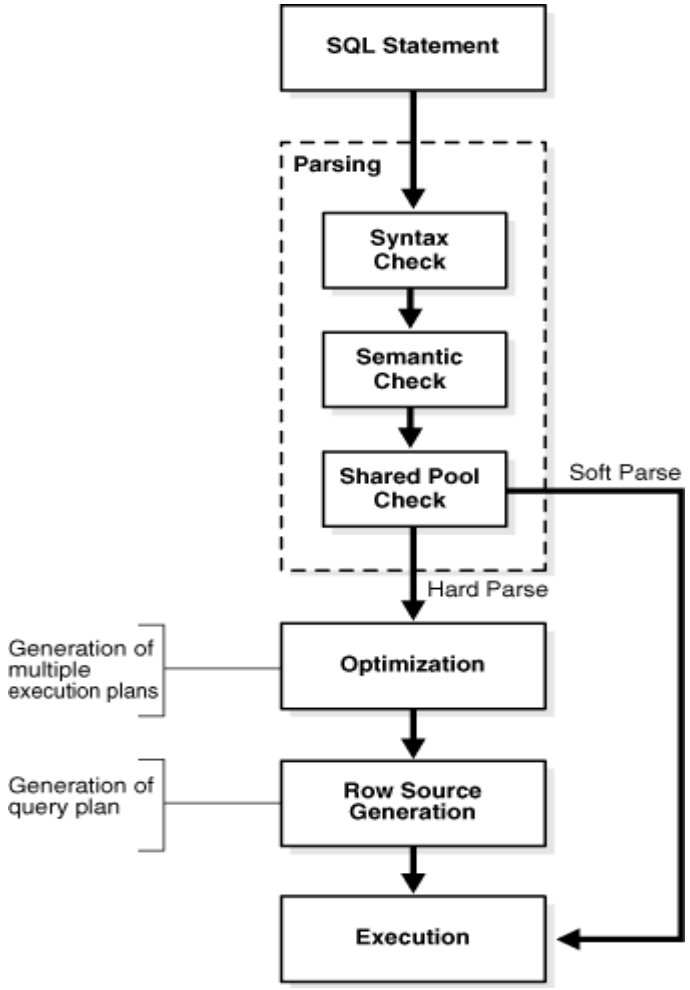
Query Optimization

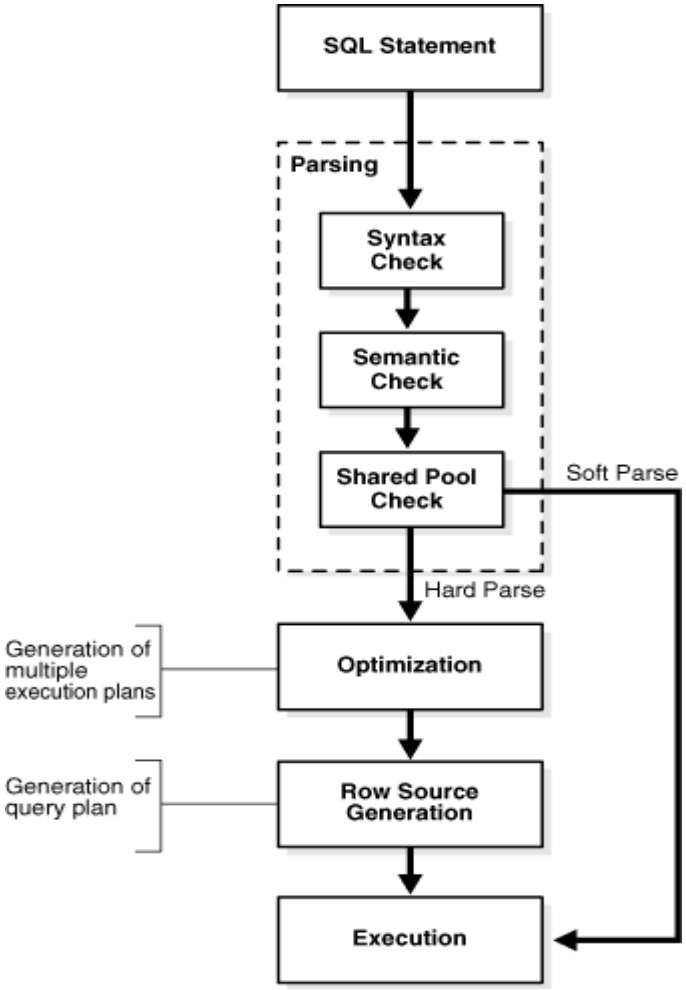
Stages of SQL Processing



SQL processing is the parsing, optimization, row source generation, and execution of a SQL statement.

Depending on the statement, the database may omit some of these stages.

SQL Parsing



The parsing stage involves separating the pieces of a SQL statement into a data structure that other routines can process.

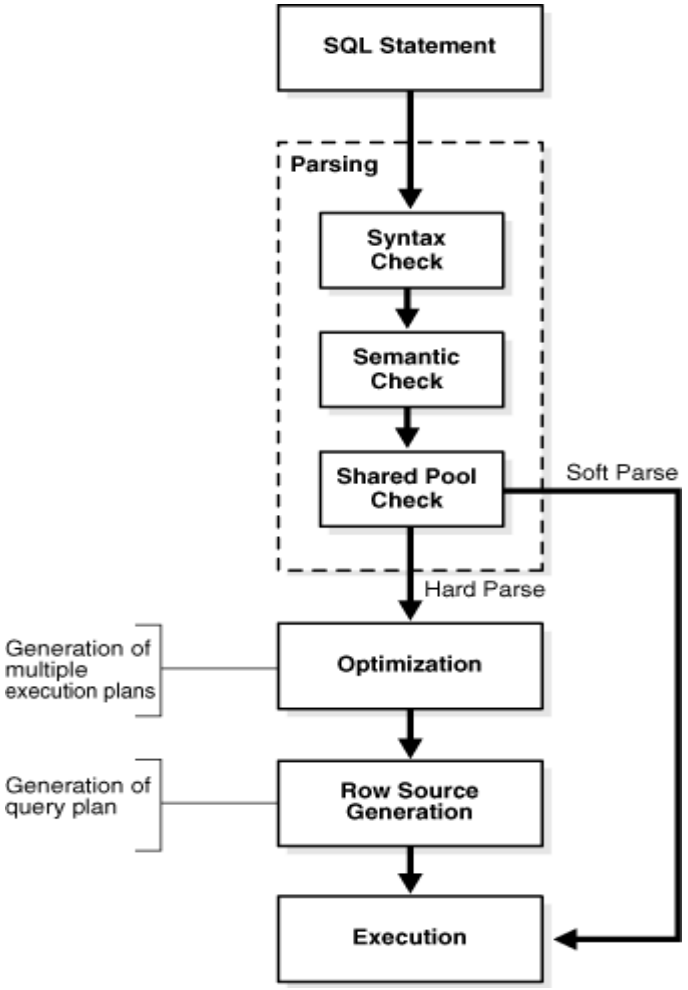
The database parses a statement when instructed by the application, which means that only the application, and not the database itself, can reduce the number of parses.

When an application issues a SQL statement, the application makes a **parse call** to the database to prepare the statement for execution. The parse call opens or creates a **cursor**, which is a handle for the session-specific **private SQL area** that holds a parsed SQL statement and other processing information. The cursor and private SQL area are in the program global area (PGA).

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

- Syntax Check
- Semantic Check
- Shared Pool Check

Parsing- Syntax Check



Oracle Database must check each SQL statement for syntactic validity.

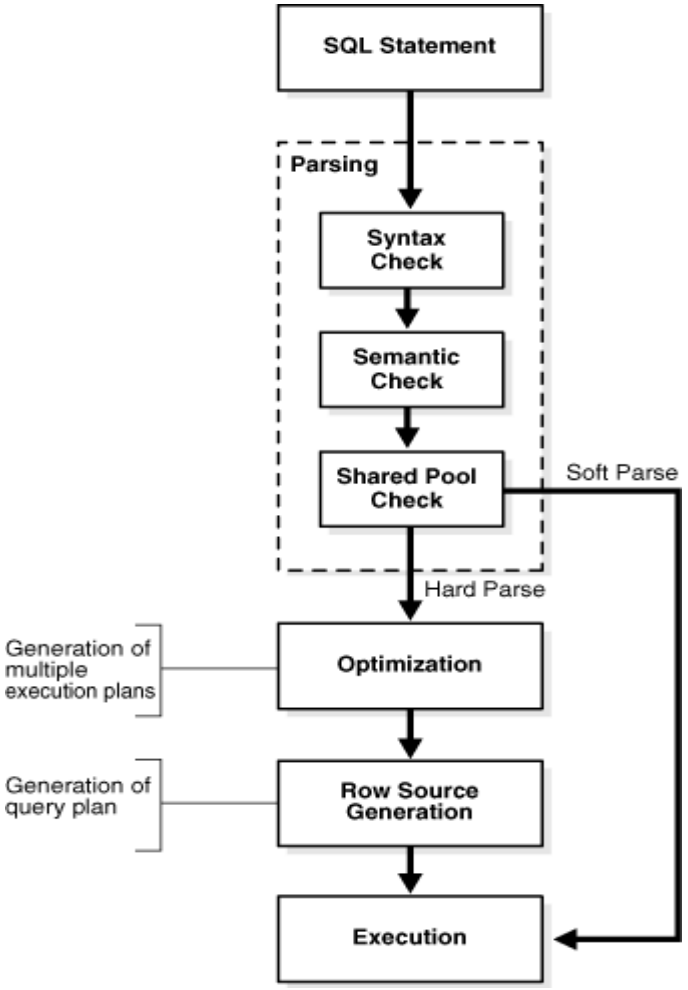
A statement that breaks a rule for well-formed SQL syntax fails the check.

