 |    BITMAP INDEX RANGE SCAN    | IX_B2     |      |       |
---------------------------------------------------------------------
Predicate: 3 - access("COUNTRY">'FR') filter("COUNTRY">'FR')
```

Combining Bitmap Indexes: Examples

```
SELECT * FROM PERF_TEAM WHERE country in('FR','DE');
```

FR	0	0	1	1	1	1	0	0	0	0	0	0

OR

DE	0	1	0	0	0	0	0	0	0	0	0	0

0	1	1	1	1	1	0	0	0	0	0

F	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**

AND

M	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**1**

0	0	1	0	1	1	0	0	0	0	0

```
SELECT * FROM EMEA_PERF_TEAM T WHERE country='FR' and gender='M';
```

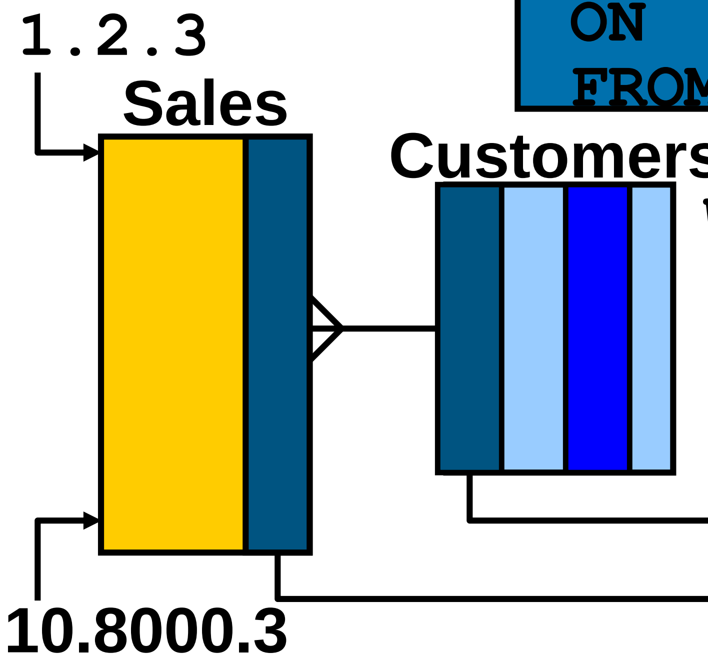
Combining Bitmap Index Access Paths

```
SELECT * FROM PERF_TEAM WHERE country in ('FR','DE');
---------------------------------------------------------------------
| Id  | Operation                      | Name      | Rows  | Bytes |
|   0 | SELECT STATEMENT               |           |     1 |    45 |
|   1 |  INLIST ITERATOR               |           |       |       |
|   2 |   TABLE ACCESS BY INDEX ROWID  | PERF_TEAM |     1 |    45 |
|   3 |    BITMAP CONVERSION TO ROWIDS |           |       |       |
|   4 |     BITMAP INDEX SINGLE VALUE  | IX_B2     |       |       |
Predicate: 4 - access("COUNTRY"='DE' OR "COUNTRY"='FR')
```

```
SELECT * FROM PERF_TEAM WHERE country='FR' and gender='M';
---------------------------------------------------------------------
| Id  | Operation                      | Name      | Rows  | Bytes |
|   0 | SELECT STATEMENT               |           |     1 |    45 |
|   1 |  TABLE ACCESS BY INDEX ROWID   | PERF_TEAM |     1 |    45 |
|   2 |   BITMAP CONVERSION TO ROWIDS  |           |       |       |
|   3 |    BITMAP AND                  |           |       |       |
|   4 |     BITMAP INDEX SINGLE VALUE  | IX_B1     |       |       |
|   5 |     BITMAP INDEX SINGLE VALUE  | IX_B2     |       |       |
Predicate: 4 - access("GENDER"='M') 5 - access("COUNTRY"='FR')
```

Bitmap Operations

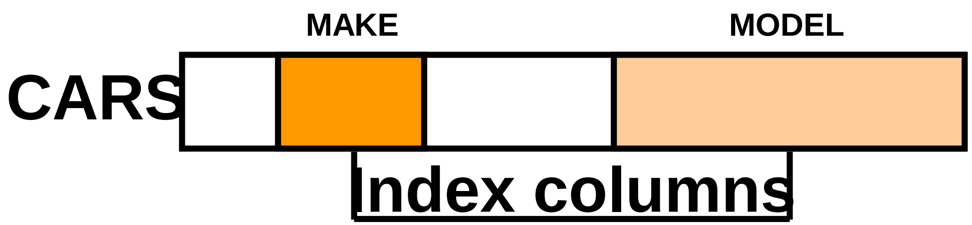
- BITMAP CONVERSION:
  - TO ROWIDS
  - FROM ROWIDS
  - COUNT
- BITMAP INDEX:
  - SINGLE VALUE
  - RANGE SCAN
  - FULL SCAN
- BITMAP MERGE
- BITMAP AND/OR
- BITMAP MINUS
- BITMAP KEY ITERATION




```
CREATE BITMAP INDEX
cust_sales_bji
ON     sales(c.cust_city)
FROM   sales s, customers
                c
WHERE  c.cust_id =
       s.cust_id;
```

```
<Rognes, 1.2.3, 10.8000.3,
         1000100100010100…>
<Aix-en-Provence, 1.2.3, 10.8000.3,
         000101000100000…>
<Marseille, 1.2.3, 10.8000.3,
```

Composite Indexes

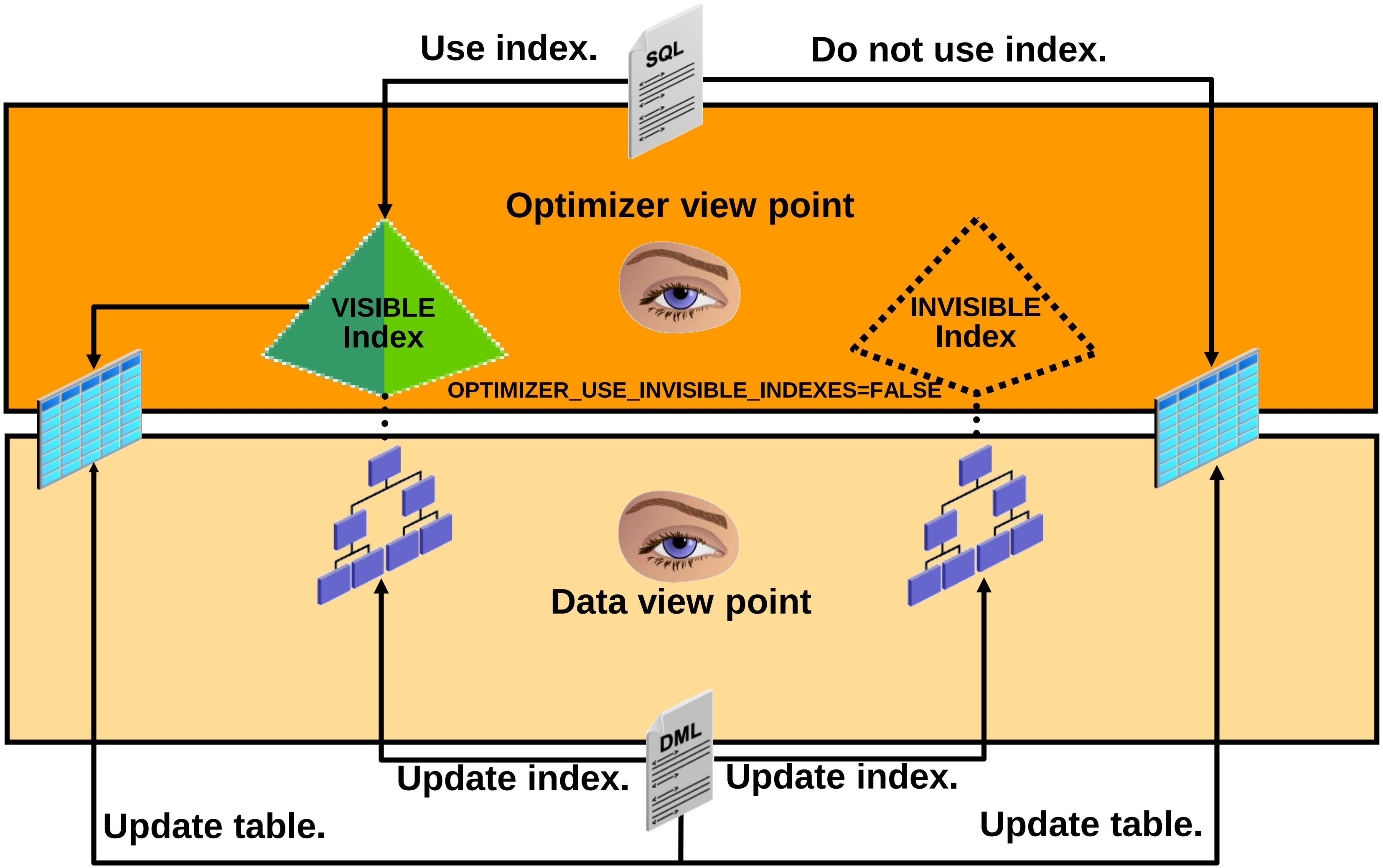



```
create index cars_make_model_idx on
cars(make, model);
```

```
select *
from cars
where make = 'CITROËN' and model = '2CV';
```

```
-----------------------------------------------------------------
| Id  | Operation                    | Name                       |
-----------------------------------------------------------------
|   0 | SELECT STATEMENT             |                            |
|   1 |  TABLE ACCESS BY INDEX ROWID| CUSTOMERS                  |
|*  2 |   INDEX RANGE SCAN           | CARS_MAKE_MODEL_IDX        |
-----------------------------------------------------------------
```

Invisible Index: Overview



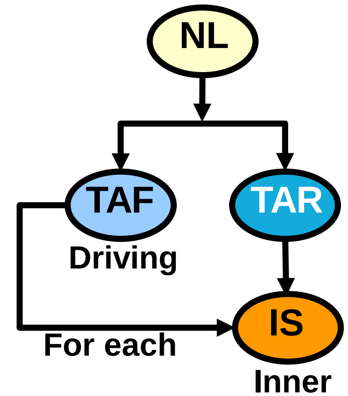
Join Methods

- A join:
  - Defines the relationship between two row sources
  - Is a method of combining data from two data sources
  - Is controlled by join predicates, which define how the objects are related
  - Join methods:
    - Nested loops
    - Sort-merge join
    - Hash join

```
SELECT e.ename, d.dname
FROM dept d JOIN emp e USING (deptno)      ← Join predicate
WHERE e.job = 'ANALYST' OR e.empno = 9999; ← Nonjoin predicate
```

```
SELECT e.ename,d.dname
FROM   emp e, dept d
WHERE  e.deptno = d.deptno AND                 ← Join predicate
(e.job = 'ANALYST' OR e.empno = 9999);         ← Nonjoin predicate
```

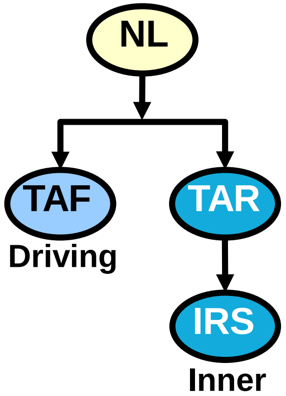
Nested Loops Join



- Driving row source is scanned.
- Each row returned drives a lookup in inner row source.
- Joining rows are then returned.

