Lecture #13: Optimizer Implementation I

15-721 Advanced Database Systems (Spring 2024)
https://15721.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Prof. Andy Pavlo

1 Introduction

Query optimization is responsible for converting SQL statements to a scheme that can be executed by the database system. It plays a vital role in the entire system, as no matter how good the storage and execution engines are, the performance can still be poor if the optimizer does not generate an efficient execution scheme.

1.1 Logical vs. Physical Plans

Logical plans contain **logical operators**, which roughly correspond to relational algebra operators. They describe what the operator's result looks like without describing how the system obtains that result.

Physical plans contain **physical operators** that specify how a logical operation is implemented. The physical operators typically depend on the physical format of the data they process.

Logical operators do not have a 1-to-1 mapping to physical operators. For instance, a logical join and a logical sort can be combined into a physical sort-merge join.

1.2 Cost Estimation

Cost estimation generates an estimate of the cost of executing a plan based on the current state of the database. Typically, the cost does not directly reflect query runtime - it is only for comparison purposes and is meaningless on its own.

1.3 Query Optimization

For a given query, the **query optimization** process finds a correct physical execution plan for that query *ideally* with the lowest cost. Here, we say *ideally* because:

- Enumerating all possible plans is impractical, so we must trim down the search space.
- The cost model typically underestimates costs by a large margin, so the ”cost-optimal” plan is not necessarily the ”true” optimal plan.

The optimization process usually involves logical-to-logical, logical-to-physical, and physical-to-physical operator transformations. It never transforms physical operators back to logical operators.

1.4 Enumeration Architectures

There are three key components in a query optimizer: **search space**, **cost estimation** (aka. cost model), and **enumeration architecture**. This lecture covers the following four enumeration architectures.

- Heuristics
- Heuristics + Cost-based Search
- Stratified Search
- Unified Search

2 Heuristics

A **heuristics-based optimizer** defines static rules (essentially if-and-else clauses) that transform logical operators into a physical operators *without* doing a cost-based search. Below are some examples of logical transformation rules.

- **Split Conjunctive Predicates**: Decompose predicates into their simplest forms to make them easier to move around.
- **Predicate Pushdown**: Move the predicate to the lowest point in the plan after Cartesian products.
- **Replace Cartesian Products with Joins**: Replace all Cartesian products with inner joins using the join predicates.
- **Projection Pushdown**: Eliminate redundant attributes before pipeline breakers to reduce materialization cost.

Heuristics-based optimizers are easy to implement and efficient. However, they only work well on simple queries and rely on magic constants to evaluate planning decisions.

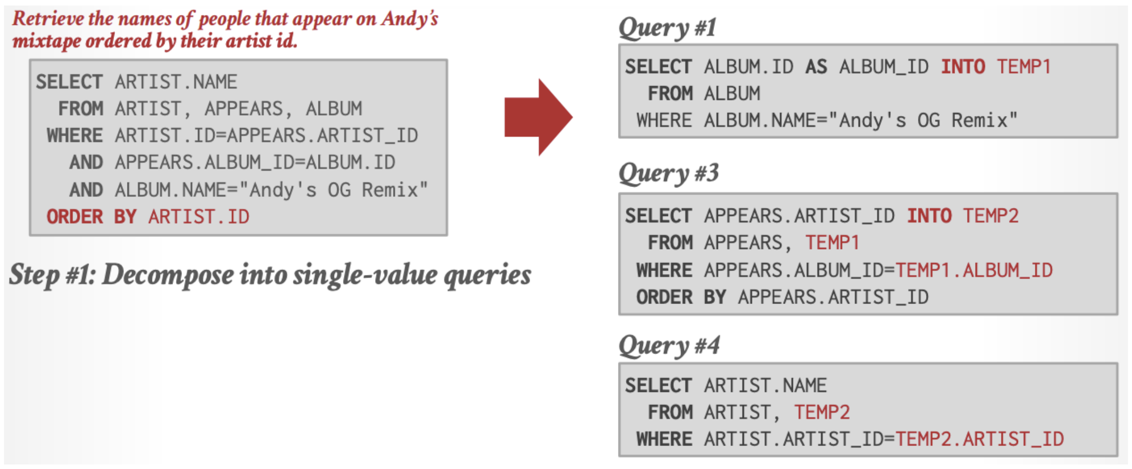
Examples:

- **Ingres** (until mid-1980s)
- **Oracle** (until mid-1990s)
- **MongoDB**
- Most DBMSs in their earliest development stages

2.1 Ingres Optimizer

Initially, Ingres was unable to perform joins. Instead, joins were handled in the following way.