For example, the following statement fails because the keyword FROM is misspelled as FORM:

SQL> SELECT * FORM employees;

* ERROR at line 1: ORA-00923: FROM keyword not found where expected

Parsing- Semantic Check



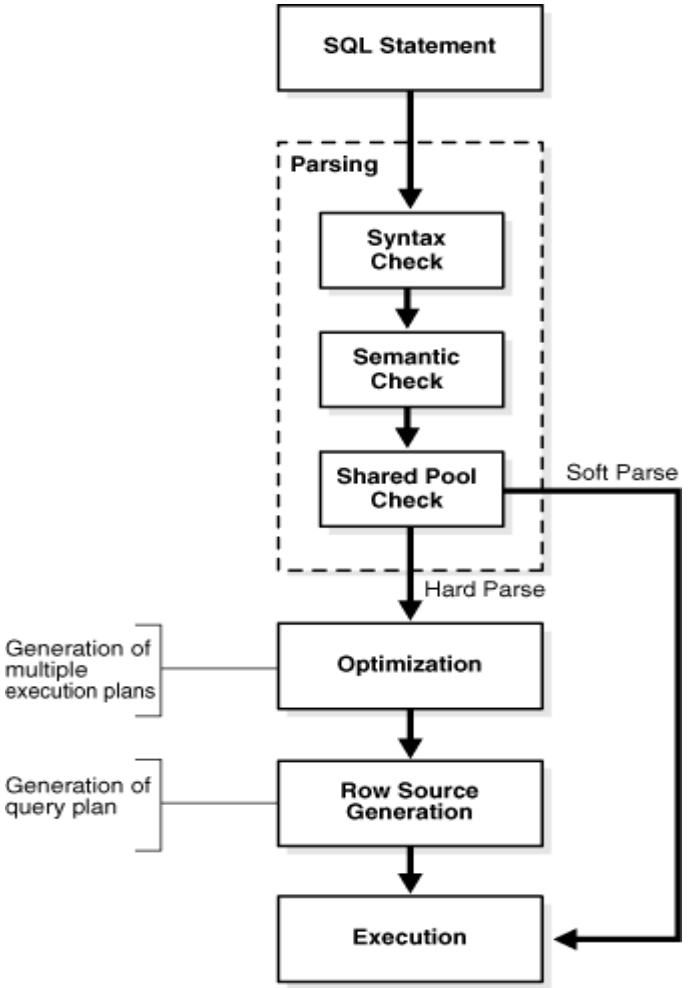
The semantics of a statement are its meaning. A semantic check determines whether a statement is meaningful, for example, whether the objects and columns in the statement exist.

A syntactically correct statement can fail a semantic check, as shown in the following example of a query of a nonexistent table:

SQL> SELECT * FROM nonexistent_table;
SELECT * FROM nonexistent_table

* ERROR at line 1: ORA-00942: table or view does not exist

Parsing- Semantic Check



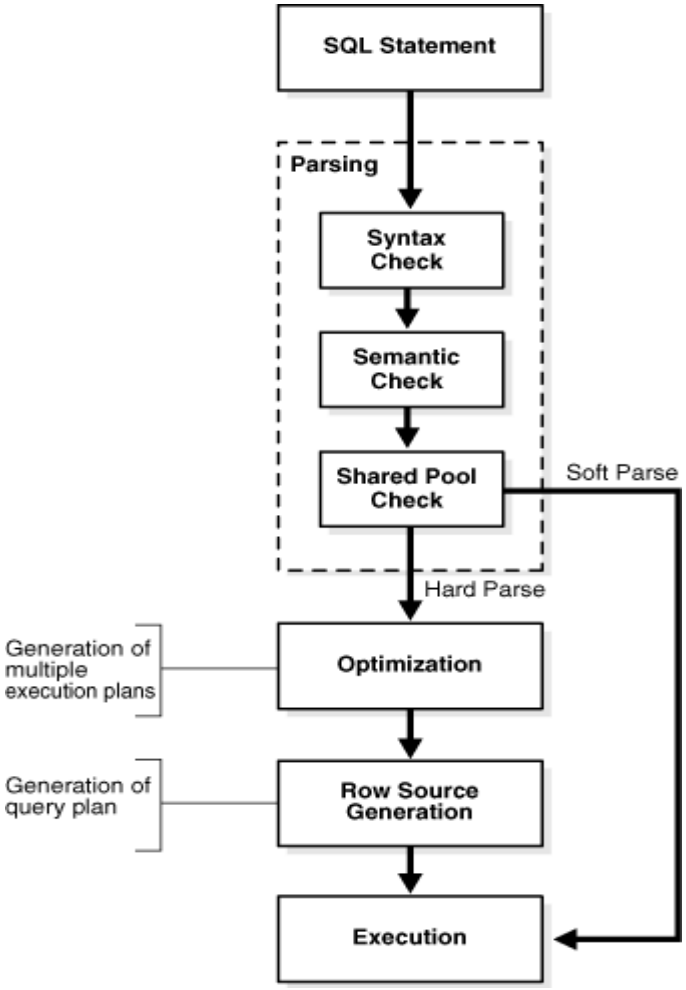
The semantics of a statement are its meaning. A semantic check determines whether a statement is meaningful, for example, whether the objects and columns in the statement exist.

A syntactically correct statement can fail a semantic check, as shown in the following example of a query of a nonexistent table:

SQL> SELECT * FROM nonexistent_table;

* ERROR at line 1: ORA-00942: table or view does not exist

Parsing- Shared Pool Check



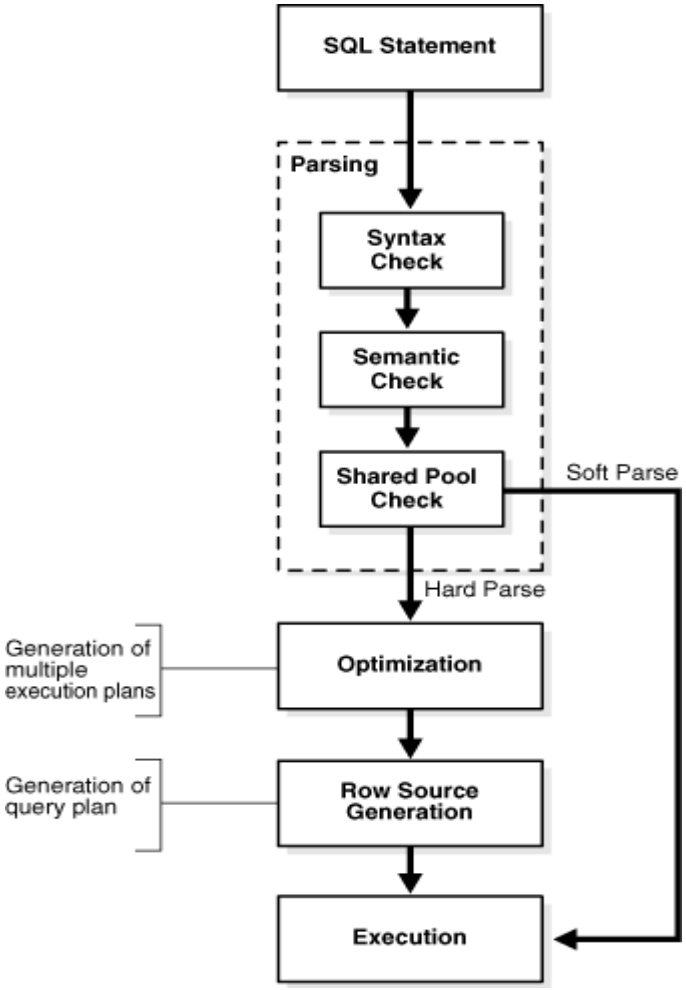
During the parse, the database performs a shared pool check to determine whether it can skip resource-intensive steps of statement processing.

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** to see if an existing parsed statement has the same hash value. Hash value of an **execution plan** for the statement.