```
select ename, e.deptno, d.deptno, d.dname
from emp e, dept d
where e.deptno = d.deptno and ename like 'A%';
---------------------------------------------------------------------
| Id  | Operation                   |  Name        | Rows   |Cost   |
---------------------------------------------------------------------
|   0 | SELECT STATEMENT            |              |      2 |      4 |
|   1 |  NESTED LOOPS               |              |      2 |      4 |
|   2 |   TABLE ACCESS FULL         | EMP          |      2 |      2 |
|   3 |   TABLE ACCESS BY INDEX ROWID| DEPT        |      1 |      1 |
|   4 |    INDEX UNIQUE SCAN        | PK_DEPT      |      1 |        |
---------------------------------------------------------------------
2 - filter("E"."ENAME" LIKE 'A%')
4 - access("E"."DEPTNO"="D"."DEPTNO")
```

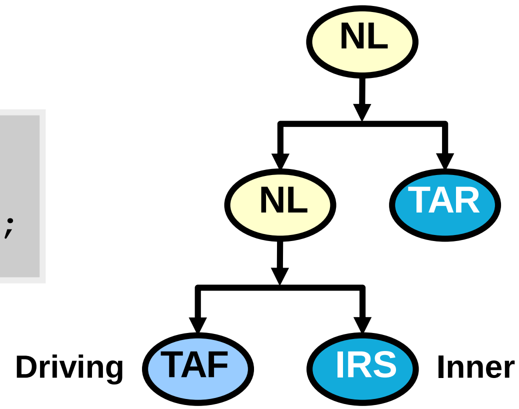
Nested Loops Join: Prefetching




```
select ename, e.deptno, d.deptno, d.dname
from emp e, dept d
where e.deptno = d.deptno and ename like 'A%';
---------------------------------------------------------------------
|    0 | SELECT STATEMENT              |       |     2 |    84 |     5
|    1 |  TABLE ACCESS BY INDEX ROWID| DEPT  |     1 |    22 |     1
|    2 |   NESTED LOOPS                |       |     2 |    84 |     5
|*   3 |    TABLE ACCESS FULL          | EMP   |     2 |    40 |     3
|*   4 |    INDEX RANGE SCAN           | IDEPT |     1 |       |     0
---------------------------------------------------------------------
3 - filter("E"."ENAME" LIKE 'A%')
4 - access("E"."DEPTNO"="D"."DEPTNO")
```

Nested Loops Join: 11*g* Implementation

```
select ename, e.deptno, d.deptno, d.dname
from emp e, dept d
where e.deptno = d.deptno and ename like 'A%';
```



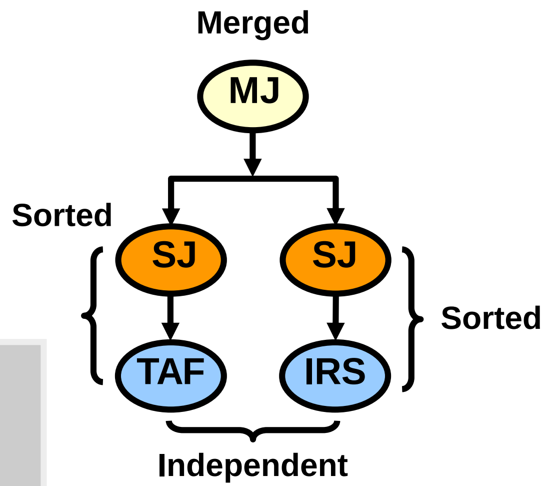
Explain Plan

0.299 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALI...
SELECT STATEMENT			4	1
NESTED LOOPS				
NESTED LOOPS			4	1
TABLE ACCESS	EMP	FULL	3	1
Filter Predicates				
ENAME LIKE 'A%'				
INDEX	PK_DEPT	UNIQUE SCAN	0	1
Access Predicates				
E.DEPTNO=D.DEPTNO				
TABLE ACCESS	DEPT	BY INDEX RO...	1	1

Sort Merge Join

- First and second row sources are sorted by the same sort key.
- Sorted rows from both tables are merged.

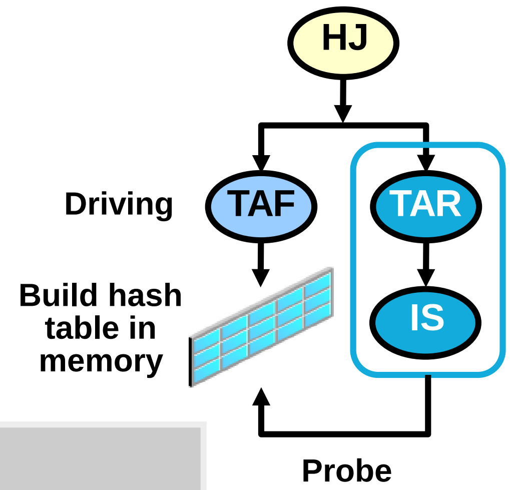



```
select /*+ USE_MERGE(d e) NO_INDEX(d) */
       ename, e.deptno, d.deptno, dname
            from emp e, dept d
where e.deptno = d.deptno and ename > 'A'
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALI...
SELECT STATEMENT			8	14
MERGE JOIN			8	14
SORT		JOIN	4	4
TABLE ACCESS	DEPT	FULL	3	4
SORT		JOIN	4	14
Access Predicates				
AND				
E.DEPTNO=D.DEPTNO				
SYS_OP_DESCEND(E.DEPT				
Filter Predicates				
E.DEPTNO=D.DEPTNO				
TABLE ACCESS	EMP	FULL	3	14
Filter Predicates				
ENAME>'A'				

Hash Join

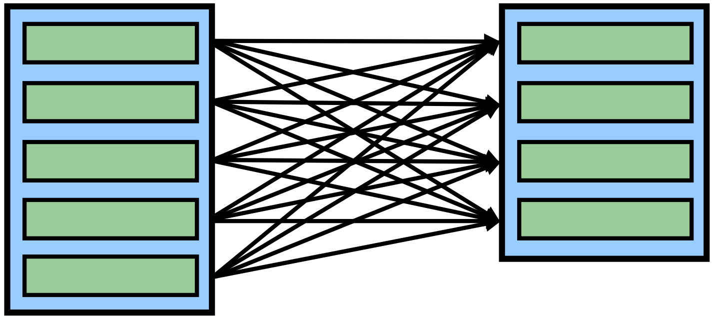
- The smallest row source is used to build a hash table.
- The second row source is hashed and checked against the hash table.




```
select /*+ USE_HASH(e d) */
ename, e.deptno, d.deptno, dname
from emp e, dept d
where e.deptno = d.deptno and ename like 'A%'
```

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALI...
SELECT STATEMENT			7	1
HASH JOIN			7	1
Access Predicates				
AND				
E.DEPTNO=D.DEPTNO				
SYS_OP_DESCEND(E.DEPTNO)				
TABLE ACCESS	EMP	FULL	3	1
Filter Predicates				
ENAME LIKE 'A%'				
TABLE ACCESS	DEPT	FULL	3	4

Cartesian Join



```sql
select ename, e.deptno, d.deptno, dname
from emp e, dept d where ename like 'A%';
```

Explain Plan

0.295 seconds

OPERATION	OBJECT_NAME	OPTIONS	COST	CARDINALI...
SELECT STATEMENT			6	4
MERGE JOIN		CARTESIAN	6	4
TABLE ACCESS	EMP	FULL	3	1
Filter Predicates				
ENAME LIKE 'A%'				
BUFFER		SORT	3	4
TABLE ACCESS	DEPT	FULL	3	4

Join Types

- A join operation combines the output from two row sources and returns one resulting row source.
- Join operation types include the following :
  - Join (Equijoin/Natural – Nonequijoin)
  - Outer join (Full, Left, and Right)
  - Semi join: `EXISTS` subquery
  - Anti join: `NOT IN` subquery
  - Star join (Optimization)

Equijoins and Nonequijoins

```sql
SELECT e.ename, e.sal, s.grade
  FROM   emp e ,salgrade s
  WHERE  e.sal = s.hisal;
```

Equijoin

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
HASH JOIN		
Access Predicates		
E.SAL=S.HISAL		
TABLE ACCESS	SALGRADE	FULL
TABLE ACCESS	EMP	FULL

Nonequijoin

```sql
SELECT e.ename, e.sal, s.grade
  FROM   emp e ,salgrade s
WHERE  e.sal between s.hisal and s.hisal;
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
MERGE JOIN		
SORT		JOIN
TABLE ACCESS	SALGRADE	FULL
FILTER		
Filter Predicates		
E.SAL<=S.HISAL		
SORT		JOIN
Access Predicates		
E.SAL>=S.HISAL		
Filter Predicates		
E.SAL>=S.HISAL		
TABLE ACCESS	EMP	FULL

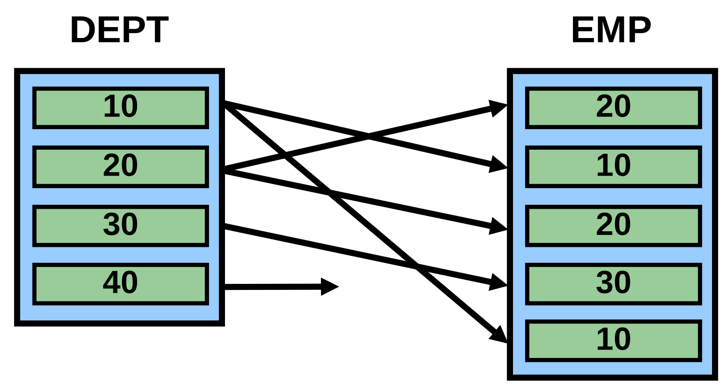
Outer Joins

➢An outer join returns a row even if no match is found.

