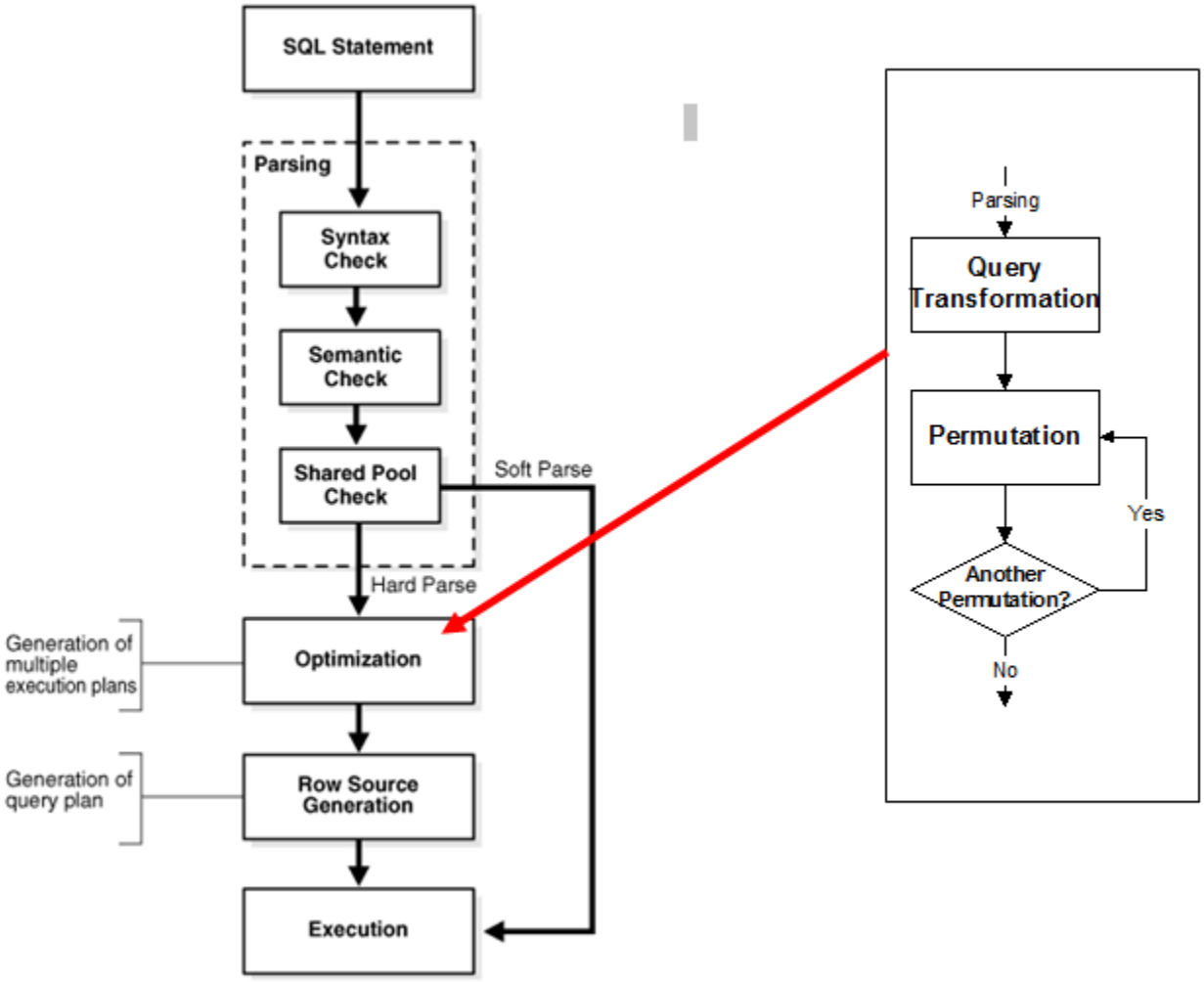
