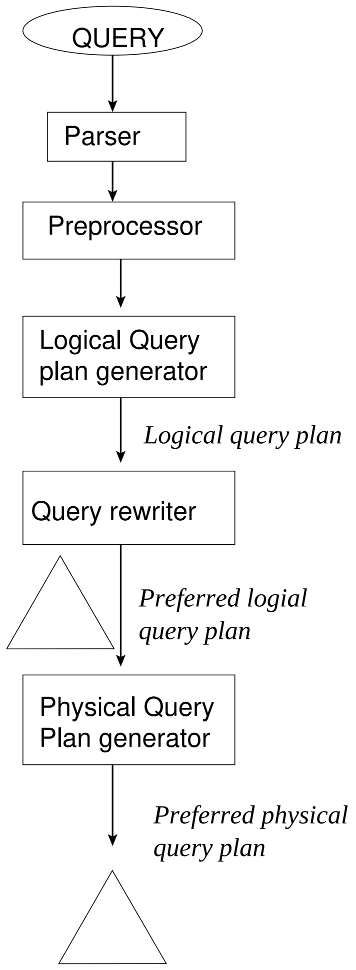
Query Processing: an Overview

Query Processing in a Nutshell



Parser

The Parser is responsible for translating the query from SQL into a *parse tree*.

Preprocessor

The Preprocessor is responsible for *semantic checking*

- Relation names (all relations are "correct")
- Attribute uses (resolve all attribute mentions)
- Types (typecheck all expressions)

Logical Query Plan Generator

The parser produces a parse tree that identifies the `SELECT`, `FROM`, `WHERE` and other clauses of the SQL query. The Logical Query Plan Generator converts the parse tree them into a relational algebra expression, and returns the tree corresponding to this expression. This tree is called the initial *logical query plan*.

A number of cases needs to be considered.

SELECT-FROM-WHERE statement with no nested queries. In this case, the conversion is straighforward (from innermost operation to the outtermost operation):

1. Product ($\times$) of all relations in the `FROM` clause;
2. Selection (σ) on all conditions in the `WHERE` clause;
3. Projection (π) onto all attributes in the `SELECT` list.

SELECT-FROM-WHERE statement with uncorrelated nested queries. The transformation is done in two steps.

Step 1. The outter query tree is built for the non-nester part of the query. The inner query tree is built. The inner query tree is attached to the outter query tree using a two-argument selection node.

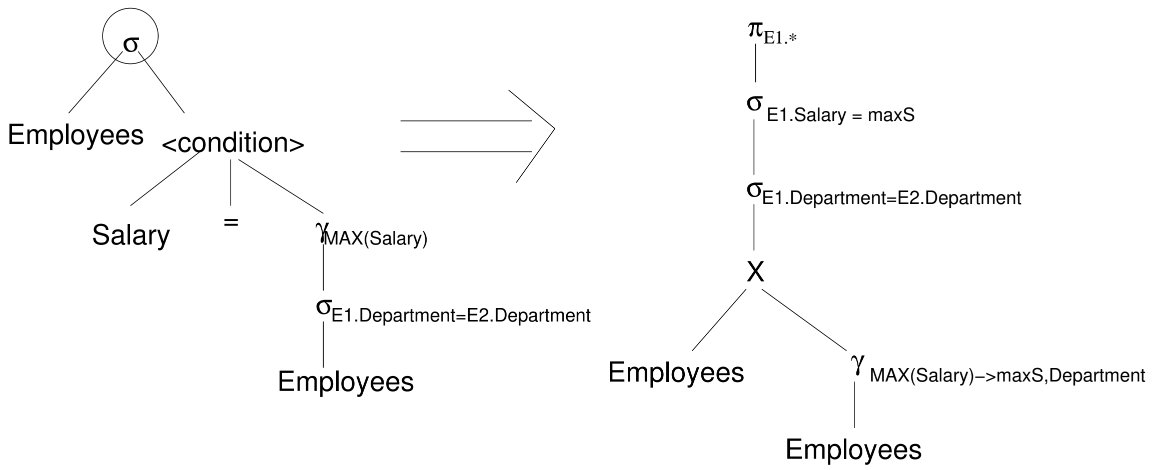
This node is marked with selection operation σ and has two children. First child is a subtree representing the outter relation The second child is the inner query subtree.

Step 2. The two-argument selection node is replaced with a **product** or **join** node.

Example. Consider the following SQL statement

```
SELECT *
FROM Employee
WHERE Employee.department IN (SELECT ID
                              FROM Departments
WHERE Name = 'Toys');
```

The figure below shows the two stages:



Note: Uncorrelated queries allow the logical query plan generator to replace the two-argument select with a product, whose second argument is the result of the nested query.

SELECT-FROM-WHERE statement with correlated nested queries The transformation is done in two steps. The first step is the same as for the uncorrelated nested query, the second step is somewhat more complex.