```sql
SELECT d.deptno,d.dname,e.empno,e.ename
FROM   emp e, dept d
WHERE  e.deptno(+)=d.deptno;
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
MERGE JOIN		OUTER
TABLE ACCESS	DEPT	BY INDEX RO
INDEX	PK_DEPT	FULL SCAN
SORT		JOIN
Access Predicates		
E.DEPTNO(+)=D.DEPTNO		
Filter Predicates		
E.DEPTNO(+)=D.DEPTNO		
TABLE ACCESS	EMP	FULL

1




```sql
SELECT /*+ USE_HASH(e d) */
d.deptno,d.dname,e.empno,e.ename
FROM   emp e, dept d
WHERE  e.deptno(+)=d.deptno;
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
HASH JOIN		OUTER
Access Predicates		
E.DEPTNO(+)=D.DEPTNO		
TABLE ACCESS	DEPT	FULL
TABLE ACCESS	EMP	FULL

2

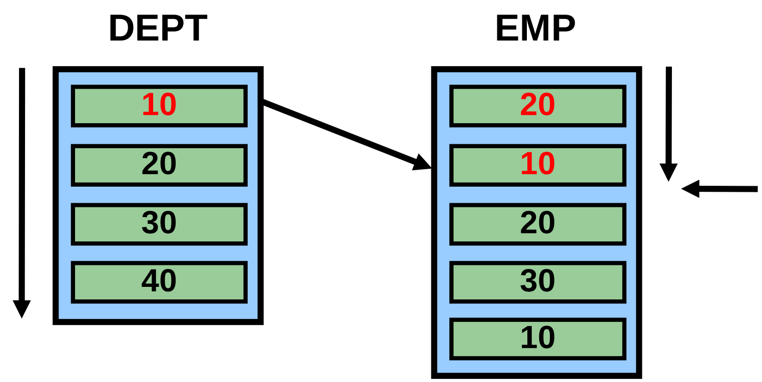
```sql
SELECT /*+ USE_NL(e d) */
d.deptno,d.dname,e.empno,e.ename
FROM   emp e, dept d
WHERE  e.deptno(+)=d.deptno;
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
NESTED LOOPS		OUTER
TABLE ACCESS	DEPT	FULL
TABLE ACCESS	EMP	FULL
Filter Predicates		
E.DEPTNO(+)=D.DEPTNO		

3

Semijoins

- Semijoins look only for the first match.




```
SELECT deptno, dname
FROM   dept
WHERE  EXISTS (SELECT 1 FROM emp WHERE emp.deptno=dept.deptno);
```

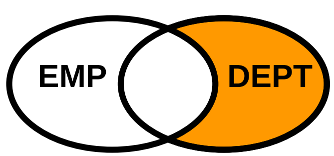
OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
MERGE JOIN		SEMI
TABLE ACCESS	DEPT	BY INDEX RO.
INDEX	PK_DEPT	FULL SCAN
SORT		UNIQUE
Access Predicates		
EMP.DEPTNO=DEPT.DEPTNO		
Filter Predicates		
EMP.DEPTNO=DEPT.DEPTNO		
TABLE ACCESS	EMP	FULL

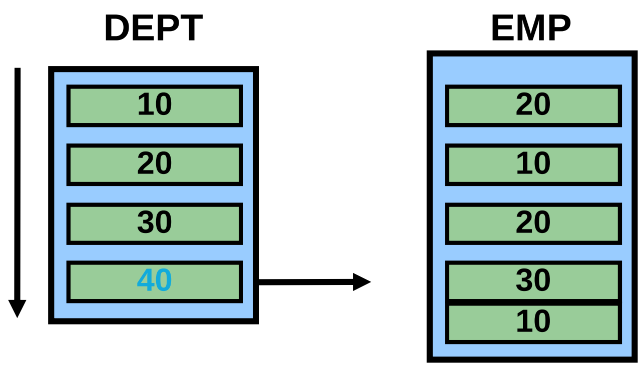
Antijoins

- Reverse of what would have been returned by a join

```sql
SELECT deptno, dname
   FROM   dept
WHERE  deptno not in
 (SELECT deptno FROM emp);
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
MERGE JOIN		ANTI NA
SORT		JOIN
TABLE ACCESS	DEPT	BY INDEX RO..
INDEX	PK_DEPT	FULL SCAN
SORT		UNIQUE
Access Predicates		
DEPTNO=DEPTNO		
Filter Predicates		
DEPTNO=DEPTNO		
TABLE ACCESS	EMP	FULL






```sql
SELECT deptno, dname FROM dept
WHERE deptno IS NOT NULL AND
deptno NOT IN
(SELECT /*+ HASH_AJ */ deptno FROM
emp WHERE deptno IS NOT NULL);
```

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
HASH JOIN		ANTI
Access Predicates		
DEPTNO=DEPTNO		
TABLE ACCESS	DEPT	FULL
TABLE ACCESS	EMP	FULL
Filter Predicates		
DEPTNO IS NOT NULL		

Invisible Indexes: Examples

- Index is altered as not visible to the optimizer:

- Op