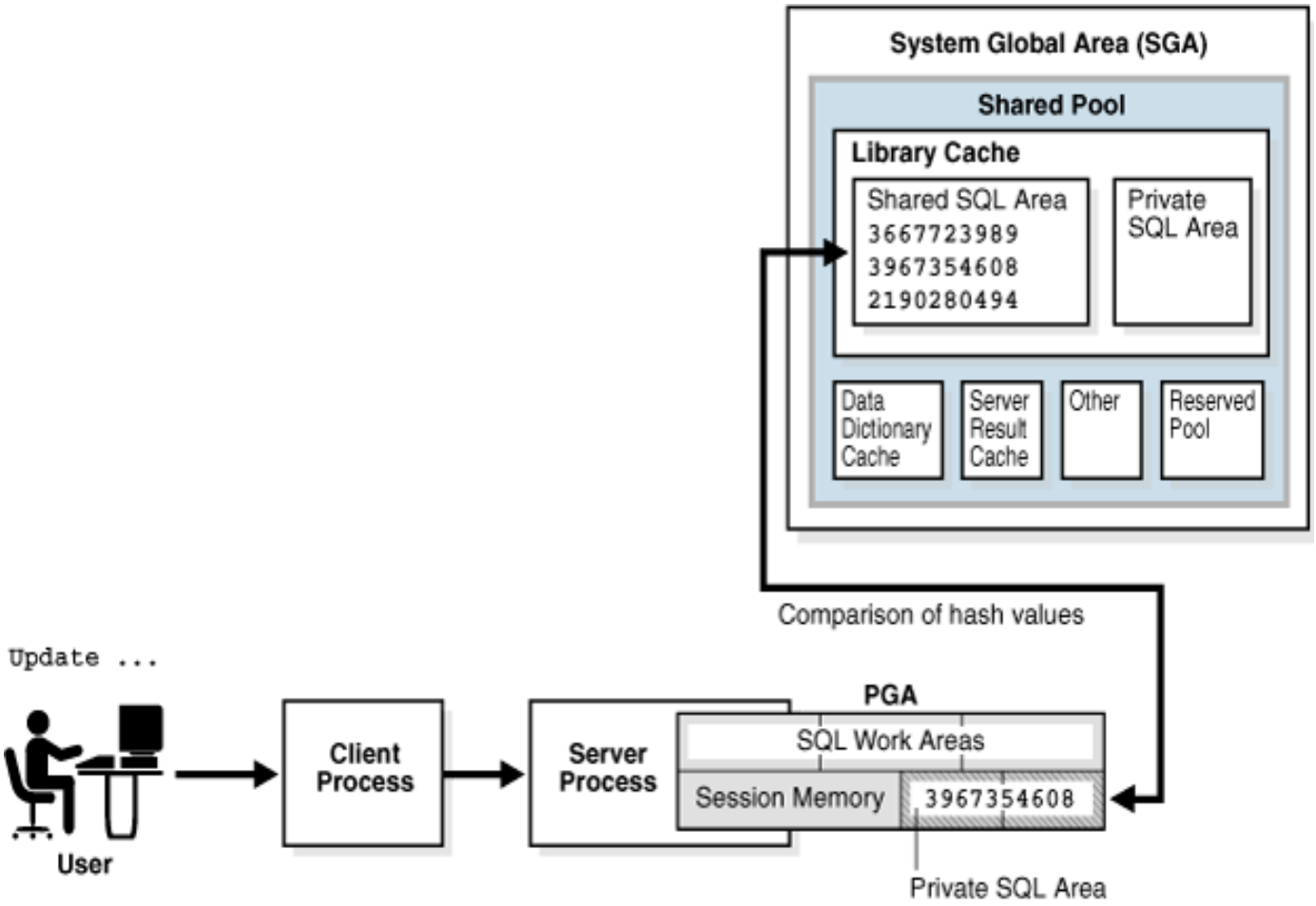
The life of an SQL statement