A SQL statement can have multiple plans in the shared pool. Typically, each plan has a different hash value. If the same SQL ID has multiple plan hash values, then the database knows that multiple plans exist for this SQL ID.

Optimization



The optimizer attempts to generate the **most optimal execution plan** for a SQL statement.

The optimizer choose the **plan** with the **lowest cost** among all considered candidate plans. The optimizer uses available statistics to calculate cost. For a specific query in a given environment, the cost computation accounts for factors of query execution such as I/O, CPU, and communication.

For example, a query might request information about employees who are managers. If the optimizer statistics indicate that 80% of employees are managers, then the optimizer may decide that a full table scan is most efficient. However, if statistics indicate that very few employees are managers, then reading an index followed by a table access by rowid may be more efficient than a full table scan.

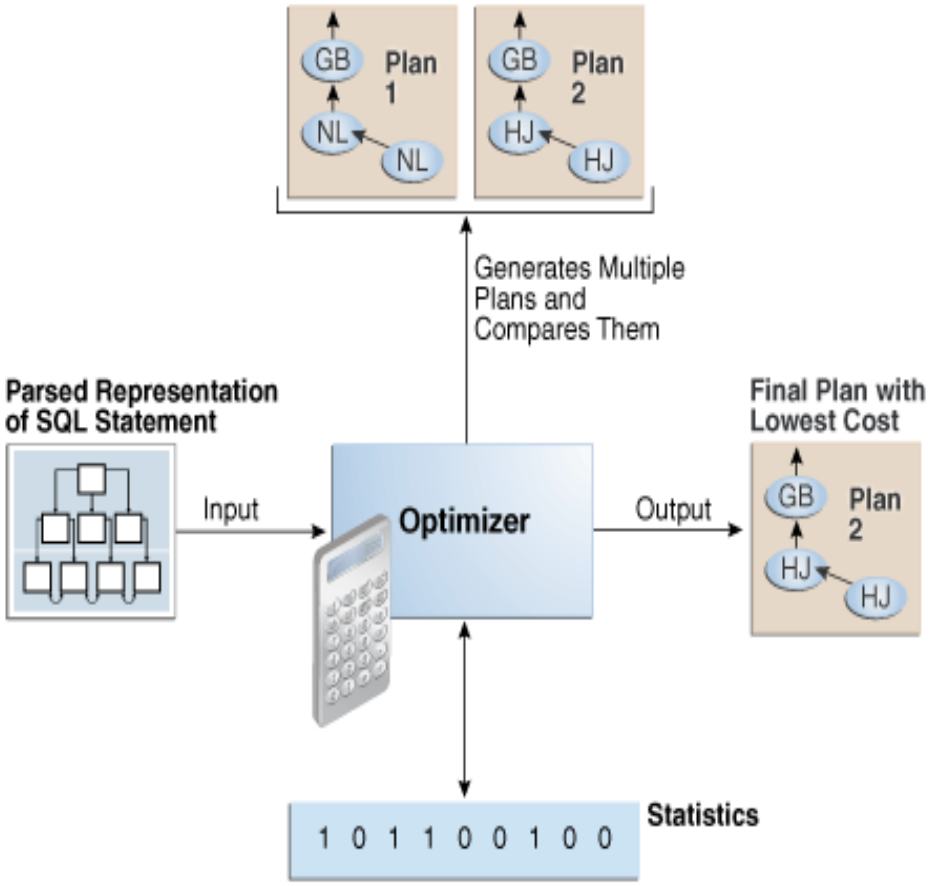
Cost based Optimization

Query optimization is the overall process of choosing the **most efficient** means of executing a SQL statement. SQL is a nonprocedural language, so the optimizer is free to merge, reorganize, and process in any order.

The database optimizes each SQL statement **based on statistics** collected about the accessed data. The optimizer determines the optimal plan for a SQL statement by examining multiple access methods, such as full table scan or index scans, different join methods such as nested loops and hash joins, different join orders, and possible transformations.

For a given query and environment, the optimizer assigns a relative numerical cost to each step of a possible plan, and then factors these values together to generate an overall cost estimate for the plan. After calculating the costs of alternative plans, the optimizer chooses the plan with the lowest cost estimate. For this reason, the optimizer is sometimes called the **cost-based optimizer (CBO)** to contrast it with the legacy rule-based optimizer (RBO).

Execution Plans



An **execution plan** describes a **recommended method** of execution for a SQL statement.

The plan shows the combination of the steps Oracle Database uses to execute a SQL statement. Each step either retrieves rows of data physically from the database or prepares them for the user issuing the statement.

An execution plan displays the cost of the entire plan, indicated on line 0, and each separate operation. The cost is an internal unit that the execution plan only displays to allow for plan comparisons.

Thus, you cannot tune or change the cost value. In the following graphic, the optimizer generates two possible execution plans for an input SQL statement, uses statistics to estimate their costs, compares their costs, and then chooses the plan with the lowest cost.

Query Blocks

The input to the optimizer is a parsed representation of a SQL statement.

Each SELECT block in the original SQL statement is represented internally by a **query block**. A query block can be a top-level statement, subquery, or unmerged view.

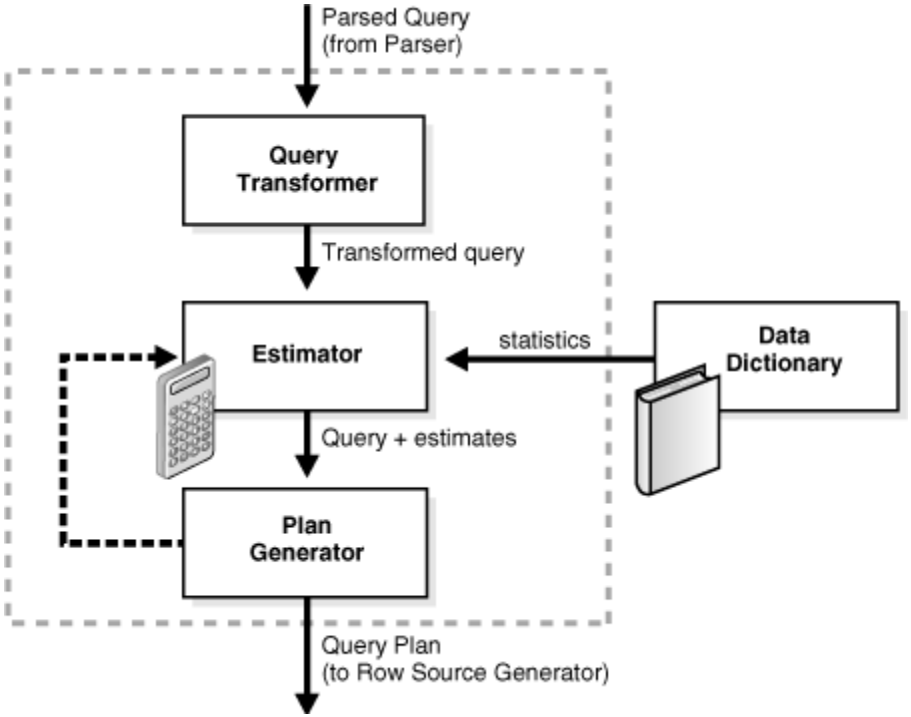
Example of Query Blocks
The following SQL statement consists of two query blocks. The subquery in parentheses is the inner query block. The outer query block, which is the rest of the SQL statement, retrieves names of employees in the departments whose IDs were supplied by the subquery. The query form determines how query blocks are interrelated.

```
SELECT first_name, last_name
        FROM hr.employees WHERE department_id IN
                (SELECT department_id
                FROM hr.departments
                WHERE location_id = 1800);
```

Query Subplans
For each query block, the optimizer generates a query subplan. The database optimizes query blocks separately from the bottom up. Thus, the database optimizes the innermost query block first and generates a subplan for it, and then generates the outer query block representing the entire query.

