Lecture #14: Query Planning & Optimization I

15-445/645 Database Systems (Fall 2020)
https://15445.courses.cs.cmu.edu/fall2020/
Carnegie Mellon University
Prof. Andy Pavlo

1 Overview

Because SQL is declarative, the query only tells the DBMS what to compute, but not how to compute it. Thus, the DBMS needs to translate a SQL statement into an executable query plan. But there are different ways to execute a query (e.g., join algorithms) and there will be differences in performance for these plans. The job of the DBMS's optimizer is to pick an optimal plan for any given query.

The first implementation of a query optimizer was IBM System R and was designed in the 1970s. Prior to this, people did not believe that a DBMS could ever construct a query plan better than a human. Many concepts and design decisions from the System R optimizer are still in use today.

There are two types of optimization strategies.

The first approach is to use static rules, or *heuristics*. Heuristics match portions of the query with known patterns to assemble a plan. These rules transform the query to remove inefficiencies. Although these rules may require consultation of the catalog to understand the structure of the data, they never need to examine the data itself.

An alternative approach is to use *cost-based search* to read the data and estimate the cost of executing equivalent plans. The cost model chooses the plan with the lowest cost.

Query optimization is the most difficult part of building a DBMS. There have been attempts to apply machine learning to the improve the accuracy and efficiency of optimizers. But no major DBMS has incorporated these new approaches.

Logical vs. Physical Plans

The optimizer generates a mapping of a *logical algebra expression* to the optimal equivalent physical algebra expression. The logical plan is roughly equivalent to the relational algebra expressions in the query.

Physical operators define a specific execution strategy using an access path for the different operators in the query plan. Physical plans may depend on the physical format of the data that is processed (i.e. sorting, compression).

There does not always exist a one-to-one mapping from logical to physical plans.

2 Relational Algebra Equivalence

Much of query optimization relies on the underlying concept that the high level properties of relational algebra are preserved across equivalent expressions. Two relational algebra expressions are *equivalent* if they generate the same set of tuples.

This technique of transforming the underlying relational algebra representation of a logical plan is known as *query rewriting*.

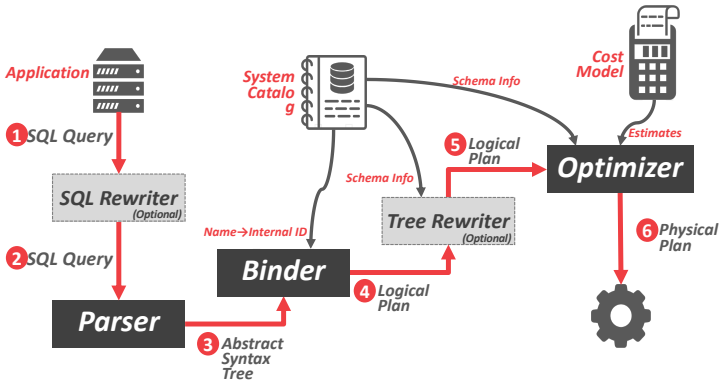

Figure 1: Architecture Overview – The application connected to the database system and sends a SQL query, which may be rewritten to a different format. The SQL string is parsed into tokens that make up the syntax tree. The binder converts named objects in the syntax tree to internal identifiers by consulting the system catalog. The binder emits a logical plan which may be fed to a tree rewriter for additional schema info. The logical plan is given to the optimizer which selects the most efficient procedure to execute the plan.

One example of relational algebra equivalence is *predicate pushdown*, in which a predicate is applied in a different position of the sequence to save work. Figure 2 shows an example of predicate pushdown.

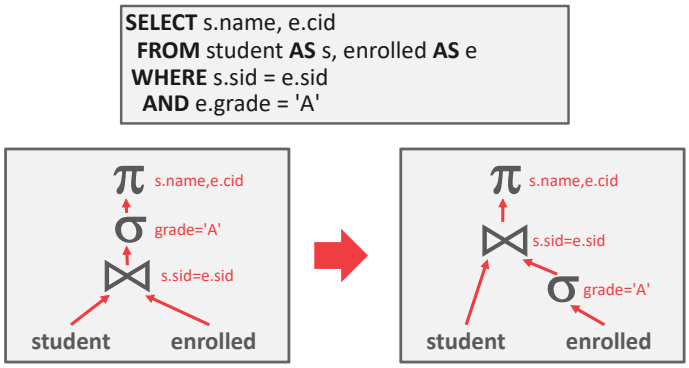

Figure 2: Predicate Pushdown: – Instead of performing the filter after the join, the filter can be applied earlier in order to pass fewer elements into the filter.

Some selection optimizations include:

- Performing filters as early as possible.
- Reordering predicates so that the DBMS applies the most selective one first.
- Breaking up a complex predicate and pushing it down.

Some projection optimizations include:

- Perform projections as early as possible to create smaller tuples and reduce intermediate results *(projection pushdown)*.
- Project out all attributes except the ones requested or requires.

An example of projection pushdown in shown in Figure 3.

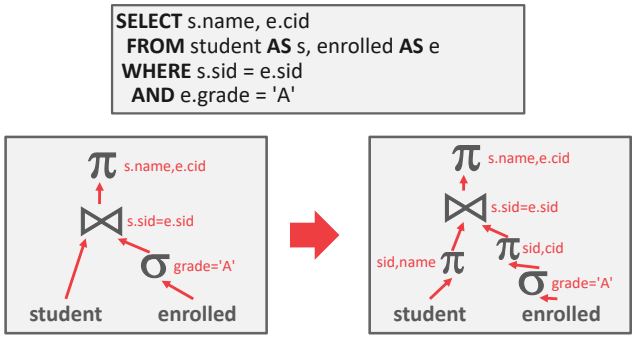

Figure 3: Projection Pushdown – Since the query only asks for the student name and ID, the DBMS can remove all columns except for those two before applying the join.

Another optimization that a DBMS can use is to remove impossible or *unnecessary predicates*. That is, remove the evaluation of predicates those result does not change per tuple in a table. Bypassing these predicates will reduce computation cost. Figure 4 shows two examples of unnecessary predicates.

Figure 4: Unnecessary Predicates – The predicate in the first query will always be false and can be disregarded. The next predicate will always return true and can also be bypassed.

Similarly, *unnecessary joins* can be eliminated as shown in Figure 5.

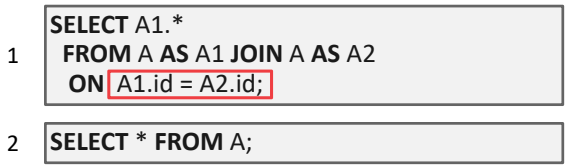

Figure 5: Join Elimination – The join in query 1 is wasteful because every tuple in A must exist in A. Query 1 can instead be written as query 2.

Another optimization is *ignoring projections* that are unnecessary. An example of this is shown in Figure 6.

```
SELECT * FROM A;
```

```
SELECT * FROM A AS A1
 WHERE EXISTS(SELECT val FROM A AS A2
         WHERE A1.id = A2.id);
```

Figure 6: Ignoring Projections – Query 1 contains a wasteful join and projection because it checks that any output found exists. It can be rewritten entirely as query 2.

The final optimization is *merging predicates* shown in Figure 7.

```
SELECT * FROM A
 WHERE val BETWEEN 1 AND 150;
```

```
SELECT * FROM A
 WHERE val BETWEEN 1 AND 100
   OR val BETWEEN 50 AND 150;
```

Figure 7: Merging Predicates – The `WHERE` predicate in query 1 has redundancy as what it is searching for is any value between 1 and 150. Query 2 shows the more succinct way to express request in query 1.