➢ Parsing

During the parse call, the database performs the following checks:

a. Syntax Check
b. Semantic Check : Whether the objects and columns in the statement exist
c. Shared Pool Check : Checks for Hash Value (Hard Parse or Soft Parse)



What happens the first time a SQL statement is executed?

The RDBMS will create a hash value for text of the statement and then uses that hash value to check for parsed SQL statements already existing in the shared pool with the same hash value. Since it is the first time it is being executed, there is no hash value match and a hard parse occurs.

Costly phase of query optimization is performed now. It then stores the parse tree and execution plan in a shared SQL area.

When it is first parsed a parent and a single child cursor are created.

Sharing of SQL code – next time same SQL is executed

The text of the SQL statement being executed is hashed.

If no matching hash value found, perform a hard parse.

If matching hash value exists in shared pool, compare the text of matched statement in shared pool with the statement which has been hashed. They need to be identical character for character, white spaces, case, comments etc.

Is objects referenced in the SQL statement match the objects referenced by the statement in the shared pool which is being attempted to be shared.

Bind variables being used must match is terms of name, data type and length

Both the session's environment needs to be identical. For example if at the database level we have OPTIMIZER_MODE=ALL_ROWS, but user A has used an ALTER SESSION command to change optimization goal for his session to say FIRST_ROWS, then that session will not be able to use existing shared SQL areas which have a different optimization goal for them.

Case study 1: Demonstrating different sql_id, hash-value but same execution plan

```
SQL> conn naveed/naveed
SQL> set line 1000
SQL> select object_name from user_objects;

OBJECT_NAME
----------------------------------------------------------------
D1
BIGTAB

SQL> select * from D1;

NAME                           DATES
------------------------------ ---------
naveed                         09-JUN-14
mcfadyen                       08-JUN-14
gfs                            10-JUN-14

SQL> grant select on D1 to gfs;

session1:

[oracle@prod dbs]$ sqlplus naveed/naveed
SQL> select * from D1;

NAME                           DATES
------------------------------ ---------
naveed                         09-JUN-14
mcfadyen                       08-JUN-14
gfs                            10-JUN-14



Session2:

[oracle@prod prod]$ sqlplus gfs/gfs
```

```
SQL> select * from naveed.D1;

NAME                           DATES
------------------------------ ---------
naveed                         09-JUN-14
mcfadyen                       08-JUN-14
gfs                            10-JUN-14
```

Session3:

```
[oracle@prod prod]$ sqlplus / as sysdba
SQL> set line 1000
SQL> col SQL_TEXT format a25;

SQL> select sql_id, sql_text, hash_value,
plan_hash_value,LOADS,FIRST_LOAD_TIME,INVALIDATIONS,PARSE_CALLS,CHIL
D_NUMBER from v$sql
where sql_text like 'select * from D1'
and length(sql_text) < 50;
```

SQL_ID	SQL_TEXT	HASH_VALUE	PLAN_HASH_VALUE	LOADS	FIRST_LOAD_TIME	INVALIDATIONS	PARSE_CALLS	CHILD_NUMBER
ahszbvprvnkaa	select * from D1	1874479434	396716558	1	2016-10-22/17:33:34	0	1	0

```
SQL> select sql_id, sql_text, hash_value,
plan_hash_value,LOADS,FIRST_LOAD_TIME,INVALIDATIONS,PARSE_CALLS,CHIL
D_NUMBER from v$sql
where sql_text like 'select * from naveed.D1'
and length(sql_text) < 50;
```

SQL_ID	SQL_TEXT	HASH_VALUE	PLAN_HASH_VALUE	LOADS	FIRST_LOAD_TIME	INVALIDATIONS	PARSE_CALLS	CHILD_NUMBER
4cwh1hj610kmj	select * from naveed.D1	1276136049	396716558	1	2016-10-22/17:39:30	0	1	0

** INVALIDATIONS === Total number of times objects in this namespace were marked invalid because a dependent object was modified like stats collection, DDL,grants etc. INVALIDATIONS should be near zero

Case study 2: Demonstrating same sql_id, hash-value and same execution plan

```
[oracle@prod prod]$ sqlplus gfs/gfs
SQL> create table D1 as select * from naveed.D1;
Table created.

[oracle@prod prod]$ sqlplus / as sysdba
SQL> alter system flush shared_pool;
SQL> alter system flush shared_pool;
SQL> alter system flush shared_pool;
```

session1:

```
[oracle@prod dbs]$ sqlplus naveed/naveed

SQL> select * from D1;

NAME                           DATES
------------------------------ ---------
naveed                         09-JUN-14
mcfadyen                       08-JUN-14
gfs                            10-JUN-14
```