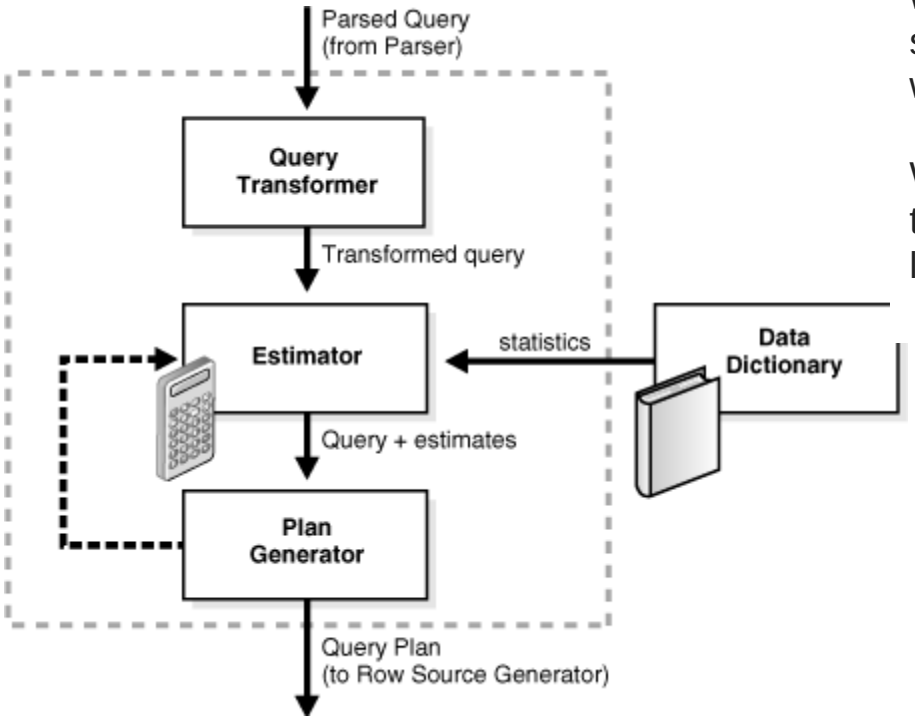
The number of possible plans for a query block is proportional to the number of objects in the FROM clause. This number rises exponentially with the number of objects. For example, the possible plans for a join of five tables are significantly higher than the possible plans for a join of two tables.

Optimizer Components



Operation	Description
Query Transformer	The optimizer determines whether it is helpful to change the form of the query so that the optimizer can generate a better execution plan.
Estimator	The optimizer estimates the cost of each plan based on statistics in the data dictionary.
Plan Generator	The optimizer compares the costs of plans and chooses the lowest-cost plan, known as the execution plan, to pass to the row source generator.

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.

When a viable alternative exists, the database calculates the cost of the alternatives separately and chooses the lowest-cost alternative.

```
SELECT *
FROM   sales
WHERE  promo_id=33
OR     prod_id=136;
```

Query Transformer

```
SELECT *
FROM   sales
WHERE  prod_id=136
UNION  ALL
SELECT *
FROM   sales
WHERE  promo_id=33
AND    LNNVL(prod_id=136);
```

Equivalent Query Trees

Two queries are equivalent if their output tables are always equivalent, regardless of the contents of the database.

Equivalent query trees can be formed by

- rearranging products
- splitting selections
- moving selections within the tree
- identifying joins
- moving aggregations inside a join
- adding projections

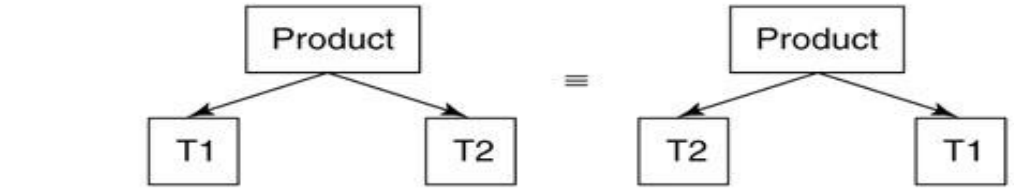
Rearranging Products

The product operator is commutative;

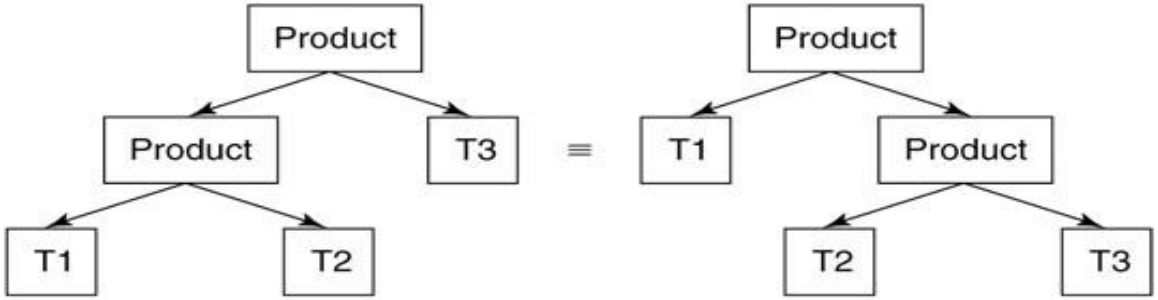
product(T1, T2) ≡ product(T2, T1)

The product operator is associative;

product(product(T1, T2), T3) ≡ product(T1, product(T2, T3))



(a) The *product* operator is commutative



(b) The *product* operator is associative

Figure 24-1
Equivalences involving the *product* operator

Rearranging Products

```
select SName
from   STUDENT, ENROLL,
       SECTION, COURSE,
       DEPT
```

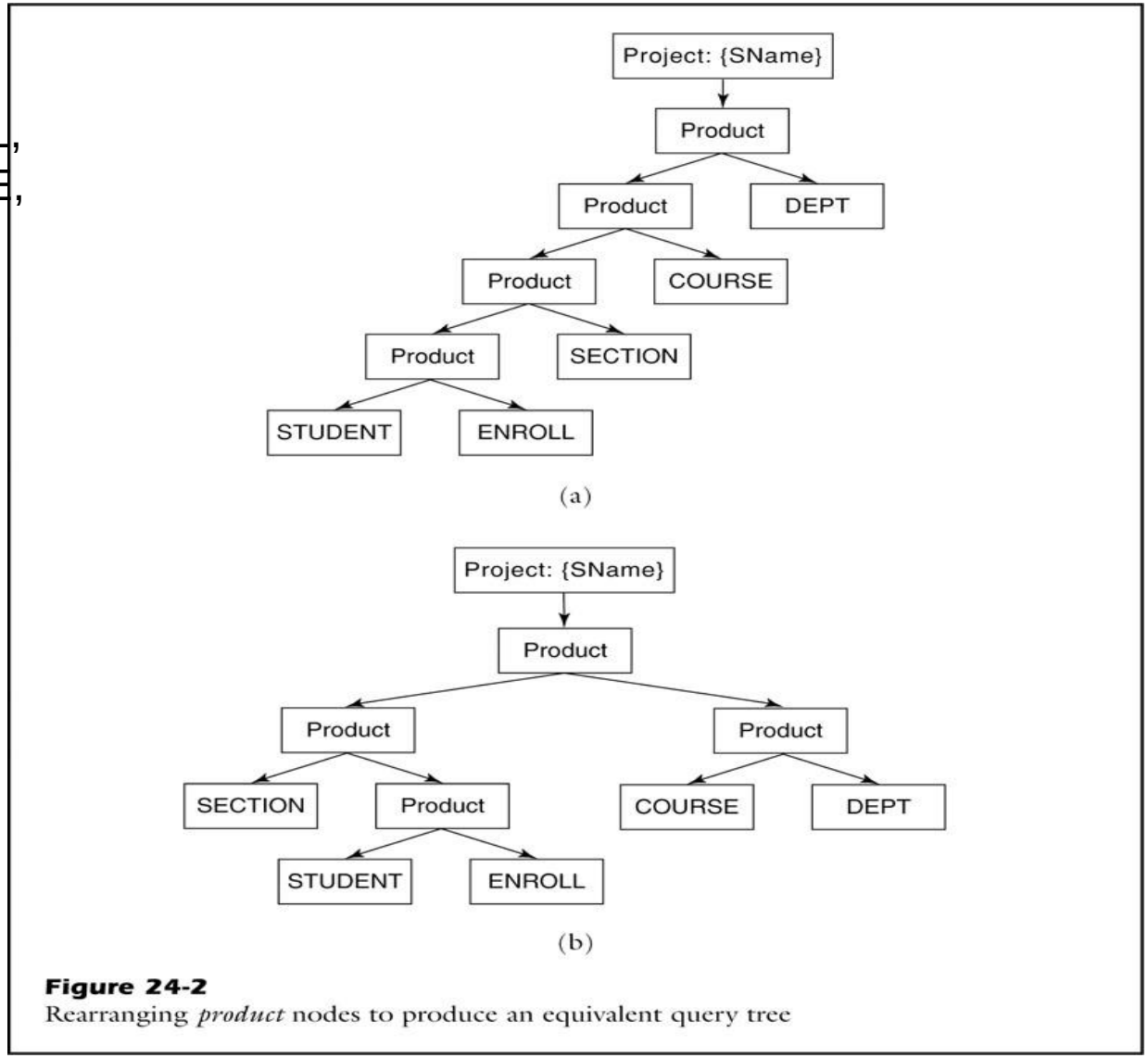


Figure 24-2
Rearranging *product* nodes to produce an equivalent query tree

Splitting Selections

Suppose that predicate p is the conjunction of two predicates p1 and p2. Instead of using p directly, we can find the records satisfying p in two steps: First we find the records satisfying p1, and then from that set we find the records satisfying p2.

select(T, p1 and p2) ≡ select(select(T, p1), p2)

