**Stages of SQL processing:**



**Syntax Check**

Oracle Database must check each SQL statement for syntactic validity.

#### Semantic Check

The semantics of a statement are its meaning. A semantic check determines whether a statement is meaningful, for example, whether the objects and columns in the statement exist.

#### Shared Pool Check

During the parse, the database performs a shared pool check to determine whether it can skip resource-intensive steps of statement processing. To this end, the database uses a hashing algorithm to generate a hash value for every SQL statement. When a user submits a SQL statement, the database searches the [**shared SQL area**](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-577D0DD3-FB98-4B1B-BCFB-95313BB35C01) to see if an existing parsed statement has the same hash value.

**Hard parse**

If Oracle Database cannot reuse existing code, then it must build a new executable version of the application code. This operation is known as a [**hard parse**](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-AB764C9E-2F03-49A9-BF8B-36A9FBD03BCE), or a [**library cache miss**](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-D12396D3-DCDA-4A61-8891-64AD81876EFD).

**Soft parse**

A [**soft parse**](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-BA91B10B-FBA3-4DF6-B59B-9AA57C683D33) is any parse that is not a hard parse. If the submitted statement is the same as a reusable SQL statement in the shared pool, then Oracle Database reuses the existing code. This reuse of code is also called a [**library cache hit**](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-22628BDA-52D1-4349-827A-071163A13D1A). If a check determines that a statement in the shared pool has the same hash value, then the database performs semantic and environment checks to determine whether the statements have the same meaning. Identical syntax is not enough. Even if two statements are semantically identical, an environmental difference can force a hard parse. In this context, the [**optimizer environment**](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-689C2576-297D-471D-8575-86FAE0185AAD) is the totality of session settings that can affect execution plan generation, such as the work area size or optimizer settings (for example, the optimizer mode).

### **SQL Optimization**

During the optimization stage, Oracle Database must perform a hard parse at least once for every unique DML statement and performs the optimization during this parse.

### **SQL Row Source Generation**

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.

The iterative plan is a binary program that, when executed by the SQL engine, produces the result set. The plan takes the form of a combination of steps. Each step returns a [**row set**](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-88998F10-D77C-43B5-B148-E649BF366DC2). The next step either uses the rows in this set, or the last step returns the rows to the application issuing the SQL statement.

### **SQL Execution**

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

**Optimizer components:**



### **Query Transformer**

For some statements, the query transformer determines whether it is advantageous to rewrite the original SQL statement into a semantically equivalent SQL statement with a lower cost.



**LNNVL** function (“logical not null value”)

Tink of it as an **opposite detector**: see details in documentation)

It takes as an argument a condition and returns TRUE if the condition is FALSE or UNKNOWN and FALSE if the condition is TRUE.

**Query Transformations**

The optimizer employs several query transformation techniques.

In **OR expansion**, the optimizer transforms a query with a WHERE clause containing OR operators into a query that uses the UNION ALL operator.

In [**view merging**](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-F4A61987-3A7A-4214-A993-2CBADF65FF31), the optimizer merges the [query block](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-F2D03097-5F4B-4A2A-9DAE-6EC76F7BED61)representing a view into the query block that contains it. View merging can improve plans by enabling the optimizer to consider additional join orders, access methods, and other transformations.

In [**simple view merging**](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-04FFFBB6-FEA7-465D-BAEA-A5BF6F403180), the optimizer merges select-project-join views.

In [**complex view merging**](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-301D0C59-1A05-4954-B0E1-B03B1AD9388F), the optimizer merges views containing GROUP BY and DISTINCT views. Like simple view merging, complex merging enables the optimizer to consider additional join orders and access paths.

In [**subquery unnesting**](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-25ECA776-F687-4C3F-81F5-C9FEF5116A46), the optimizer transforms a nested query into an equivalent join statement, and then optimizes the join.

In the cost-based transformation known as [**join factorization**](https://docs.oracle.com/database/121/TGSQL/glossary.htm#GUID-C35508EC-6259-4DB4-9576-844DFCC3F3D1), the optimizer can factorize common computations from branches of a UNION ALL query.

### **Estimator**

The **estimator** is the component of the optimizer that determines the overall cost of a given execution plan.



**Plan generator**

The **plan generator** explores various plans for a query block by trying out different access paths, join methods, and join orders. Many plans are possible because of the various combinations that the database can use to produce the same result. The **optimizer picks** the plan with **the lowest cost**.



In Oracle Database, **adaptive query optimization** enables the optimizer to make run-time adjustments to execution plans and discover additional information that can lead to better statistics.

**Example:**

SELECT dname, job, AVG(sal)

FROM nikovits.emp NATURAL JOIN nikovits.dept

WHERE hiredate > to\_date('1981.01.01', 'yyyy.mm.dd')

GROUP BY dname, job HAVING SUM(sal) > 5000

ORDER BY AVG(sal) DESC;



Steps and result sets of the previous execution plan:

TMP1 := ΠJOB,SAL,DEPTNO (σHiredate > ….EMP) EMP FULL

|  |  |  |
| --- | --- | --- |
| **JOB** | **SAL** | **DEPTNO** |
| SALESMAN | 1600 | 30 |
| SALESMAN | 1250 | 30 |
| MANAGER | 2975 | 20 |
| MANAGER | 2450 | 10 |
| … | … | … |

TMP2 := ΠDEPTNO,DNAME DEPT DEPT FULL

|  |  |
| --- | --- |
| **DEPTNO** | **DNAME** |
| 10 | ACCOUNTING |
| 20 | RESEARCH |
| … | … |

TMP3 := ΠDNAME,JOB,SAL TMP1 ⋈ TMP2 HASH JOIN

|  |  |  |
| --- | --- | --- |
| **DNAME** | **JOB** | **SAL** |
| SALES | SALESMAN | 1600 |
| SALES | SALESMAN | 1250 |
| RESEARCH | MANAGER | 2975 |
| ACCOUNTING | MANAGER | 2450 |
| RESEARCH | ANALYST | 3000 |
| … | … | … |

TMP4 := γDNAME,JOB,COUNT(SAL), SUM(SAL)TMP3 GROUP BY

|  |  |  |  |
| --- | --- | --- | --- |
| **DNAME** | **JOB** | **COUNT(SAL)** | **SUM(SAL)** |
| SALES | SALESMAN | 4 | 5600 |
| RESEARCH | ANALYST | 2 | 6000 |
| MARKETING | SALESMAN | 1 | 1600 |
| ACCOUNTING | MANAGER | 1 | 2450 |
| ACCOUNTING | PRESIDENT | 1 | 5000 |
| … | … | … | … |

TMP5 := ΠDNAME,JOB,SumSal/Cnt -> AvgSal (σSumSal > 5000 TMP4) FILTER

|  |  |  |
| --- | --- | --- |
| **DNAME** | **JOB** | **AVG(SAL)** |
| SALES | SALESMAN | 1400 |
| RESEARCH | ANALYST | 3000 |

RESULT := τAvgSal TMP5 SORT

|  |  |  |
| --- | --- | --- |
| **DNAME** | **JOB** | **AVG(SAL)** |
| RESEARCH | ANALYST | 3000 |
| SALES | SALESMAN | 1400 |