```
ALTER INDEX ind1 INVISIBLE;
```

- Optimizer considers this index:

- Cre

```
SELECT /*+ index(TAB1 IND1) */ COL1 FROM
TAB1 WHERE …;
```

```
ALTER INDEX ind1 VISIBLE;
```

```
CREATE INDEX IND1 ON TAB1(COL1) INVISIBLE;
```

Guidelines for Managing Indexes

- Create indexes after inserting table data.
- Index the correct tables and columns.
- Order index columns for performance.
- Limit the number of indexes for each table.
- Drop indexes that are no longer required.
- Specify the tablespace for each index.
- Consider parallelizing index creation.
- Consider creating indexes with `NOLOGGING`.
- Consider costs and benefits of coalescing or rebuilding indexes.
- Consider cost before disabling or dropping constraints.

Investigating Index Usage

- An index may not be used for one of many reasons:
  - There are functions being applied to the predicate.
  - There is a data type mismatch.
  - Statistics are old.
  - The column can contain null.
  - Using the index would actually be slower than not using it.

Optimizer Statistics

- Describe the database and the objects in the database
- Information used by the query optimizer to estimate:
  - Selectivity of predicates
  - Cost of each execution plan
  - Access method, join order, and join method
  - CPU and input/output (I/O) costs
- Refreshing optimizer statistics whenever they are stale is as important as gathering them:
  - Automatically gathered by the system
  - Manually gathered by the user with `DBMS_STATS`

Types of Optimizer Statistics

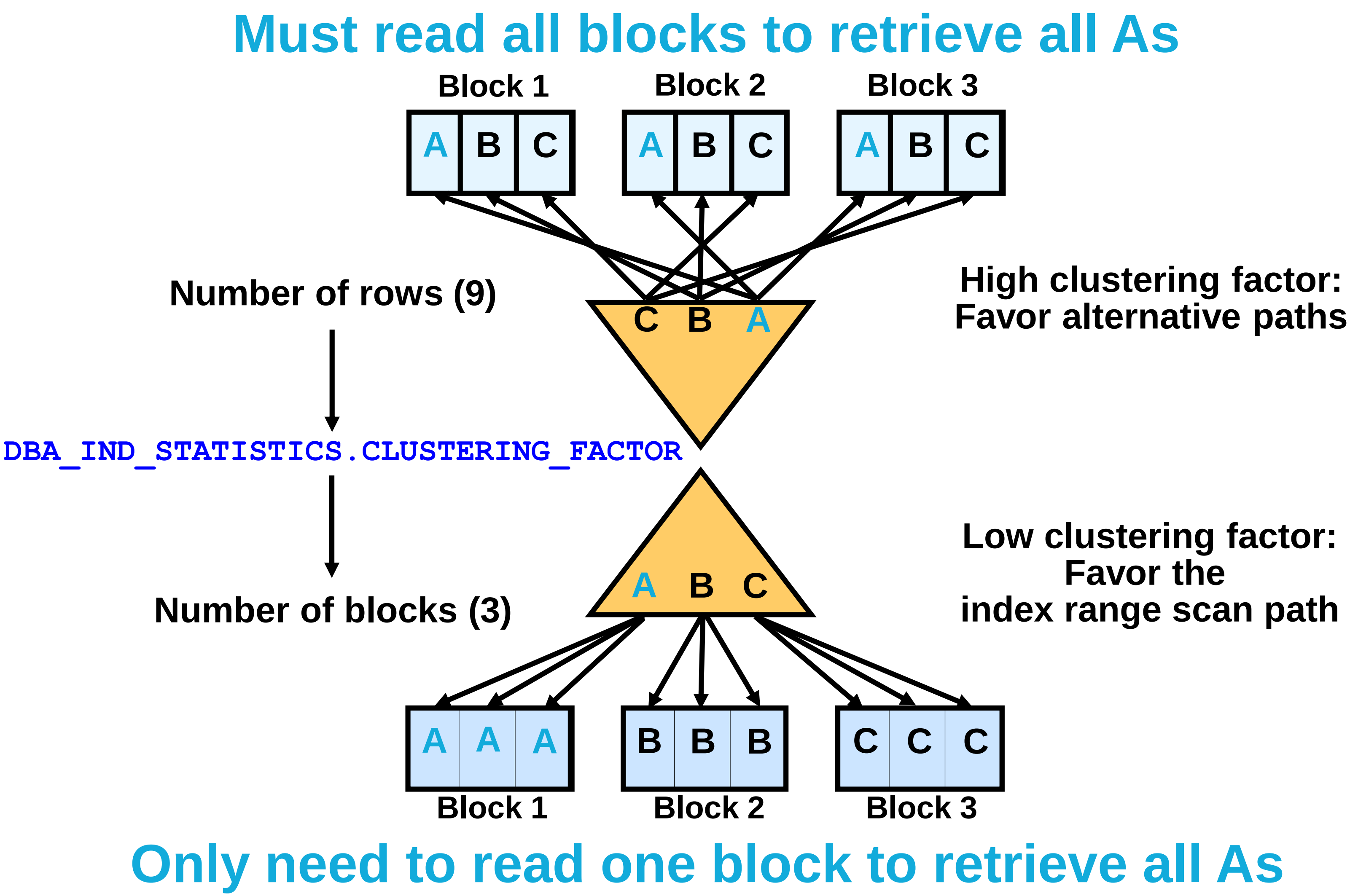
- Table statistics:
  - Number of rows
  - Number of blocks
  - Average row length
- Index Statistics:
  - B*-tree level
  - Distinct keys
  - Number of leaf blocks
  - Clustering factor
- System statistics
  - I/O performance and utilization
  - CPU performance and utilization
- Column statistics
  - Basic: Number of distinct values, number of nulls, average length, min, max
  - Histograms (data distribution when the column data is skewed)
  - Extended statistics

Table Statistics (`DBA_TAB_STATISTICS`)

- Table statistics are used to determine:
  - Table access cost
  - Join cardinality
  - Join order
- Some of the table statistics gathered are:
  - Row count (`NUM_ROWS`)
  - Block count (`BLOCKS`) *Exact*
  - Average row length (`AVG_ROW_LEN`)
  - Statistics status (`STALE_STATS`)

Index Statistics (DBA_IND_STATISTICS)

- Used to decide:
  - Full table scan versus index scan
- Statistics gathered are:
  - B*-tree level (BLEVEL) *Exact*
  - Leaf block count (LEAF_BLOCKS)
  - Clustering factor (CLUSTERING_FACTOR)
  - Distinct keys (DISTINCT_KEYS)
  - Average number of leaf blocks in which each distinct value in the index appears (AVG_LEAF_BLOCKS_PER_KEY)
  - Average number of data blocks in the table pointed to by a distinct value in the index (AVG_DATA_BLOCKS_PER_KEY)
  - Number of rows in the index (NUM_ROWS)

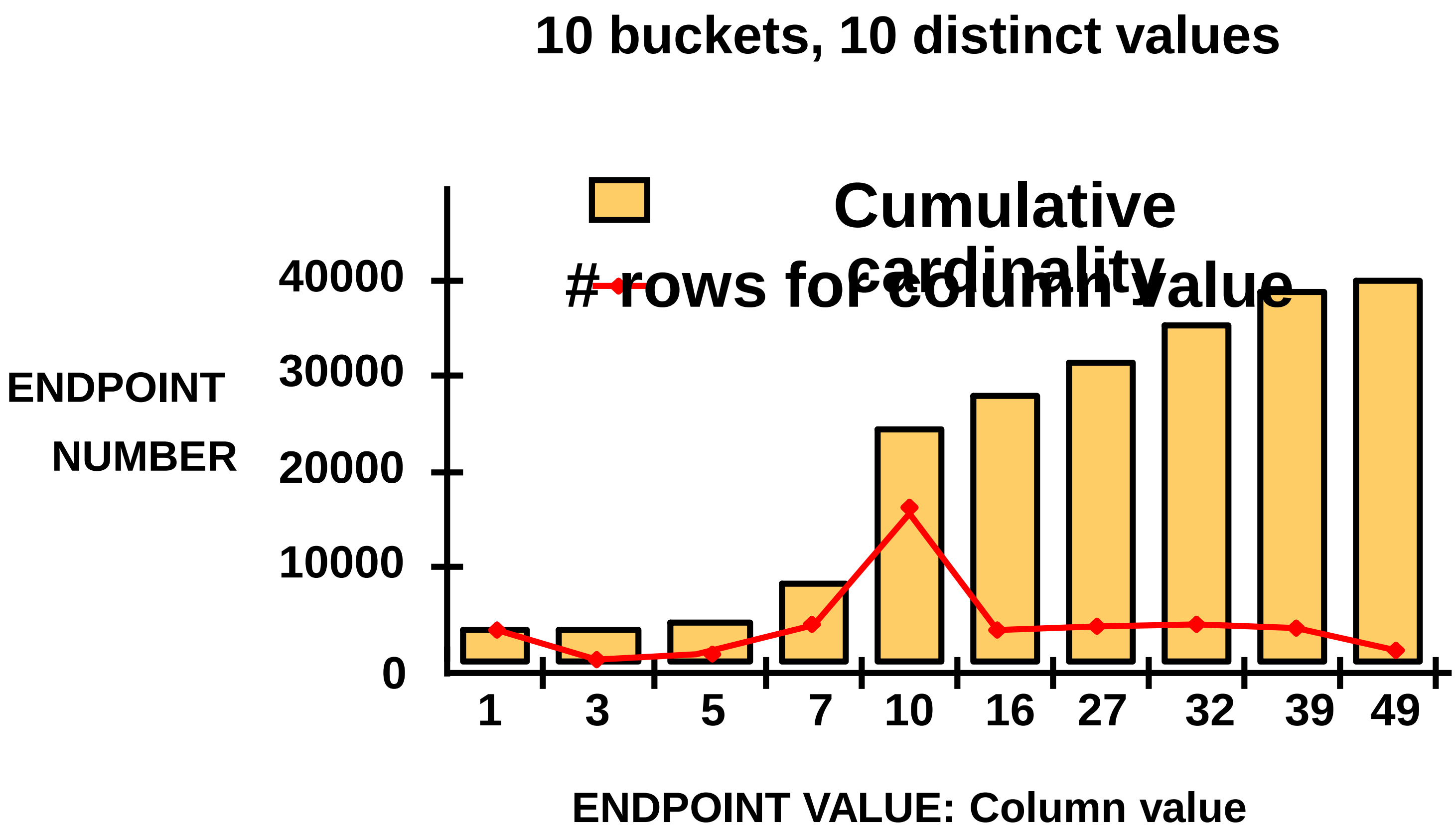

Must read all blocks to retrieve all As
Block 1
Block 2
Block 3
A B C
A B C
A B C
C B A
Number of rows (9)
High clustering factor:
Favor alternative paths
DBA_IND_STATISTICS.CLUSTERING_FACTOR
A B C
Low clustering factor:
Favor the
index range scan path
Number of blocks (3)
A A A
B B B
C C C
Block 1
Block 2
Block 3
Only need to read one block to retrieve all As

Column Statistics (DBA_TAB_COL_STATISTICS)

- Count of distinct values of the column (NUM_DISTINCT)
- Low value (LOW_VALUE) *Exact*
- High value (HIGH_VALUE) *Exact*
- Number of nulls (NUM_NULLS)
- Selectivity estimate for nonpopular values (DENSITY)
- Number of histogram buckets (NUM_BUCKETS)
- Type of histogram (HISTOGRAM)

Histograms

- The optimizer assumes uniform distributions; this may lead to suboptimal access plans in the case of data skew.
- Histograms:
  - Store additional column distribution information
  - Give better selectivity estimates in the case of nonuniform distributions
- With unlimited resources you could store each different value and the number of rows for that value.
- This becomes unmanageable for a large number of distinct values and a different approach is used:
  - Frequency histogram (#distinct values ≤ #buckets)
  - Height-balanced histogram (#buckets < #distinct values)
- They are stored in `DBA_TAB_HISTOGRAMS`.



Distinct values: 1, 3, 5, 7, 10, 16, 27, 32, 39, 49

Number of rows: 40001

Viewing Frequency Histograms

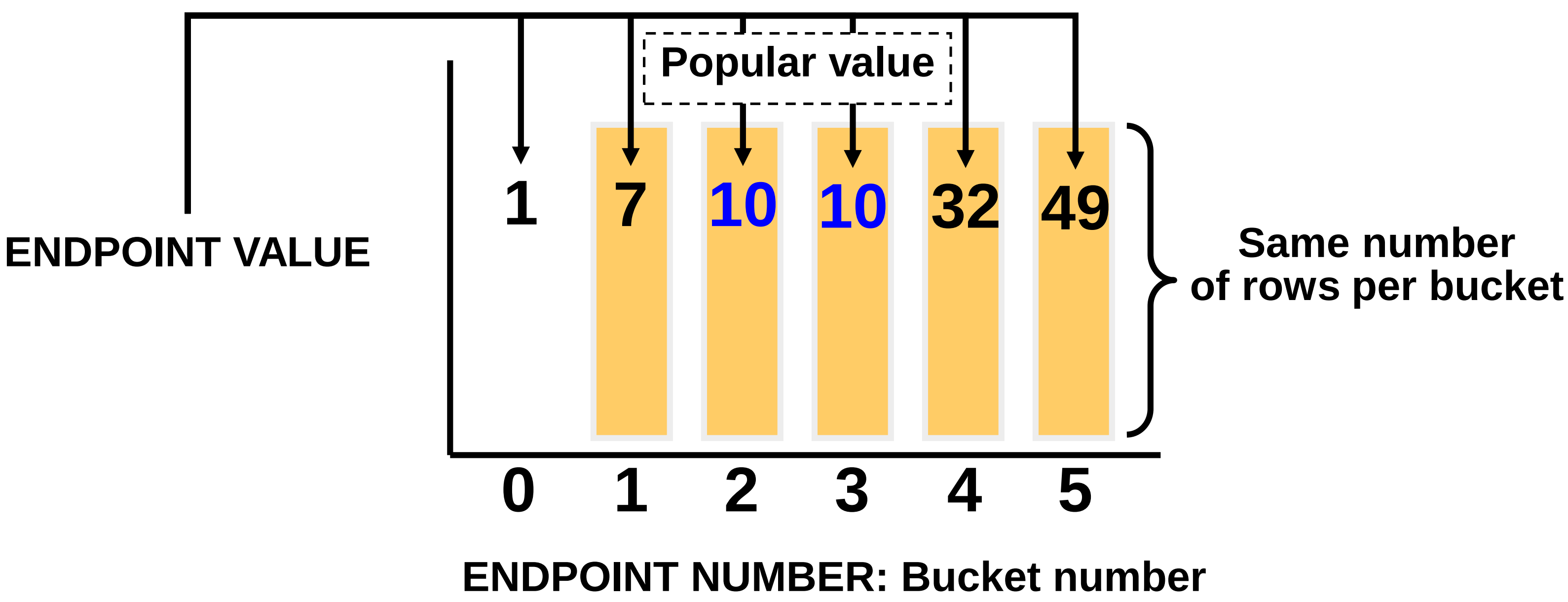
```
BEGIN
DBMS_STATS.gather_table_STATS (OWNNAME=>'OE', TABNAME=>'INVENTORIES',
METHOD_OPT => 'FOR COLUMNS SIZE 20 warehouse_id');
END;
```

```
SELECT column_name, num_distinct, num_buckets, histogram
FROM   USER_TAB_COL_STATISTICS
WHERE  table_name = 'INVENTORIES' AND
       column_name = 'WAREHOUSE_ID';

COLUMN_NAME  NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------ ------------ ----------- ---------
WAREHOUSE_ID            9           9 FREQUENCY
```

```
SELECT endpoint_number, endpoint_value
FROM   USER_HISTOGRAMS
WHERE  table_name = 'INVENTORIES' and column_name = 'WAREHOUSE_ID'
ORDER BY endpoint_number;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
36                           1
213                          2
261                          3
…
```

Height-Balanced Histograms

5 buckets, 10 distinct values
(8000 rows per bucket)



Distinct values: 1, 3, 5, 7, 10, 16, 27, 32, 39, 49

Number of rows: 40001

Viewing Height-Balanced Histograms

```
BEGIN
  DBMS_STATS.gather_table_STATS(OWNNAME =>'OE', TABNAME=>'INVENTORIES',
  METHOD_OPT => 'FOR COLUMNS SIZE 10 quantity_on_hand');
END;
```

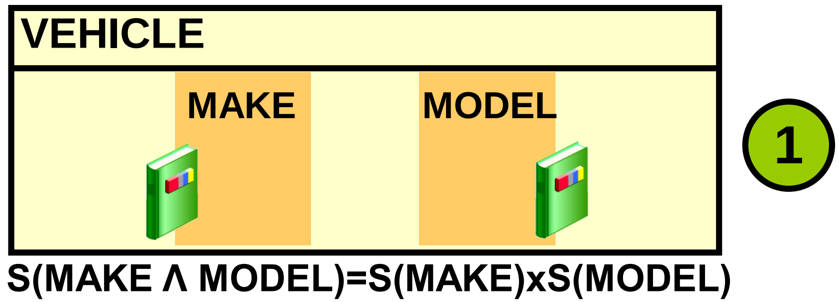
```
SELECT column_name, num_distinct, num_buckets, histogram
FROM USER_TAB_COL_STATISTICS
WHERE table_name = 'INVENTORIES' AND column_name = 'QUANTITY_ON_HAND';

COLUMN_NAME                     NUM_DISTINCT NUM_BUCKETS HISTOGRAM
------------------------------ ------------ ----------- ---------------
QUANTITY_ON_HAND                        237          10 HEIGHT BALANCED
```

```
SELECT endpoint_number, endpoint_value
FROM USER_HISTOGRAMS
WHERE table_name = 'INVENTORIES' and column_name = 'QUANTITY_ON_HAND'
ORDER BY endpoint_number;