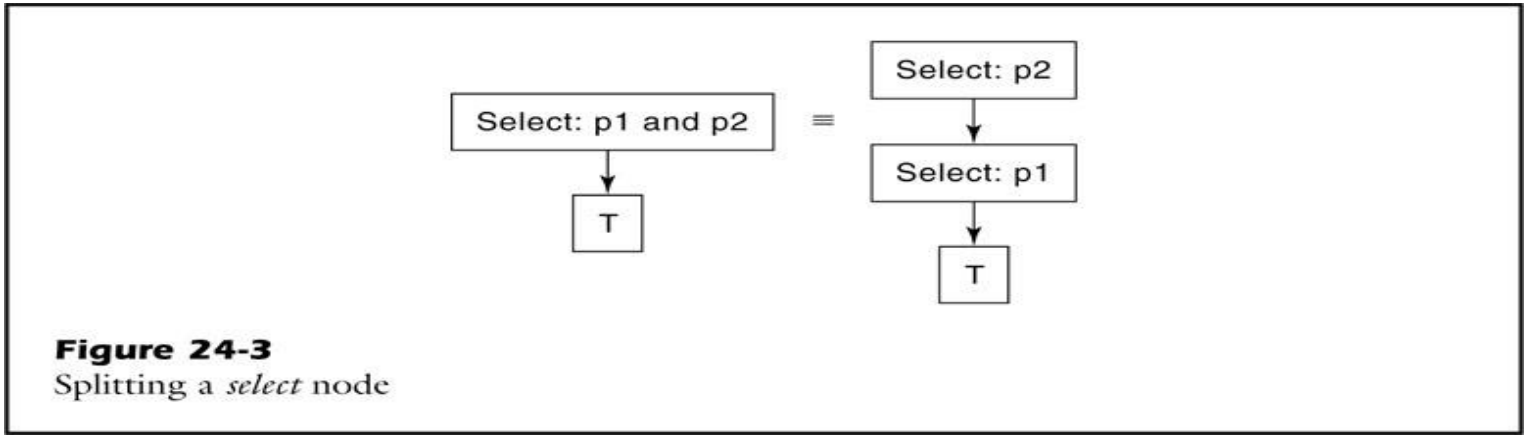


Figure 24-3
Splitting a *select* node

The query optimizer strives to split predicates into as many conjuncts as possible. It does so by transforming each predicate into **conjunctive normal form (or CNF)**.

A predicate is in CNF if it is a conjunction of sub-predicates, none of which

Conjunctive Normal Form (CNF)

```
select SName
from STUDENT
where (MajorId=10 and SId=3) or(GradYear=2004)
```

```
select SName
from STUDENT
where (MajorId=10 or GradYear=2004) and
        (SID=3 or GradYear=2004)
```

Moving Selections

```
select SName
from STUDENT, DEPT
where (DName = 'math')
    and (MajorId = DId )
```

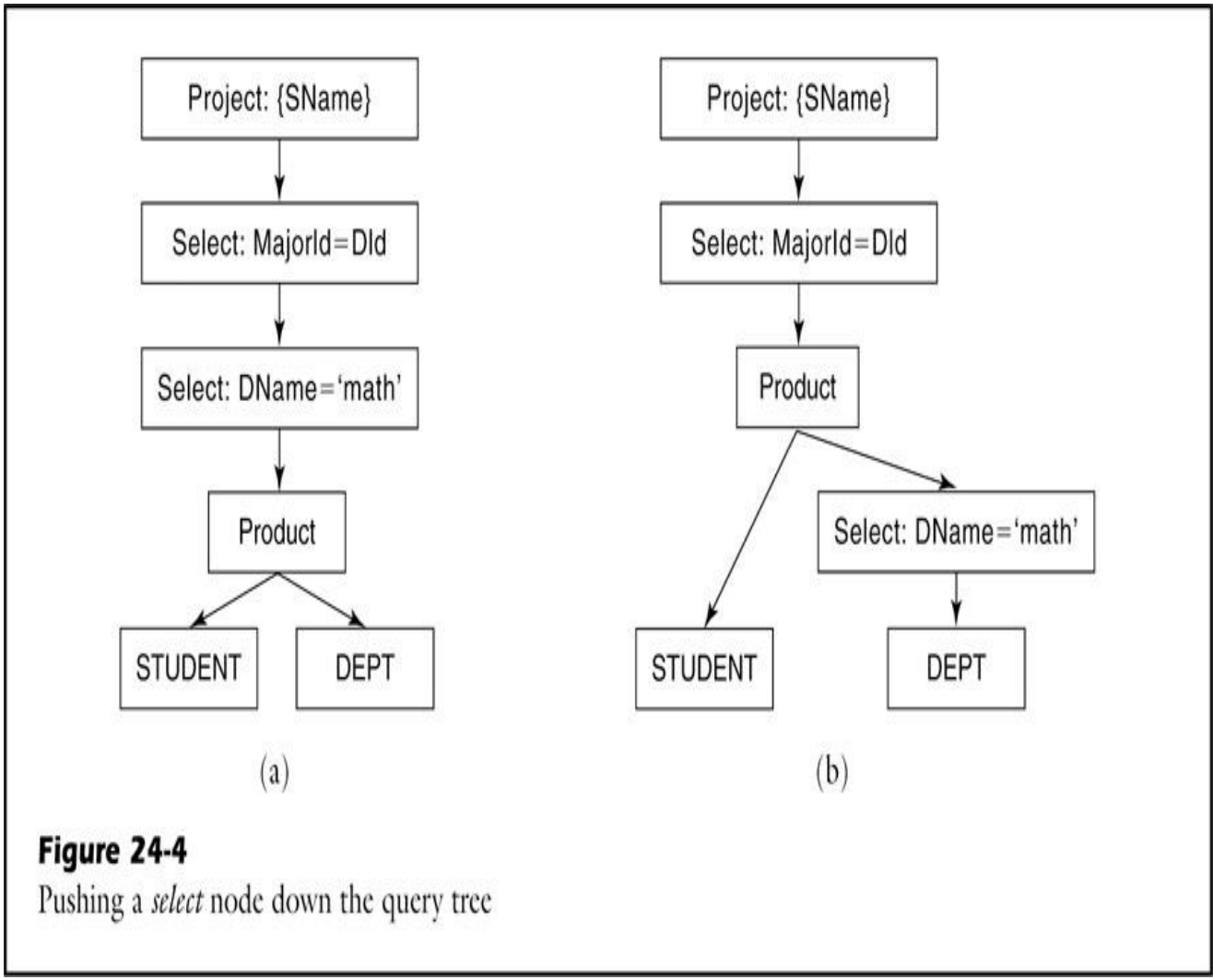


Figure 24-4
Pushing a *select* node down the query tree

Moving Selections

select(product(T1, T2), p) ≡ product(select(T1, p), T2)