Step 1. The outter query tree is built for the non-nester part of the query. The inner query tree is built. The inner query tree is attached to the outter query tree using a two-argument selection node.

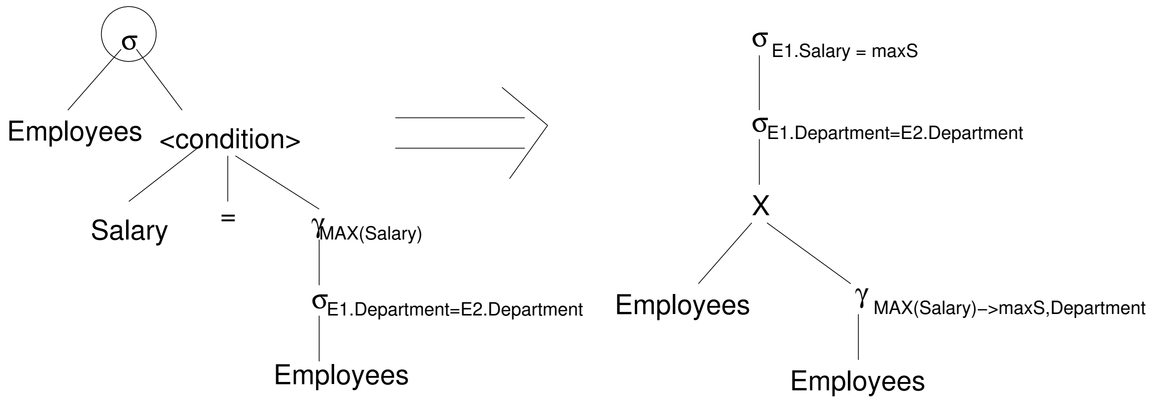
This node is marked with selection operation σ and has two children. First child is a subtree representing the outter relation The second child is the inner query subtree.

Step 2. The two-argument selection node is replaced with a **product** node. The correlated selection condition from the right-hand-side subtree of the two-argument selection is *moved above the product*.

Example. Consider the following SQL statement

```
SELECT *
FROM Employee E1
WHERE Employee.Salary = (SELECT MAX(Salary)
                         FROM Employee E2
  WHERE E1.Department = E2.Department)
```

The figure below depicts the two steps:



Note: The conversion in Step 2 is done as follows:

- The grouping/aggregation operator (or the projection operator) is expanded to include the attribute(s) participating in the correlated conditions.
- Selection node for the correlated condition is deleted from the tree.
- The two-argument selection node is replaced with a product node.
- Two selection nodes are pus above the product node. The first node represents the correlated condition(s) from the nested query. The second node represents the selection condition from the outter query that involved the nested query.
- If needed, a projection node, projecting out all attributes from the result of the "nested" query is added as the root of the logical query plan.

Rewriting Logical Query Plans

Logical query plan created on the previous stage may not be the best possible way to execute the query. The logical query plan rewriting stage analyzes the original logical query plan and suggests one or more better logical query plans.

Rewriting logical query plans is based on query rewrite rules. Query rewrite rules are equivalences of the underlying relational algebra. The list of query rewrite rules for the relational algebra is presented in a separate handout. Query rewriting proceeds as follows:

- A rewrite rule is selected.
- A sub-expression in the current logical query plan which has the same structure as one of the sides of the rewrite rule is identified.
- Any conditions required for the rule application are verified.
- The selected subexpression is replaced with the other side of the rewrite rule.

Example. Consider the following relational algebra expression, representing a logical query plan:

$$\sigma_{R.A=S.B}(\pi_{R.A,R.C,S.B}(R \times S)) \bowtie T.$$

We select the following relational algebra equivalence, as our rewrite rule:

$$\sigma_C(\pi_L(R)) \equiv \pi_L(\sigma_C(R)).$$

This rule is applicable iff conditions in C mention only attributes in L.

We identify the part of the logical query plan that matches the left-hand side of the rewrite rule:

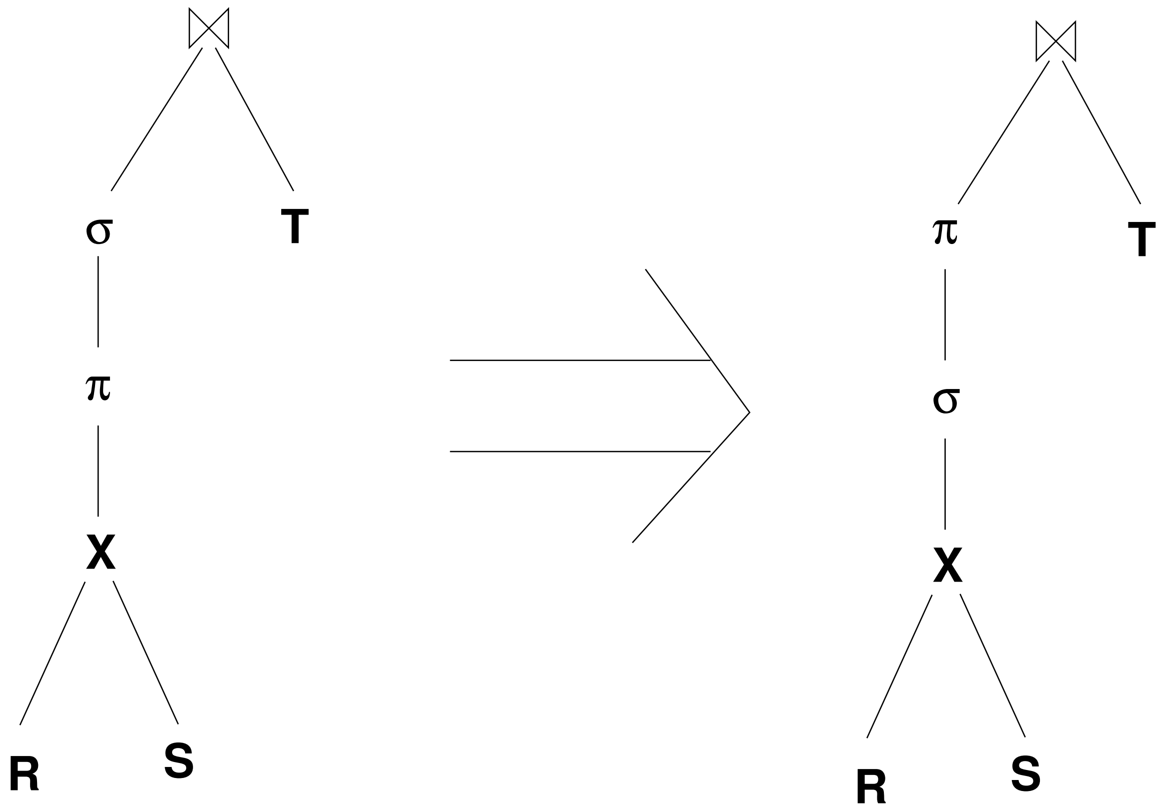
$$\sigma_{\mathbf{R.A=S.B}}(\pi_{\mathbf{R.A,R.C,S.B}}(\mathbf{R} \times \mathbf{S})) \bowtie T$$

We verify that our selection condition contains only attributes mentioned in the projection list. [1].

Finally, we are replacing the left-hand side of the rewrite rule with the right-hand side:

$$\pi_{R.A,R.C,S.B}(\sigma_{R.A=S.B}(R \times S)) \bowtie T.$$

As a tree, the quey plan will be rewritten as follows:



How do we determine, which logical query plan is better?

Typically, it is done in one of two ways:

1. Cost-based estimation.

2. Use of heuristic rules.

[1] Although, for our application this is not necessary. This condition is important when we are trying to push projection inside the selection.

Cost-based estimation will be discussed together with physical plan construction.

Heuristic rules are rules which specify, which direction of a rewrite rule is more favorable and which rewrite rules should be preferred.

Traditional heuristic rules in a logical query rewrite system include:

- Pushing down selections. By pushing selection operations down (i.e., making them execute as early as possible), we usually are decreasing the sizes of relations with which we need to work earlier.

  Note: in some rare cases, involving nested queries or views, selections can actually be pushed up first, and then pushed down on more than one path of the tree for best effect.

  Note: Selection conditions combined with an `AND` connective can be split and pushed down separately.

- Pushing down projections. Projection operation may not reduce the number of tuples, but will reduce the size of the relation with which other operations have to work. By pushing projections down, or, sometimes, by introducing new projections to reduce the size of the output, we may improve the cost of the query.

- Moving duplicate elimination operations. Duplicate eliminations can be costly. However, sometimes they can be eliminated, combined, or postponed. In some other cases, they can actually be pushed down and yield significantly smaller relations passed to the next operations.

- Eliminating cartesian product. Cartesian product operations can be combined with selection operations (and sometimes, with projection operations) which use data from both relations to form joins. Join execution algorithms are typically faster than a cartesian product followed by selecection (at the very least, because selection in this case may require an extra scan).

Physical Query Plan Optimization

Logical query rewriting can produce one *(or more candidate)* query plan(s). Physical query plan optimization stage involves the following operations:

- Selection of the order and grouping in which associative-and-commutative operations are to be executed.

- Selection of the appropriate algorithm for each logical query plan operator.

- Insertion of additional operations: scans, sorts, etc., needed for faster performance.

- Selection of the means of passing results of one operation to the next operation: through main memory buffer, through temporary disk storage or via tuple-at-a-time iterators.