session2:

```
[oracle@prod prod]$ sqlplus gfs/gfs

SQL> select * from D1;

NAME                           DATES
------------------------------ ---------
naveed                         09-JUN-14
mcfadyen                       08-JUN-14
gfs                            10-JUN-14


[oracle@prod prod]$ sqlplus / as sysdba

SQL> set line 1000
SQL> col SQL_TEXT format a25;
SQL> select sql_id, sql_text, hash_value,
plan_hash_value,LOADS,FIRST_LOAD_TIME,EXECUTIONS,INVALIDATIONS,PARSE
_CALLS,CHILD_NUMBER from v$sql
where sql_text like 'select * from D1'
and length(sql_text) < 50;
```

SQL_ID	SQL_TEXT	HASH_VALUE	PLAN_HASH_VALUE	LOADS	FIRST_LOAD_TIME	INVALIDATIONS	PARSE_CALLS	CHILD_NUMBER
ahszbvprvnkaa	select * from D1	1874479434	396716558	1	2016-10-22/17:48:02	0	1	0
ahszbvprvnkaa	select * from D1	1874479434	396716558	1	2016-10-22/17:48:02	0	1	1

```
SQL> select version_count, invalidations
     from v$sqlarea where sql_id='ahszbvprvnkaa';
```

```
VERSION_COUNT INVALIDATIONS
------------- -------------
            2             0
```

Note SQL statements will get **invalidated** and be flushed from the shared pool and will also age out in case Oracle needs to load new statements in the shared pool. Once the shared pool gets full, Oracle will use a LRU algorithm to age out the 'older' SQL statements.

How do we tell if a **parse** has been a hard parse or a soft parse? The V$SQL or V$SQLAREA views will not provide us this information. We will have to turn on tracing and use TKPROF to identify this.

Why was the cursor not shared? Will cover in later chapters

➢ Optimizer

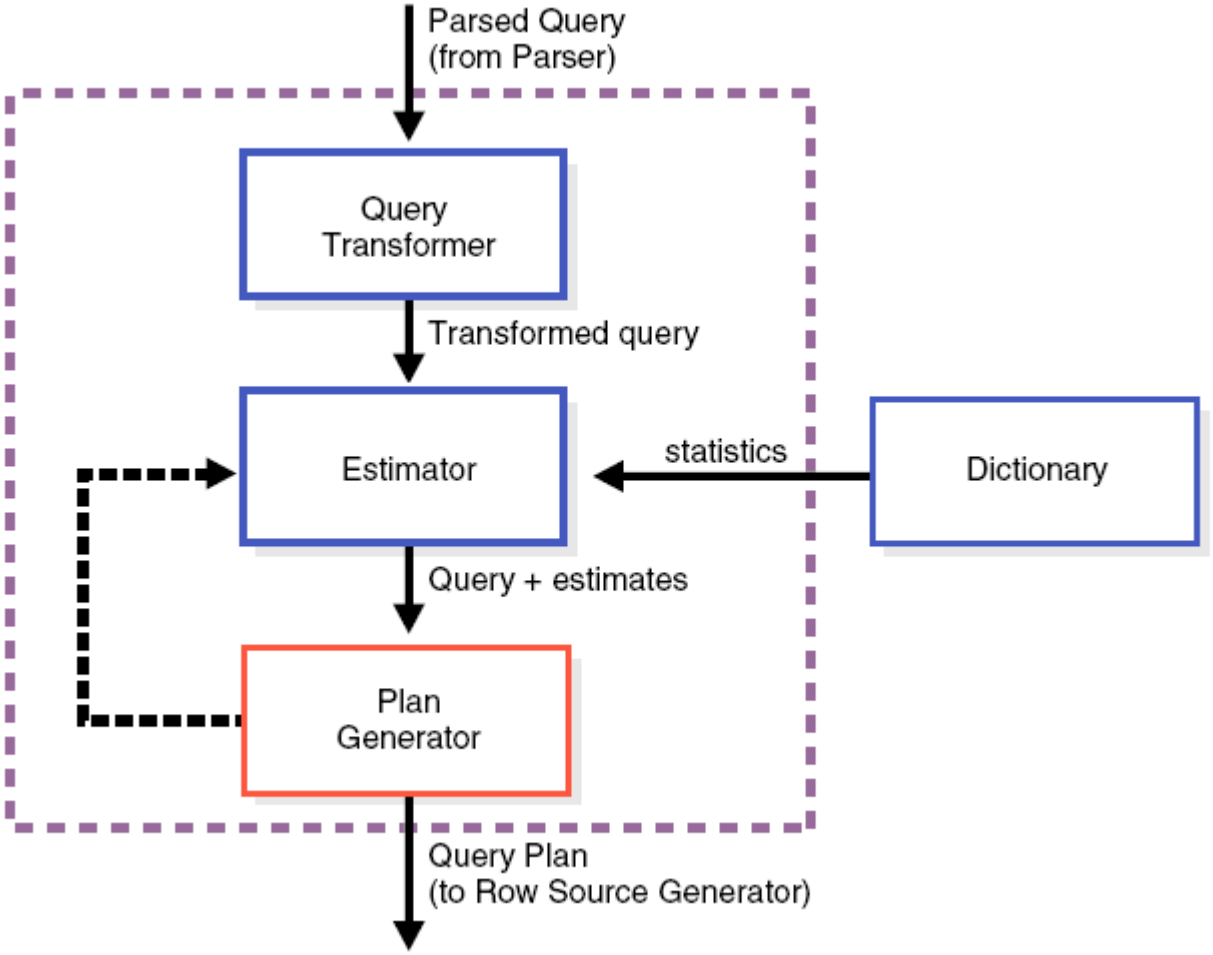
The CBO determines which execution plan is most efficient by considering available access paths and by factoring in information based on statistics for the schema objects (tables or indexes) accessed by the SQL statement. The CBO also considers hints, which are optimization suggestions placed in a comment in the statement.