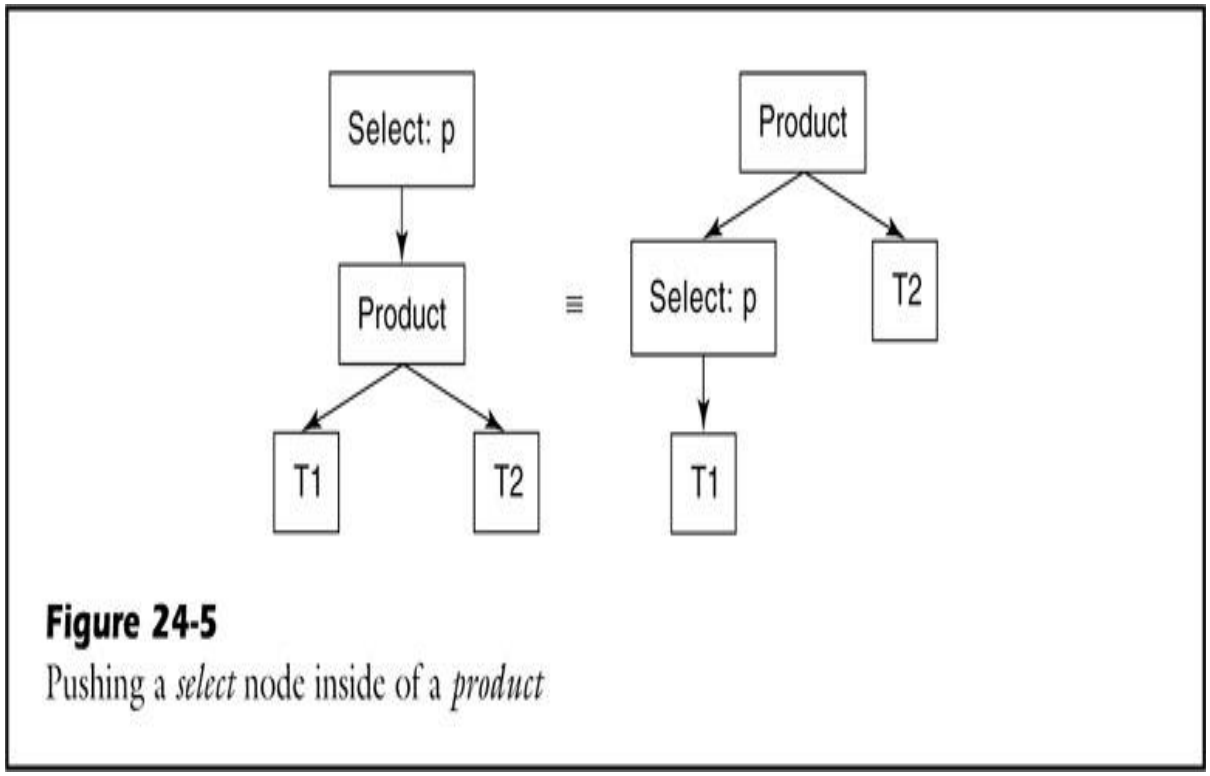
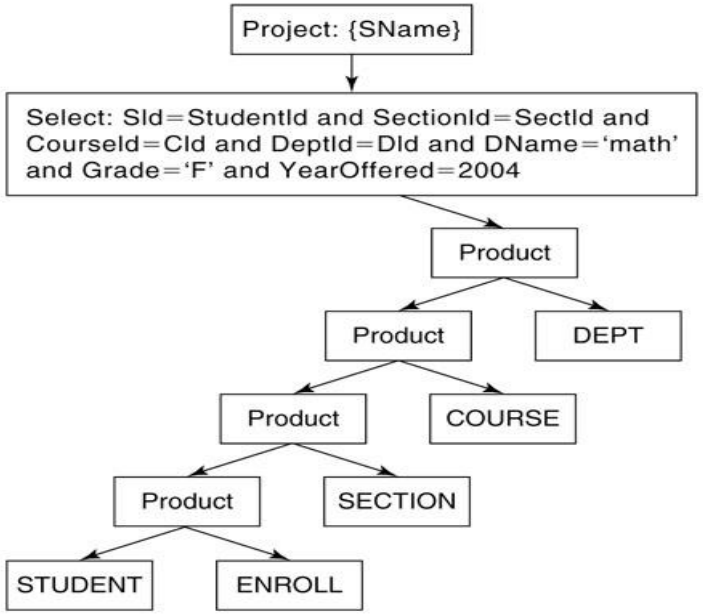


Figure 24-5
Pushing a *select* node inside of a *product*

Pushing Selections Down

```
select SName
from STUDENT, ENROLL, SECTION, COURSE, DEPT
where SId=StudentId and SectionId=SectId
and CourseId=CId and DeptId=DId
and DName='math' and Grade='F' and YearOffered=2004
```

(a) The SQL query

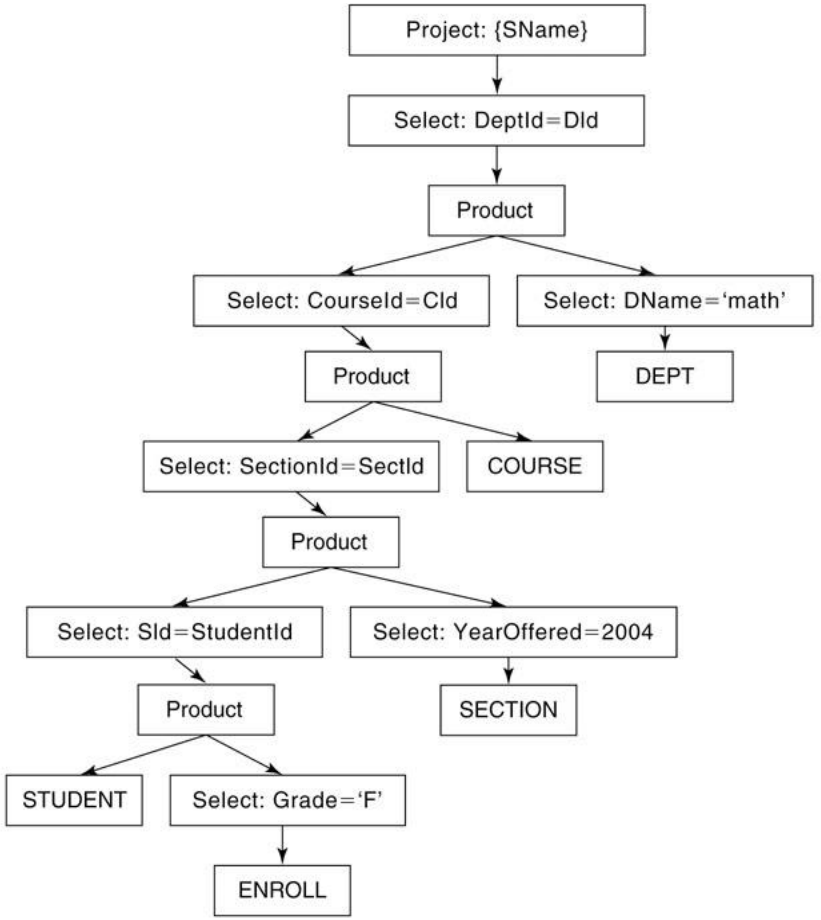


(b) The query tree created by the basic planner

Figure 24-6
Pushing selections down a query tree

Pushing Selections Down

```
select SName
from STUDENT, ENROLL, SECTION, COURSE, DEPT
where (SId=StudentId) and
      (SectionId=SectId) and
      (CourseId=Cid) and
      (DeptId=Did) and
      (DName='math') and
      (Grade='F') and
      (YearOffered=2004)
```



(c) The query tree resulting from pushing *select* nodes

Figure 24-6 *(Continued)*

Identifying Joins

```
select SName
from STUDENT, ENROLL, SECTION, COURSE, DEPT
where (SId=StudentId) and
      (SectionId=SectId) and
      (CourseId=Cid) and
      (DeptId=Did) and
      (DName='math') and
      (Grade='F') and
      (YearOffered=2004)
```

join(T1, T2, p) ≡ select(product(T1, T2), p)

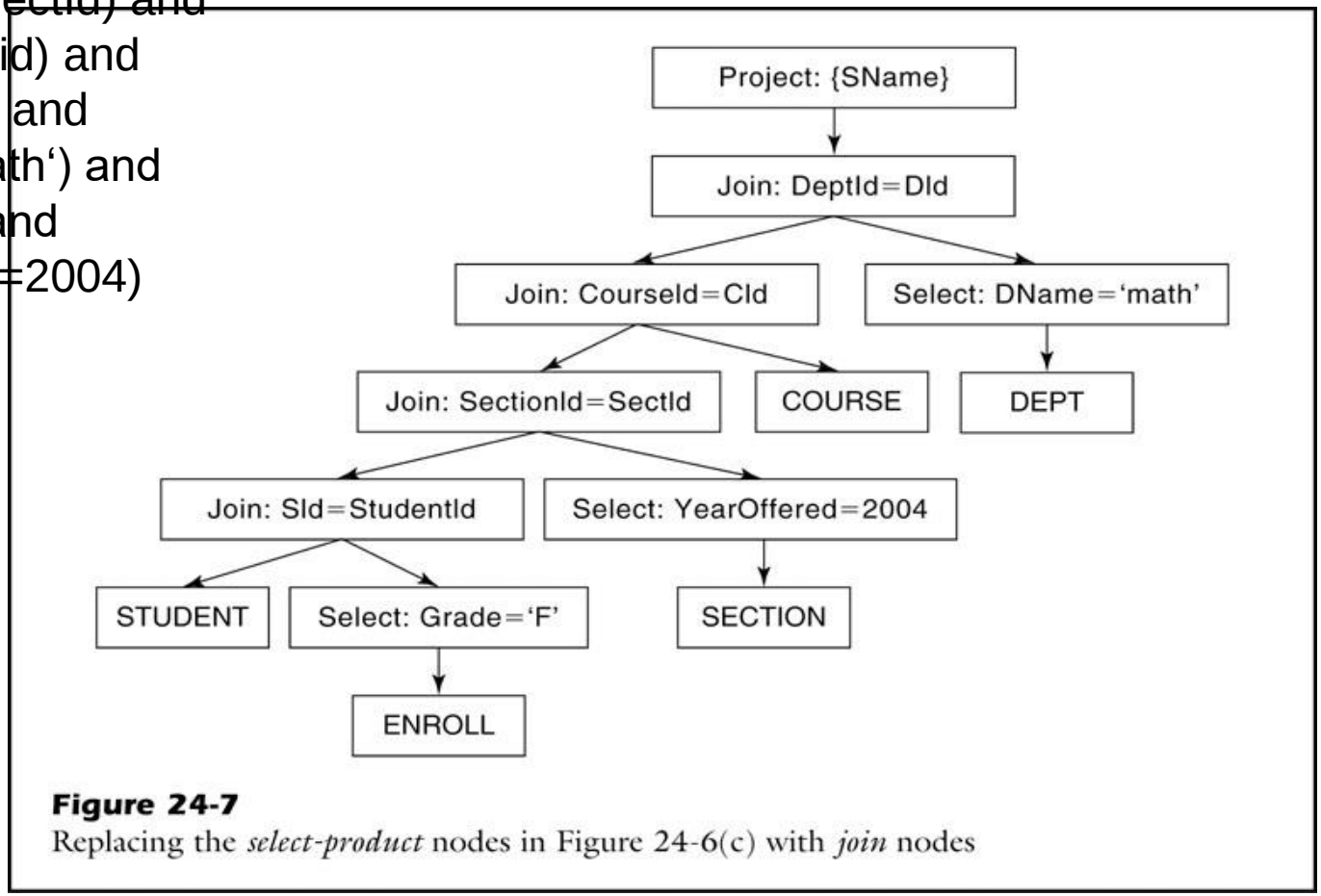
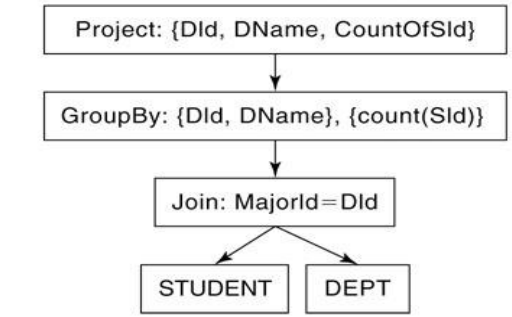