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0              0
              1             27
              2             42
              3             57
…
```

Histogram Considerations

- Histograms are useful when you have a high degree of skew in the column distribution.
- Histograms are *not* useful for:
  - Columns which do not appear in the `WHERE` or `JOIN` clauses
  - Columns with uniform distributions
  - Equality predicates with unique columns
- The maximum number of buckets is the least (254,# distinct values).
- Do not use histograms unless they substantially improve performance.

Multicolumn Statistics: Overview

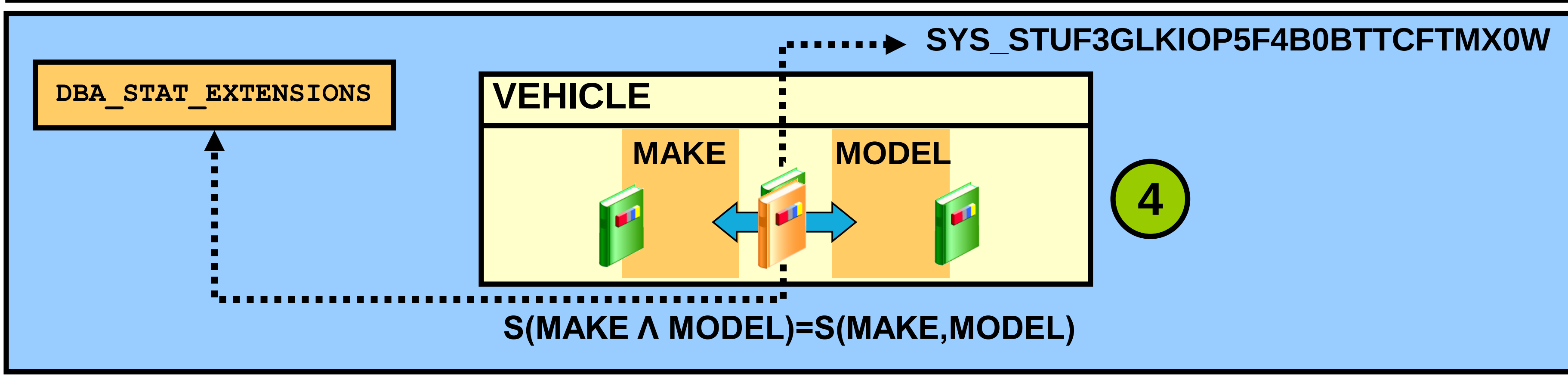



```
select
dbms_stats.create_extended_stats('jfv','vehicle','(make,model)')
from dual;
```

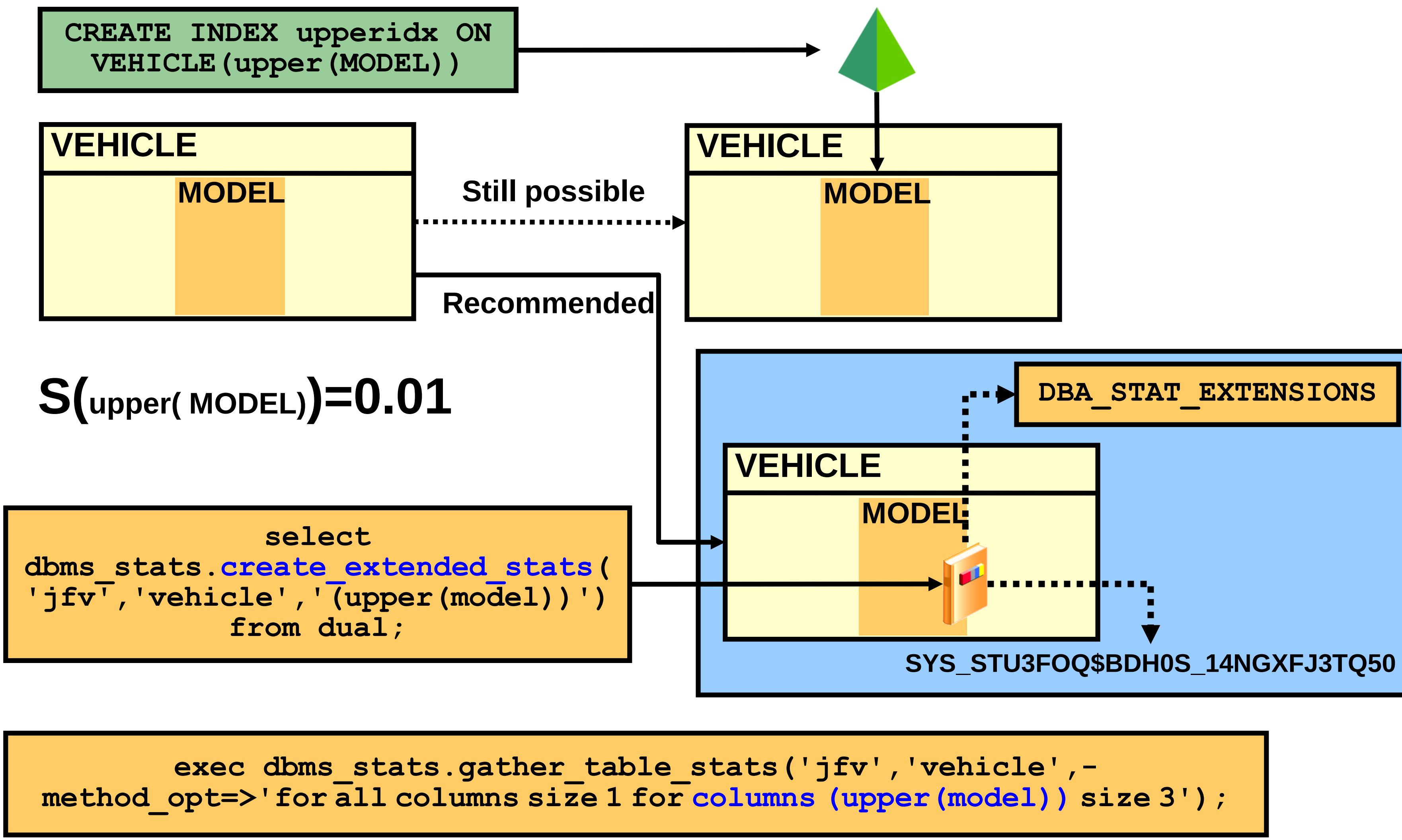
2

```
exec dbms_stats.gather_table_stats('jfv','vehicle',-
method_opt=>'for all columns size 1 for columns (make,model) size 3')
```

3



Expression Statistics: Overview



Gathering System Statistics

- System statistics enable the CBO to use CPU and I/O characteristics.
- System statistics must be gathered on a regular basis; this does not invalidate cached plans.
- Gathering system statistics equals analyzing system activity for a specified period of time:
- Procedures:
  - `DBMS_STATS.GATHER_SYSTEM_STATS`
  - `DBMS_STATS.SET_SYSTEM_STATS`
  - `DBMS_STATS.GET_SYSTEM_STATS`
- `GATHERING_MODE`:
  - `NOWORKLOAD|INTERVAL`
  - `START|STOP`

When to Gather Statistics Manually

- Rely mostly on automatic statistics collection:
  - Change the frequency of automatic statistics collection to meet your needs.
  - Remember that `STATISTICS_LEVEL` should be set to `TYPICAL` or `ALL` for automatic statistics collection to work properly.
- Gather statistics manually for:
  - Objects that are volatile
  - Objects modified in batch operations: Gather statistics as part of the batch operation.
  - External tables, system statistics, fixed objects
  - New objects: Gather statistics right after object creation.

Mechanisms for Gathering Statistics

- Automatic statistics gathering
  - `gather_stats_prog` automated task
- Manual statistics gathering
  - `DBMS_STATS` package
- Dynamic sampling
- When statistics are missing:

Selectivity:	
Equality	1%
Inequality	5%
Other predicates	5%
Table row length	20
# of index leaf blocks	25
# of distinct values	100
Table cardinality	100
Remote table cardinality	2000

Manual Statistics Collection: Factors

- Monitor objects for DMLs.
- Determine the correct sample sizes.
- Determine the degree of parallelism.
- Determine if histograms should be used.
- Determine the cascading effects on indexes.
- Procedures to use in `DBMS_STATS`:
  - `GATHER_INDEX_STATS`
  - `GATHER_TABLE_STATS`
  - `GATHER_SCHEMA_STATS`
  - `GATHER_DICTIONARY_STATS`
  - `GATHER_DATABASE_STATS`
  - `GATHER_SYSTEM_STATS`

Managing Statistics Collection: Example

```
dbms_stats.gather_table_stats
('sh'                     -- schema
,'customers'              -- table
, null                    -- partition
, 20                      -- sample size(%)
, false                   -- block sample?
,'for all columns'        -- column spec
, 4                       -- degree of parallelism
,'default'                -- granularity
, true );                 -- cascade to indexes
```

```
dbms_stats.set_param('CASCADE',
                     'DBMS_STATS.AUTO_CASCADE');
dbms_stats.set_param('ESTIMATE_PERCENT','5');
dbms_stats.set_param('DEGREE','NULL');
```

Locking Statistics

- Prevents automatic gathering
- Is mainly used for volatile tables:
  - Lock without statistics implies dynamic sampling.
  - Lock with statistics for representative values.
- The FORCE

```
BEGIN
DBMS_STATS.DELETE_TABLE_STATS('OE','ORDERS');
DBMS_STATS.LOCK_TABLE_STATS('OE','ORDERS');
END;
```

```
BEGIN
DBMS_STATS.GATHER_TABLE_STATS('OE','ORDERS');
DBMS_STATS.LOCK_TABLE_STATS('OE','ORDERS');
END;
```

```
SELECT stattype_locked FROM
dba_tab_statistics;
```

Restoring Statistics

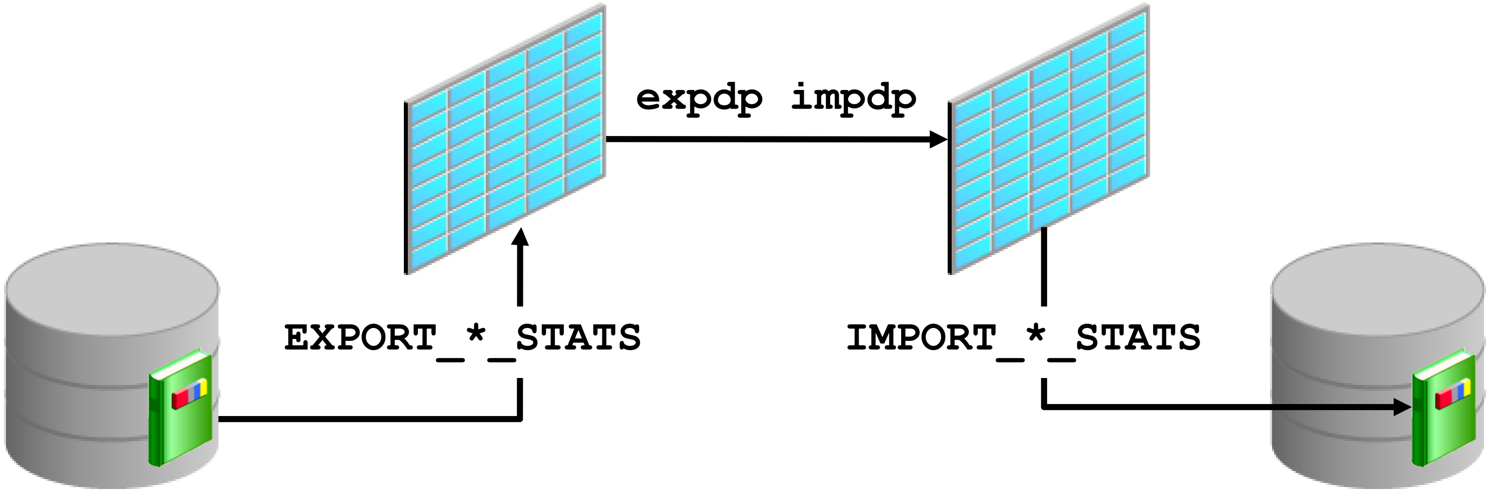
- Past Statistics may be restored with `DBMS_STATS.RESTORE_*_STATS` procedures

- Sta
  - \
  - \
- Sta
  - \
  - /

```
BEGIN
DBMS_STATS.RESTORE_TABLE_STATS(
OWNNAME=>'OE', TABNAME=>'INVENTORIES',
AS_OF_TIMESTAMP=>'15-JUL-10 09.28.01.597526000 AM -05:00');
END;
```

Export and Import Statistics

- Use `DBMS_STATS` procedures:
  - `CREATE_STAT_TABLE` creates the statistics table.
  - `EXPORT_*_STATS` moves the statistics to the statistics table.
  - Use Data Pump to move the statistics table.
  - `IMPORT_*_STATS` moves the statistics to data dictionary.



Optimizer Hints: Overview

- Optimizer hints:
  - Influence optimizer decisions
  - Example:
  - <u>HINTS SHOULD ONLY BE USED AS A LAST RESORT.</u>
  - When you use a hint, it is good practice to also add a comment about that hint.