Optimizer Operations

Operation	Description
Evaluation of expressions and conditions	The optimizer first evaluates expressions and conditions containing constants as fully as possible.
Statement transformation	For complex statements involving, for example, correlated subqueries or views, the optimizer might transform the original statement into an equivalent join statement.
Choice of optimizer goals	The optimizer determines the goal of optimization.
Choice of access paths	For each table accessed by the statement, the optimizer chooses one or more of the available access paths to obtain table data. See "Overview of Optimizer Access Paths".
Choice of join orders	For a join statement that joins more than two tables, the optimizer chooses which pair of tables is joined first, and then which table is joined to the result, and so on.

Components of the Query Optimizer

The three main query optimizer operations are Query transformation, Estimation and Plan generation. The following figure illustrates the components of optimizer.



Each query portion is considered as a query block. The input to the query transformer is a parsed query that is defined by a set of query blocks.

The transformer decides whether it is beneficial to rewrite the existing statements into semantically equivalent statements that could be processed effectively.

The estimator evaluates the total cost of an execution plan. Estimator normally uses the available statistics in computing the measures. These statistics can enhance the degree of accuracy.

By trying out different access paths and join methods, the plan generator investigates a variety of plans for a query block. The generator actually selects the plan with the lowest cost.

The query transformer employs many transformation methods including View merging and Predicate pushing.

The view merging optimization is applicable to views that contain selections, joins and projections. In order to authorize the optimizer to use view merging, you should grant the MERGE ANY VIEW privilege to the user. You need to grant the MERGE VIEW privilege to the user on specific views to allow the optimizer to use view merging. In case of predicate pushing, the optimizer pushes relevant predicates into the view query block.

➢ Row Source Generator

The **row source generator** is software that receives the optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database. It outputs the execution plan for the SQL statement.

The execution plan is a collection of row sources structured in the form of a tree. A row source is an iterative control structure. It processes a set of rows, one row at a time, in an iterated manner. A row source produces a row set.

```
SQL>  SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR('4cwh1hj610kmj'));

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------
SQL_ID  4cwh1hj610kmj, child number 0
-------------------------------------
select * from naveed.D1

Plan hash value: 396716558

-----------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |       |       |     3 (100)|          |
|   1 |  TABLE ACCESS FULL| D1   |     3 |    42 |     3   (0)| 00:00:01 |
```

➢ SQL Execution

During execution, the SQL engine executes each row source in the tree produced by the row source generator.