Figure 24-7
Replacing the *select-product* nodes in Figure 24-6(c) with *join* nodes

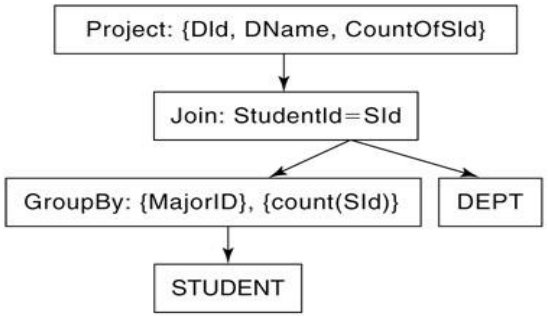
Moving Aggregations

```
select DId, DName, Count(SId)
from STUDENT, DEPT
where MajorId = DId
group by DId, DName
```

(a) The SQL query



(b) The aggregation placed after the join



(c) Pushing the aggregation inside the join

Figure 24-8
Transforming a query tree having a *groupby* node

Moving Aggregations

Generalization of the example;
The join predicate equates the foreign key of table T1 with the key of table T2.
The grouping fields in the groupby node contain only fields from T2, and include the key.
The aggregation functions mention only fields from T1.

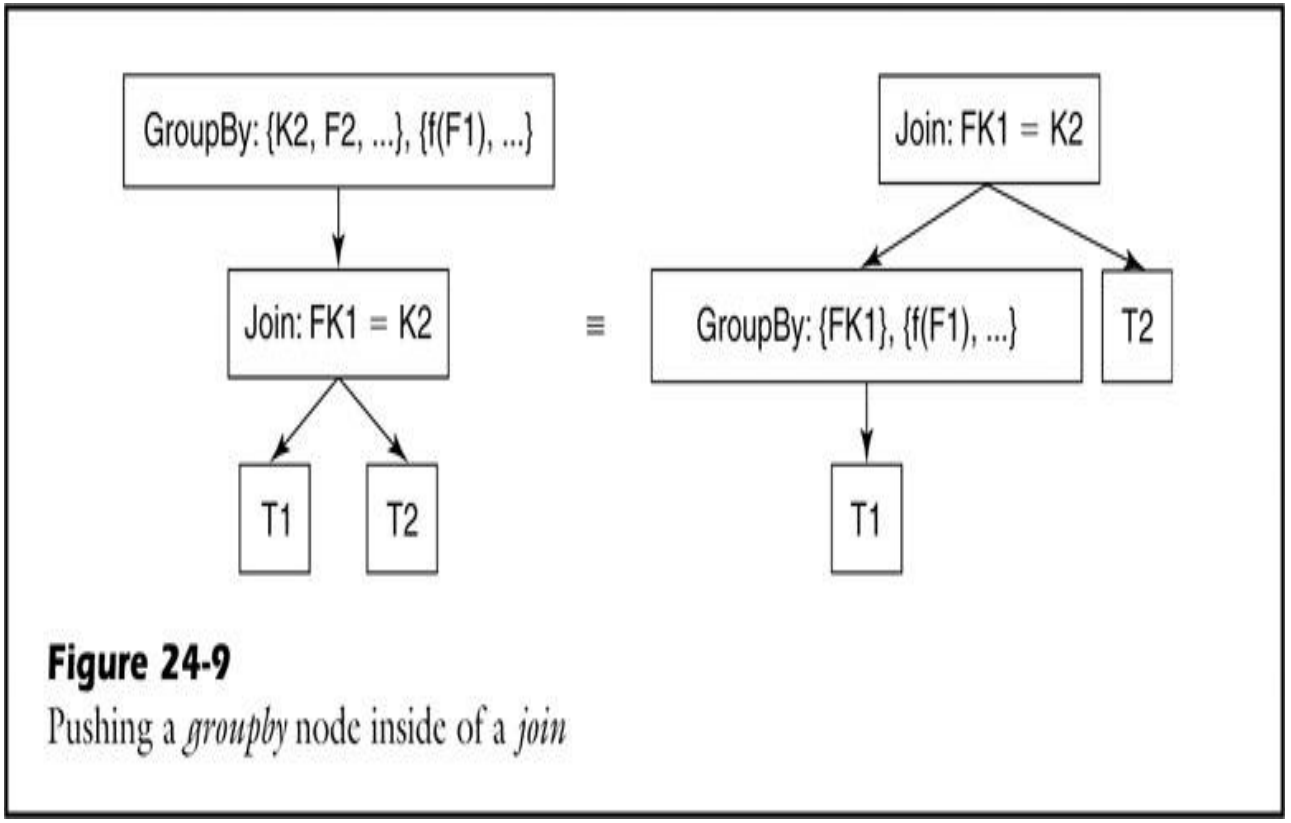


Figure 24-9
Pushing a *groupby* node inside of a *join*

Adding Projections

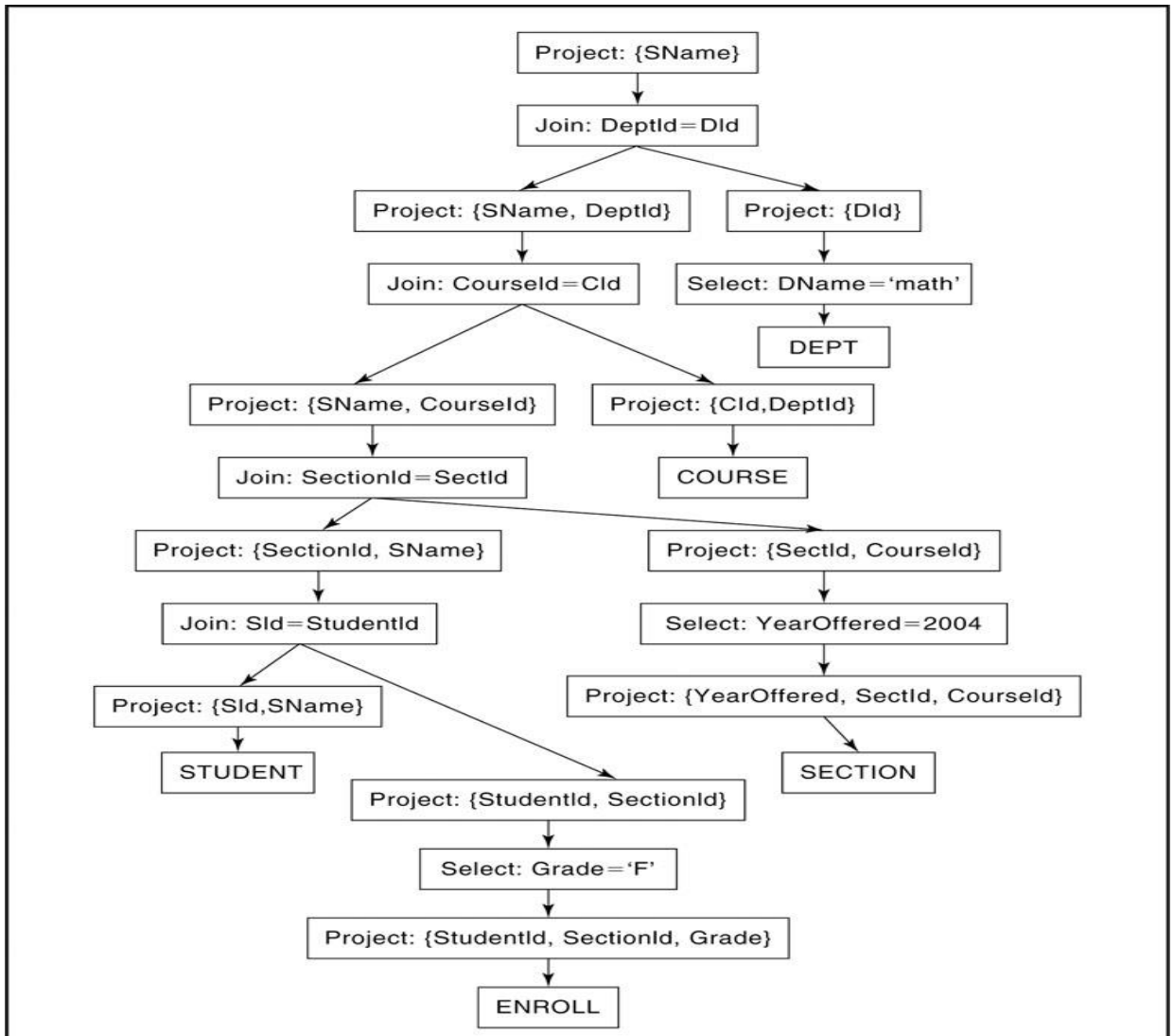
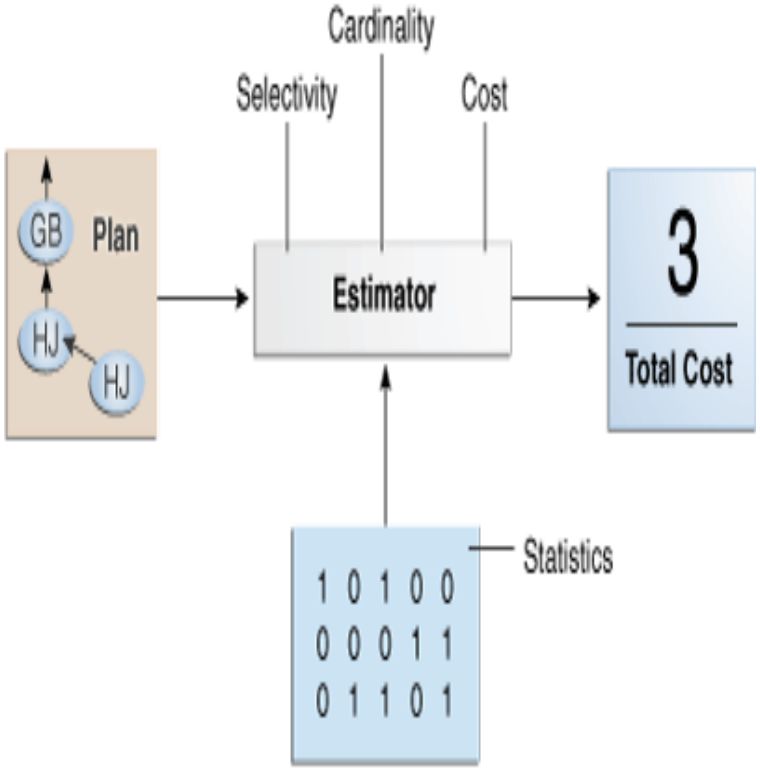


Figure 24-10
Adding projections to the query tree of Figure 24-7

Estimator



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

The estimator uses three different measures to determine cost:

•Selectivity

The percentage of rows in the row set that the query selects, with 0 meaning no rows and 1 meaning all rows.

Selectivity is tied to a query predicate, such as WHERE last_name LIKE 'A%', or a combination of predicates. A predicate becomes more selective as the selectivity value approaches 0 and less selective (or more unselective) as the value approaches 1.

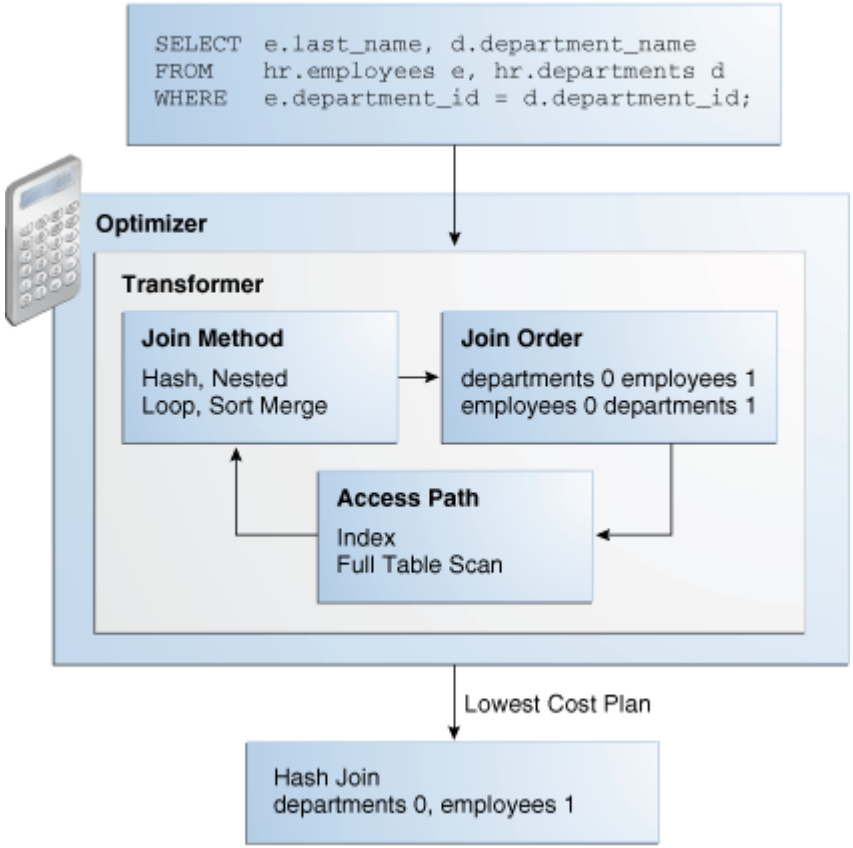
•Cardinality

The cardinality is the number of rows returned by each operation in an execution plan. This input, which is crucial to obtaining an optimal plan, is common to all cost functions. The estimator can derive cardinality from the table statistics collected by DBMS_STATS, or derive it after accounting for effects from predicates (filter, join, and so on), DISTINCT or GROUP BY operations, and so on.
The Rows column in an execution plan shows the estimated cardinality.

•Cost

This measure represents units of work or resource used. The query optimizer uses disk I/O, CPU usage, and memory usage as units of work.

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.

Typical SQL Performance Problems

- Inefficiently designed SQL statements (user)
- Suboptimal execution plans (optimizer)
- Missing SQL access structures
- Stale optimizer statistics
- Hardware problems

Inefficiently designed SQL statements

If a SQL statement is written so that it performs unnecessary work, then the optimizer cannot do much to improve its performance. Examples of inefficient design include