```
SELECT /*+ INDEX(e empfirstname_idx) skewed col */ *
FROM employees e
WHERE first_name='David'
```

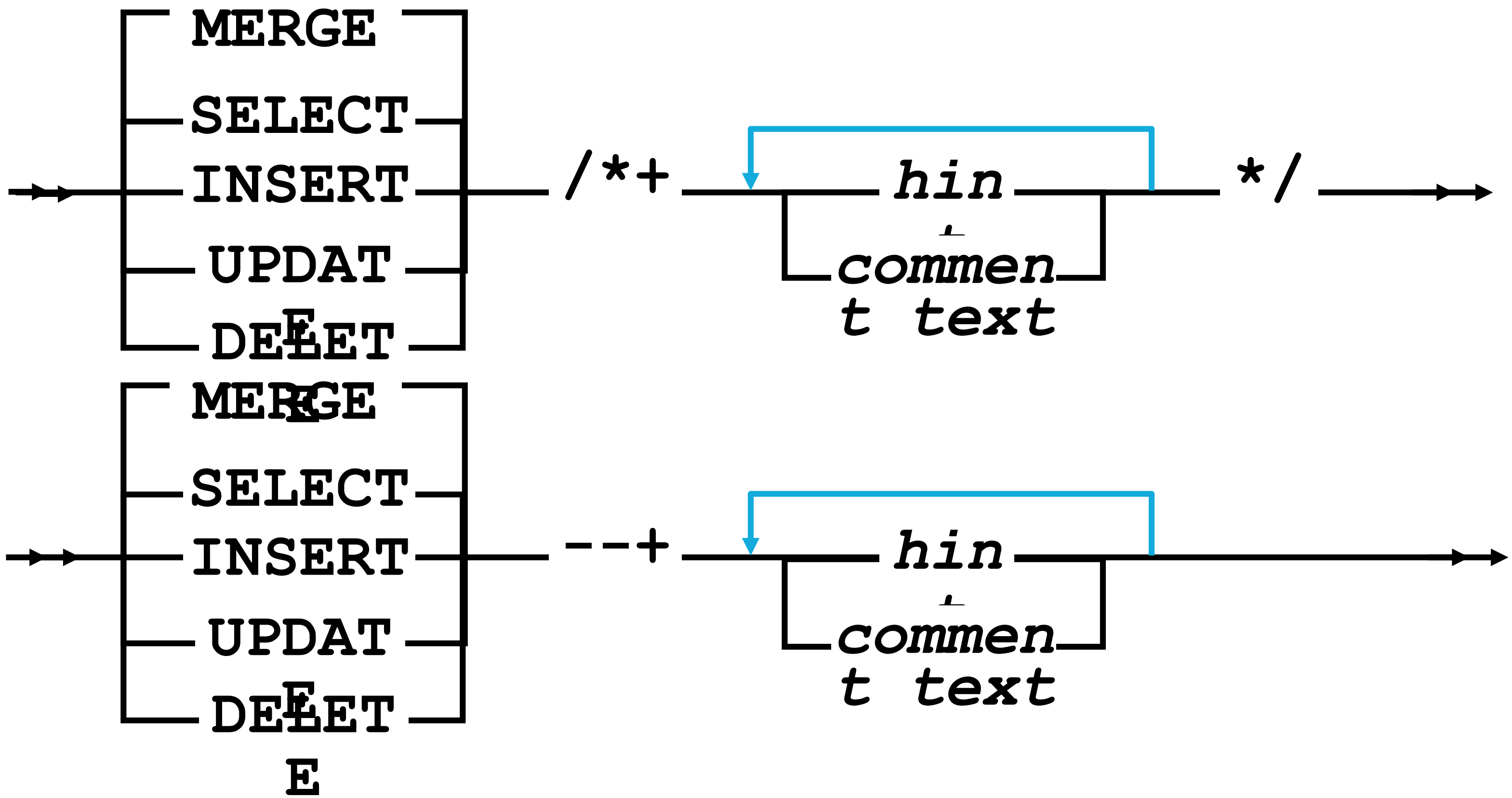
Types of Hints

Single-table hints	Specified on one table or view
Multitable hints	Specify more than one table or view
Query block hints	Operate on a single query block
Statement hints	Apply to the entire SQL statement

Specifying Hints

➢Hints apply to the optimization of only one statement block:

- A self-contained DML statement against a table
- A top-level DML or a subquery



Rules for Hints

- Place hints immediately after the first SQL keyword of a statement block.
- Each statement block can have only one hint comment, but it can contain multiple hints.
- Hints apply to only the statement block in which they appear.
- If a statement uses aliases, hints must reference the aliases rather than the table names.
- The optimizer ignores hints specified incorrectly without raising errors.

Hint Recommendations

- Use hints carefully because they imply a high-maintenance load.
- Be aware of the performance impact of hard-coded hints when they become less valid.

Optimizer Hint Syntax: Example

```
UPDATE /*+ INDEX(p PRODUCTS_PROD_CAT_IX)*/
products p
SET   p.prod_min_price =
(SELECT
(pr.prod_list_price*.95)
FROM products pr
WHERE p.prod_id = pr.prod_id)
WHERE p.prod_category = 'Men'
AND   p.prod_status = 'available, on stock'
/
```

Hint Categories

- There are hints for:
  - Optimization approaches and goals
  - Access paths
  - Query transformations
  - Join orders
  - Join operation
  - Parallel execution
  - Additional hints

Optimization Goals and Approaches

`ALL_ROWS`	Selects a cost-based approach with a goal of best throughput
`FIRST_ROWS(`*`n`*`)`	Instructs the Oracle server to optimize an individual SQL statement for fast response

- **Note:** The `ALTER SESSION... SET OPTIMIZER_MODE` statement does not affect SQL that is run from within PL/SQL.

Hints for Access Paths

FULL	Performs a full table scan
CLUSTER	Accesses table using a cluster scan
HASH	Accesses table using a hash scan
ROWID	Accesses a table by ROWID
INDEX	Selects an index scan for the specified table
INDEX_ASC	Scans an index in ascending order
INDEX_COMBINE	Explicitly chooses a bitmap access path

Hints for Access Paths

INDEX_JOIN	Instructs the optimizer to use an index join as an access path
INDEX_DESC	Scans an index in descending order
INDEX_FFS	Performs a fast-full index scan
INDEX_SS	Performs an index skip scan
NO_INDEX	Does not allow using a set of indexes

The INDEX_COMBINE Hint: Example

```
SELECT /*+INDEX_COMBINE(CUSTOMERS)*/
                cust_last_name
          FROM  SH.CUSTOMERS
      WHERE ( CUST_GENDER= 'F' AND
    CUST_MARITAL_STATUS =  'single')
OR     CUST_YEAR_OF_BIRTH BETWEEN '1917'
             AND '1920';
```

The `INDEX_COMBINE` Hint: Example

```
Execution Plan
---------------------------------------------------
|   0 | SELECT STATEMENT                        |
|   1 |  TABLE ACCESS BY INDEX ROWID   | CUSTOMERS
|   2 |    BITMAP CONVERSION TO ROWIDS |
|   3 |     BITMAP OR                  |
|   4 |      BITMAP MERGE              |
|   5 |       BITMAP INDEX RANGE SCAN  | CUST_YOB_BIX
|   6 |      BITMAP AND                |
|   7 |       BITMAP INDEX SINGLE VALUE| CUST_MARITAL_BIX
|   8 |       BITMAP INDEX SINGLE VALUE| CUST_GENDER_BIX
```

Hints for Query Transformation

NO_QUERY_TRANSFORMATION	Skips all query transformation
USE_CONCAT	Rewrites OR into UNION ALL and disables INLIST processing
NO_EXPAND	Prevents OR expansions
REWRITE	Rewrites query in terms of materialized views
NO_REWRITE	Turns off query rewrite
UNNEST	Merges subquery bodies into surrounding query block
NO_UNNEST	Turns off unnesting

Hints for Query Transformation

`MERGE`	Merges complex views or subqueries with the surrounding query
`NO_MERGE`	Prevents merging of mergeable views
`STAR_TRANSFORMATION`	Makes the optimizer use the best plan in which the transformation can be used
`FACT`	Indicates that the hinted table should be considered as a fact table
`NO_FACT`	Indicates that the hinted table should not be considered as a fact table

Hints for Join Orders

ORDERED	Causes the Oracle server to join tables in the order in which they appear in the `FROM` clause
LEADING	Uses the specified tables as the first table in the join order

Hints for Join Operations

USE_NL	Joins the specified table using a nested loop join
NO_USE_NL	Does not use nested loops to perform the join
USE_NL_WITH_INDEX	Similar to `USE_NL`, but must be able to use an index for the join
USE_MERGE	Joins the specified table using a sort-merge join
NO_USE_MERGE	Does not perform sort-merge operations for the join
USE_HASH	Joins the specified table using a hash join
NO_USE_HASH	Does not use hash join
DRIVING_SITE	Instructs the optimizer to execute the query at a different site than that selected by the database

Additional Hints

APPEND	Enables direct-path `INSERT`
NOAPPEND	Enables regular `INSERT`
CURSOR_SHARING_EXACT	Prevents replacing literals with bind variables
CACHE	Overrides the default caching specification of the table
PUSH_PRED	Pushes join predicate into view
PUSH_SUBQ	Evaluates nonmerged subqueries first
DYNAMIC_SAMPLING	Controls dynamic sampling to improve server performance

MONITOR	Forces real-time query monitoring
NO_MONITOR	Disables real-time query monitoring
RESULT_CACHE	Caches the result of the query or query fragment
NO_RESULT_CACHE	Disables result caching for the query or query fragment
OPT_PARAM	Sets initialization parameter for query duration

Hints and Views

- Do not use hints in views.
- Use view-optimization techniques:
  - Statement transformation
  - Results accessed like a table
- Hints can be used on mergeable views and nonmergeable views.

Global Table Hints

- Extended hint syntax enables specifying for tables that appear in views
- References a table name in the hint with a recursive dot notation

```
CREATE view city_view AS
SELECT *
FROM   customers c
WHERE  cust_city like 'S%';
```

```
SELECT /*+ index(v.c cust_credit_limit_idx) */
  v.cust_last_name, v.cust_credit_limit
FROM   city_view v
WHERE  cust_credit_limit > 5000;
```

Specifying a Query Block in a Hint

```
explain plan for
select /*+ FULL(@strange dept) */ ename
from emp e, (select /*+ QB_NAME(strange) */ *
             from dept where deptno=10) d
where e.deptno = d.deptno and d.loc = 'C';
```

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(NULL, NULL, 'ALL'));

Plan hash value: 615168685

---------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost(%CPU)|
---------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    41 |      7 (15)|
|*  1 |  HASH JOIN         |      |     1 |    41 |      7 (15)|
|*  2 |   TABLE ACCESS FULL| DEPT |     1 |    21 |      3  (0)|
|*  3 |   TABLE ACCESS FULL| EMP  |     3 |    60 |      3  (0)|
---------------------------------------------------------------
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
              1 - SEL$DB579D14
              2 - SEL$DB579D14 / DEPT@STRANGE
              3 - SEL$DB579D14 / E@SEL$1
```

Specifying a Full Set of Hints

```
SELECT /*+ LEADING(e2 e1) USE_NL(e1)
   INDEX(e1 emp_emp_id_pk) USE_MERGE(j) FULL(j) */
     e1.first_name, e1.last_name, j.job_id,
      sum(e2.salary) total_sal
FROM hr.employees e1, hr.employees e2,
hr.job_history j
WHERE e1.employee_id = e2.manager_id
AND e1.employee_id = j.employee_id
AND e1.hire_date = j.start_date
GROUP BY e1.first_name, e1.last_name, j.job_id
ORDER BY total_sal;
```

Materialized Views

Objectives

➢ After completing this lesson, you should be able to do the following:

- Identify the characteristics and benefits of materialized views
- Use materialized views to enable query rewrites
- Verify the properties of materialized views
- Perform refreshes on materialized views

Materialized Views

- A materialized view:
  - Is a precomputed set of results
  - Has its own data segment and offers:
    - Space management options
    - Use of its own indexes
  - Is useful for:
    - Expensive and complex joins
    - Summary and aggregate data

If Materialized Views Are Not Used

```
SELECT c.cust_id, SUM(amount_sold)
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id;
```

```
CREATE TABLE cust_sales_sum AS
SELECT c.cust_id, SUM(amount_sold) AS amount
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id;
```

```
SELECT * FROM cust_sales_sum;
```

Benefits of Using Materialized Views

```
CREATE MATERIALIZED VIEW cust_sales_mv
ENABLE QUERY REWRITE AS
SELECT c.cust_id, SUM(amount_sold) AS amount
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id;
```

```
SELECT c.cust_id, SUM(amount_sold)
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id;
```

```
Execution Plan
----------------------------------------------------------
  0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=6 …)

  1    0   MAT_VIEW REWRITE ACCESS (FULL) OF 'CUST_SALES_MV' (MAT_VIEW
            REWRITE) (Cost=6 …)
```

How Many Materialized Views?

- One materialized view for multiple queries:
  - One materialized view can be used to satisfy multiple queries.
  - Less disk space is needed.
  - Less time is needed for maintenance.
- Query rewrite chooses the materialized view to use.
- One materialized view per query:
  - Is not recommended because it consumes too much disk space
  - Improves one query's performance

Creating Materialized Views: Syntax Options

```
CREATE MATERIALIZED VIEW mview_name
    [TABLESPACE ts_name]
    [PARALLEL (DEGREE n)]
    [BUILD {IMMEDIATE|DEFERRED}]
    [{ REFRESH {FAST|COMPLETE|FORCE}
    [{ON COMMIT|ON DEMAND}]
     | NEVER REFRESH } ]
    [{ENABLE|DISABLE} QUERY REWRITE]

AS SELECT … FROM …
```

Creating Materialized Views: Example

```
CREATE MATERIALIZED VIEW cost_per_year_mv
ENABLE QUERY REWRITE
AS
SELECT    t.week_ending_day
,         t.calendar_year
,         p.prod_subcategory
,         sum(c.unit_cost) AS dollars
FROM      costs c
,         times t
,         products p
WHERE     c.time_id = t.time_id
AND       c.prod_id = p.prod_id
GROUP BY  t.week_ending_day
,         t.calendar_year
,         p.prod_subcategory;

Materialized view created.
```

Types of Materialized Views

- Materialized views with aggregates

```
CREATE MATERIALIZED VIEW cust_sales_mv AS
SELECT c.cust_id, s.channel_id,
       SUM(amount_sold) AS amount
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id, s.channel_id;
```

```
CREATE MATERIALIZED VIEW sales_products_mv AS
SELECT s.time_id, p.prod_name
FROM sales s, products p
WHERE s.prod_id = p.prod_id;
```

Refresh Methods

- You can specify how you want your materialized views to be refreshed from the detail tables by selecting one of four options:
  - `COMPLETE`
  - `FAST`
  - `FORCE`
  - `NEVER`
- You can view the `REFRESH_METHOD` in the `ALL_MVIEWS` data dictionary view.

Refresh Modes

- Manual refresh
  - Specify `ON DEMAND` option
  - By using the `DBMS_MVIEW` package
- Automatic refresh Synchronous
  - Specify `ON COMMIT option`
  - Upon commit of changes to the underlying tables but independent of the committing transaction
- Automatic refresh Asynchronous
  - Specify using `START WITH` and `NEXT` clauses
  - Defines a refresh interval for the materialized view
- `REFRESH_MODE` in `ALL_MVIEWS`

Manual Refresh with `DBMS_MVIEW`

- For `ON DEMAND` refresh only
- Three procedures with the `DBMS_MVIEW` package:
  - `REFRESH`
  - `REFRESH_ALL_MVIEWS`
  - `REFRESH_DEPENDENT`

Materialized Views: Manual Refresh

Specific materialized views:

```
Exec DBMS_MVIEW.REFRESH('cust_sales_mv');
```

Materialized views based on one or more tables:

```
VARIABLE fail NUMBER;
exec DBMS_MVIEW.REFRESH_DEPENDENT(-
:fail,'CUSTOMERS,SALES');
```

All materialized views due for refresh:

```
VARIABLE fail NUMBER;
exec DBMS_MVIEW.REFRESH_ALL_MVIEWS(:fail);
```

Query Rewrites

- If you want to use a materialized view instead of the base tables, a query must be rewritten.
- Query rewrites are transparent to applications.
- Query rewrites do not require special privileges on the materialized view.
- A materialized view can be enabled or disabled for query rewrites.

Query Rewrites

- Use `EXPLAIN PLAN` or `AUTOTRACE` to verify that query rewrites occur.
- Check the query response:
  - Fewer blocks are accessed.
  - Response time should be significantly better.

Enabling and Controlling Query Rewrites

- Query rewrites are available with cost-based optimization only.

```
QUERY_REWRITE_ENABLED = {true|false|force}
QUERY_REWRITE_INTEGRITY =
{enforced|trusted|stale_tolerated}
```

Query Rewrite: Example

```
EXPLAIN PLAN FOR
SELECT   t.week_ending_day
,        t.calendar_year
,        p.prod_subcategory
,        sum(c.unit_cost) AS dollars
FROM     costs c
,        times t
,        products p
WHERE    c.time_id = t.time_id
   . . .
```

```
Execution Plan
--------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost…)
   1    0   MAT_VIEW REWRITE ACCESS (FULL) OF 'costs_per_year_mv' (
             MAT_VIEW REWRITE) (Cost…)
```

Query Rewrite: Example

```
SELECT    t.week_ending_day
,         t.calendar_year
,         p.prod_subcategory
,         sum(c.unit_cost) AS dollars
FROM      costs c, times t, products p
WHERE     c.time_id = t.time_id
AND       c.prod_id = p.prod_id
AND       t.calendar_year = '1999'
GROUP BY  t.week_ending_day, t.calendar_year
,         p.prod_subcategory
HAVING    sum(c.unit_cost) > 10000;
```

```
SELECT    week_ending_day
,         prod_subcategory
,         dollars
FROM      cost_per_year_mv
WHERE     calendar_year = '1999'
AND       dollars > 10000;
```

Verifying Query Rewrite

```
CREATE MATERIALIZED VIEW cust_orders_mv
ENABLE QUERY REWRITE AS
SELECT c.customer_id, SUM(order_total) AS amt
FROM   oe.orders s, oe.customers c
WHERE  s.customer_id = c.customer_id
GROUP BY c.customer_id;
```

```
SELECT /*+ REWRITE_OR_ERROR */ c.customer_id,
SUM(order_total)AS amt
FROM   oe.orders s, oe.customers c
WHERE  s.customer_id = c.customer_id
GROUP BY c.customer_id;
```

```
ORA-30393: a query block in the statement did
not rewrite
```

SQL Access Advisor

➢ For a given workload, the SQL Access Advisor:

- Recommends creating the appropriate:
  - Materialized views
  - Materialized view logs
  - Indexes
- Provides recommendations to optimize for:
  - Fast refresh
  - Query rewrite
- Can be run:
  - From Oracle Enterprise Manager by using the SQL Access Advisor Wizard
  - By invoking the `DBMS_ADVISOR` package

Using the DBMS_MVIEW Package

- DBMS_MVIEW methods
  - EXPLAIN_MVIEW
  - EXPLAIN_REWRITE
  - TUNE_MVIEW

```
DBMS_ADVISOR.TUNE_MVIEW (
        task_name IN OUT VARCHAR2,
        mv_create_stmt IN [CLOB | VARCHAR2]
                            );
```

Results of `Tune_MVIEW`

- `IMPLEMENTATION` recommendations
  - `CREATE MATERIALIZED VIEW LOG` statements
  - `ALTER MATERIALIZED VIEW LOG FORCE` statements
  - One or more `CREATE MATERIALIZED VIEW` statements
- `UNDO` recommendations
  - `DROP MATERIALIZED VIEW` statements

`DBMS_MVIEW.EXPLAIN_MVIEW` Procedure

- Accepts:
  - Materialized view name
  - SQL statement
- Advises what is and what is not possible:
  - For an existing materialized view
  - For a potential materialized view before you create it
- Stores results in `MV_CAPABILITIES_TABLE` (relational table) or in a `VARRAY`
- `utlxmv.sql` must be executed as the current user to create `MV_CAPABILITIES_TABLE`.