- Neglecting to add a **join condition**, which leads to a **Cartesian product**
- Using hints to specify a large table as the **driving table** in a join
- Specifying UNION instead of UNION ALL
- Making a **subquery** execute for every row in an outer query

Suboptimal Execution Plans

The **query optimizer** (also called the **optimizer**) is internal software that determines which **execution plan** is most efficient. Sometimes the optimizer chooses a plan with a suboptimal **access path**, which is the means by which the database retrieves data from the database.

For example, the plan for a query predicate with low **selectivity** may use a **full table scan** on a large table instead of an index. You can compare the execution plan of an optimally performing SQL statement to the plan of the statement when it performs suboptimally. This comparison, along with information such as changes in data volumes, can help identify causes of performance degradation.

Missing SQL Access Structures

Absence of SQL access structures, such as indexes and materialized views, is a typical reason for suboptimal SQL performance.

The optimal set of access structures can improve SQL performance by orders of magnitude.

Stale Statistics

Statistics gathered by DBMS_STATS can become stale (incorrect/missing) when the statistics maintenance operations, either automatic or manual, cannot keep up with the changes to the table data caused by DML (insert/delete/update).

Because stale statistics on a table do not accurately reflect the table data, the optimizer can make decisions based on faulty information and generate suboptimal execution plans.

HW Problems

Suboptimal performance might be connected with
- memory,
- I/O, and
- CPU problems.

Conclusion

SQL Processing is composed of Parsing, Optimization, Execution Plan Generation and Execution Steps.

The optimizer attempts to generate the **most optimal execution plan** for a SQL statement.

Query optimizer has 3 components: Query Transformer, Estimator and Plan Generator.

Statistics are important for query optimizer. They can be kept up-to-date automatically (start-up, frequently or instant) or manually.

Typical SQL Performance Problems can be due to inefficiently designed SQL statements, suboptimal execution plans, missing SQL access structures, stale statistics and HW problems.

References

Oracle Database Online Documentation 12c, Release 1 (12.1) / Database Administration

Database SQL Tuning Guide