Explain Materialized View: Example

```
EXEC dbms_mview.explain_mview (
       'cust_sales_mv', '123');
```

```
SELECT capability_name, possible, related_text,msgtxt
FROM mv_capabilities_table
WHERE statement_id = '123' ORDER BY seq;
```

```
CAPABILITY_NAME    P RELATED_TE MSGTXT
------------------ - ---------- --------------------
...
REFRESH_COMPLETE   Y
REFRESH_FAST       N
REWRITE            N
PCT_TABLE          N SALES      no partition key or
                                PMARKER in select
                                list
PCT_TABLE          N CUSTOMERS  relation is not a
                                partitioned
                                table

...
```

Designing for Query Rewrite

- Query rewrite considerations:
  - Constraints
  - Outer joins
  - Text match
  - Aggregates
  - Grouping conditions
  - Expression matching
  - Date folding
  - Statistics

Materialized View Hints

`REWRITE`	**Rewrites a query in terms of materialized views**
`REWRITE_OR_ERROR`	**Forces an error if a query rewrite is not possible**
`NO_REWRITE`	**Disables query rewrite for the query block**

Summary

- In this lesson, you should have learned how to:
  - Create materialized views
  - Enable query rewrites using materialized views

Objectives

- After completing this lesson, you should be able to do the following:
  - Identify the characteristics and benefits of materialized views
  - Use materialized views to enable query rewrites
  - Verify the properties of materialized views
  - Perform refreshes on materialized views

Materialized Views

- A materialized view:
  - Is a precomputed set of results
  - Has its own data segment and offers:
    - Space management options
    - Use of its own indexes
  - Is useful for:
    - Expensive and complex joins
    - Summary and aggregate data

If Materialized Views Are Not Used

```
SELECT c.cust_id, SUM(amount_sold)
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id;
```

```
CREATE TABLE cust_sales_sum AS
SELECT c.cust_id, SUM(amount_sold) AS amount
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id;
```

```
SELECT * FROM cust_sales_sum;
```

Benefits of Using Materialized Views

```
CREATE MATERIALIZED VIEW cust_sales_mv
ENABLE QUERY REWRITE AS
SELECT c.cust_id, SUM(amount_sold) AS amount
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id;
```

```
SELECT c.cust_id, SUM(amount_sold)
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id;
```

```
Execution Plan
----------------------------------------------------------
  0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=6 …)

  1    0   MAT_VIEW REWRITE ACCESS (FULL) OF 'CUST_SALES_MV' (MAT_VIEW
            REWRITE) (Cost=6 …)
```

How Many Materialized Views?

- One materialized view for multiple queries:
  - One materialized view can be used to satisfy multiple queries.
  - Less disk space is needed.
  - Less time is needed for maintenance.
- Query rewrite chooses the materialized view to use.
- One materialized view per query:
  - Is not recommended because it consumes too much disk space
  - Improves one query's performance

Creating Materialized Views: Syntax Options

```
CREATE MATERIALIZED VIEW mview_name
    [TABLESPACE ts_name]
    [PARALLEL (DEGREE n)]
    [BUILD {IMMEDIATE|DEFERRED}]
    [{ REFRESH {FAST|COMPLETE|FORCE}
    [{ON COMMIT|ON DEMAND}]
     | NEVER REFRESH } ]
    [{ENABLE|DISABLE} QUERY REWRITE]

AS SELECT … FROM …
```

Creating Materialized Views: Example

```
CREATE MATERIALIZED VIEW cost_per_year_mv
ENABLE QUERY REWRITE
AS
SELECT    t.week_ending_day
,         t.calendar_year
,         p.prod_subcategory
,         sum(c.unit_cost) AS dollars
FROM      costs c
,         times t
,         products p
WHERE     c.time_id = t.time_id
AND       c.prod_id = p.prod_id
GROUP BY t.week_ending_day
,         t.calendar_year
,         p.prod_subcategory;

Materialized view created.
```

Types of Materialized Views

- Materialized views with aggregates

```
CREATE MATERIALIZED VIEW cust_sales_mv AS
SELECT c.cust_id, s.channel_id,
       SUM(amount_sold) AS amount
FROM   sales s, customers c
WHERE  s.cust_id = c.cust_id
GROUP BY c.cust_id, s.channel_id;
```

-

```
CREATE MATERIALIZED VIEW sales_products_mv AS
SELECT s.time_id, p.prod_name
FROM sales s, products p
WHERE s.prod_id = p.prod_id;
```

Refresh Methods

- You can specify how you want your materialized views to be refreshed from the detail tables by selecting one of four options:
  - `COMPLETE`
  - `FAST`
  - `FORCE`
  - `NEVER`
- You can view the `REFRESH_METHOD` in the `ALL_MVIEWS` data dictionary view.

Refresh Modes

- Manual refresh
  - Specify `ON DEMAND` option
  - By using the `DBMS_MVIEW` package
- Automatic refresh Synchronous
  - Specify `ON COMMIT option`
  - Upon commit of changes to the underlying tables but independent of the committing transaction
- Automatic refresh Asynchronous
  - Specify using `START WITH` and `NEXT` clauses
  - Defines a refresh interval for the materialized view
- `REFRESH_MODE` in `ALL_MVIEWS`

Manual Refresh with `DBMS_MVIEW`

- For `ON DEMAND` refresh only
- Three procedures with the `DBMS_MVIEW` package:
  - `REFRESH`
  - `REFRESH_ALL_MVIEWS`
  - `REFRESH_DEPENDENT`

Materialized Views: Manual Refresh

Specific materialized views:

```
Exec DBMS_MVIEW.REFRESH('cust_sales_mv');
```

Materialized views based on one or more tables:

```
VARIABLE fail NUMBER;
exec DBMS_MVIEW.REFRESH_DEPENDENT(-
:fail,'CUSTOMERS,SALES');
```

All materialized views due for refresh:

```
VARIABLE fail NUMBER;
exec DBMS_MVIEW.REFRESH_ALL_MVIEWS(:fail);
```

Query Rewrites

- If you want to use a materialized view instead of the base tables, a query must be rewritten.
- Query rewrites are transparent to applications.
- Query rewrites do not require special privileges on the materialized view.
- A materialized view can be enabled or disabled for query rewrites.

Query Rewrites

- Use `EXPLAIN PLAN` or `AUTOTRACE` to verify that query rewrites occur.
- Check the query response:
  - Fewer blocks are accessed.
  - Response time should be significantly better.

Enabling and Controlling Query Rewrites

- Query rewrites are available with cost-based optimization only.
- The following optimizer hints influence query rewrites:

```
QUERY_REWRITE_ENABLED = {true|false|force}
QUERY_REWRITE_INTEGRITY =
{enforced|trusted|stale_tolerated}
```

Query Rewrite: Example

```
EXPLAIN PLAN FOR
SELECT    t.week_ending_day
,         t.calendar_year
,         p.prod_subcategory
,         sum(c.unit_cost) AS dollars
FROM      costs c
,         times t
,         products p
WHERE     c.time_id = t.time_id
   ...
```

```
Execution Plan
----------------------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost…)
   1    0   MAT_VIEW REWRITE ACCESS (FULL) OF 'costs_per_year_mv' (
             MAT_VIEW REWRITE) (Cost…)
```

Query Rewrite: Example

```
SELECT    t.week_ending_day
,         t.calendar_year
,         p.prod_subcategory
,         sum(c.unit_cost) AS dollars
FROM      costs c, times t, products p
WHERE     c.time_id = t.time_id
AND       c.prod_id = p.prod_id
AND       t.calendar_year = '1999'
GROUP BY  t.week_ending_day, t.calendar_year
,         p.prod_subcategory
HAVING    sum(c.unit_cost) > 10000;
```

```
SELECT    week_ending_day
,         prod_subcategory
,         dollars
FROM      cost_per_year_mv
WHERE     calendar_year = '1999'
AND       dollars > 10000;
```

Verifying Query Rewrite

```
CREATE MATERIALIZED VIEW cust_orders_mv
ENABLE QUERY REWRITE AS
SELECT c.customer_id, SUM(order_total) AS amt
FROM   oe.orders s, oe.customers c
WHERE  s.customer_id = c.customer_id
GROUP BY c.customer_id;
```

```
SELECT /*+ REWRITE_OR_ERROR */ c.customer_id,
SUM(order_total)AS amt
FROM   oe.orders s, oe.customers c
WHERE  s.customer_id = c.customer_id
GROUP BY c.customer_id;
```

```
ORA-30393: a query block in the statement did
not rewrite
```

SQL Access Advisor

➢ For a given workload, the SQL Access Advisor:

- Recommends creating the appropriate:
  - Materialized views
  - Materialized view logs
  - Indexes
- Provides recommendations to optimize for:
  - Fast refresh
  - Query rewrite
- Can be run:
  - From Oracle Enterprise Manager by using the SQL Access Advisor Wizard
  - By invoking the `DBMS_ADVISOR` package

Using the DBMS_MVIEW Package

- DBMS_MVIEW methods
  - EXPLAIN_MVIEW
  - EXPLAIN_REWRITE
  - TUNE_MVIEW

Tuning Materialized Views for
Fast Refresh and Query Rewrite

```
DBMS_ADVISOR.TUNE_MVIEW (
    task_name IN OUT VARCHAR2,
    mv_create_stmt IN [CLOB | VARCHAR2]
);
```

Results of `Tune_MVIEW`

- `IMPLEMENTATION` recommendations
  - `CREATE MATERIALIZED VIEW LOG` statements
  - `ALTER MATERIALIZED VIEW LOG FORCE` statements
  - One or more `CREATE MATERIALIZED VIEW` statements
- `UNDO` recommendations
  - `DROP MATERIALIZED VIEW` statements

DBMS_MVIEW.EXPLAIN_MVIEW Procedure

- Accepts:
  - Materialized view name
  - SQL statement
- Advises what is and what is not possible:
  - For an existing materialized view
  - For a potential materialized view before you create it
- Stores results in MV_CAPABILITIES_TABLE (relational table) or in a VARRAY
- utlxmv.sql must be executed as the current user to create MV_CAPABILITIES_TABLE.

```
EXEC dbms_mview.explain_mview (
       'cust_sales_mv', '123');
```

```
SELECT capability_name, possible,
related_text,msgtxt
FROM mv_capabilities_table
WHERE statement_id = '123' ORDER BY
```

```
CAPABILITY_NAME    P RELATED_TE MSGTXT
------------------ - ---------- --------
------------
...
REFRESH_COMPLETE   Y
REFRESH_FAST       N
REWRITE            N
PCT_TABLE          N SALES      no
partition key or
                                PMARKER
in select
                                list
PCT_TABLE          N CUSTOMERS  relation
```

Designing for Query Rewrite

- Query rewrite considerations:
  - Constraints
  - Outer joins
  - Text match
  - Aggregates
  - Grouping conditions
  - Expression matching
  - Date folding
  - Statistics

Materialized View Hints

`REWRITE`	**Rewrites a query in terms of materialized views**
`REWRITE_OR_ERROR`	**Forces an error if a query rewrite is not possible**
`NO_REWRITE`	**Disables query rewrite for the query block**

SUMMARY

- In this lesson, you should have learned how to:
  - Create materialized views
  - Enable query rewrites using materialized views

SUMMARY

- Introduction to SQL tuning
- Describe why the SQL statements are performing poorly
- Introduction to Oracle Optimizer
- Discuss the need for Optimizer
- Explain the various phases of Optimization
- Gather Execution Plans
- Interpret Execution Plans
- Interpret the output of TKPROF
- Gather Optimizer statistics
- Use Hints appropriately