1. Decompose a join query into simpler queries where the FROM clause contains at most two tables joining. Except for the first query (see Query #1 in Figure 1), the other queries contain a join between one of the tables in the original query and the materialized result of the previous query.



Retrieve the names of people that appear on Andy's mixtape ordered by their artist id.

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
 ORDER BY ARTIST.ID
```

Step #1: Decompose into single-value queries

Query #1

```
SELECT ALBUM.ID AS ALBUM_ID INTO TEMP1
  FROM ALBUM
 WHERE ALBUM.NAME="Andy's OG Remix"
```

Query #3

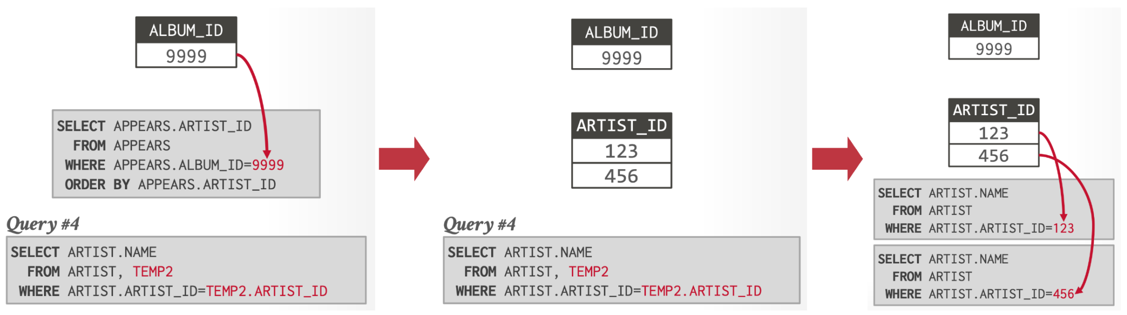
```
SELECT APPEARS.ARTIST_ID INTO TEMP2
  FROM APPEARS, TEMP1
 WHERE APPEARS.ALBUM_ID=TEMP1.ALBUM_ID
 ORDER BY APPEARS.ARTIST_ID
```

Query #4

```
SELECT ARTIST.NAME
  FROM ARTIST, TEMP2
 WHERE ARTIST.ARTIST_ID=TEMP2.ARTIST_ID
```

Figure 1: Join decomposition

2. Loop through the values from the previous query, substituting them into one side of the join predicate and removing the join (see Figure 2). The queries generated in this process involve only a single table and are called **single-value queries**.



```
SELECT APPEARS.ARTIST_ID
  FROM APPEARS
 WHERE APPEARS.ALBUM_ID=9999
 ORDER BY APPEARS.ARTIST_ID
```

```
SELECT ARTIST.NAME
  FROM ARTIST, TEMP2
 WHERE ARTIST.ARTIST_ID=TEMP2.ARTIST_ID
```

```
SELECT ARTIST.NAME
  FROM ARTIST
 WHERE ARTIST.ARTIST_ID=123
```

```
SELECT ARTIST.NAME
  FROM ARTIST
 WHERE ARTIST.ARTIST_ID=456
```

Figure 2: Single-value substitution

3 Heuristics + Cost-Based Search

In a **heuristics + cost-based search optimizer**, optimization is split into two stages. In the first stage, we apply heuristics, as we do above. In the second stage, we enumerate some number of possible query plans and select the best one according to its estimated cost.

Examples:

- **System R**
- **DB2** (in its early days)
- **Postgres**
- Most open-source DBMSs

3.1 System R

The System R optimizer [4] was the optimizer that pioneered this architecture. In the first stage, the system *heuristically* applied logical-to-logical transformations. In the second stage, the system ”searched” through different plans by performing a series of logical-to-physical transformations and estimating the cost of each searched subplan.

System R’s search process was **bottom-up** as the system would first find the best physical plan for subplans involving a single relation, and then the best physical plan for subplans involving two relations, and so on (see Figure 3). To prune the search space, the system only considered **left-deep trees**, or trees where the right subplan of all join nodes does not have any join nodes of its own. Additionally, System R would mark subplans with **physical properties** such as whether the results are sorted or not. Plans using those subplans could take advantage of such physical properties.

4 Aside: Optimizer Generators

The two optimizers discussed so far, Ingres and System R, were both standalone, monolithic programs. However, as the field of query optimization matured, the idea of an **optimizer generator**, or optimizer framework, emerged. With an optimizer generator, the search algorithm of the optimizer is decoupled from the specific transformation rules used in the search process. Furthermore, such transformation rules are generally written in a custom declarative programming language, which makes them easier to program. Additionally, an optimizer written using a framework is more extensible because it is easier to plug in new transformation rules to the optimizer as the DBMS receives new features.

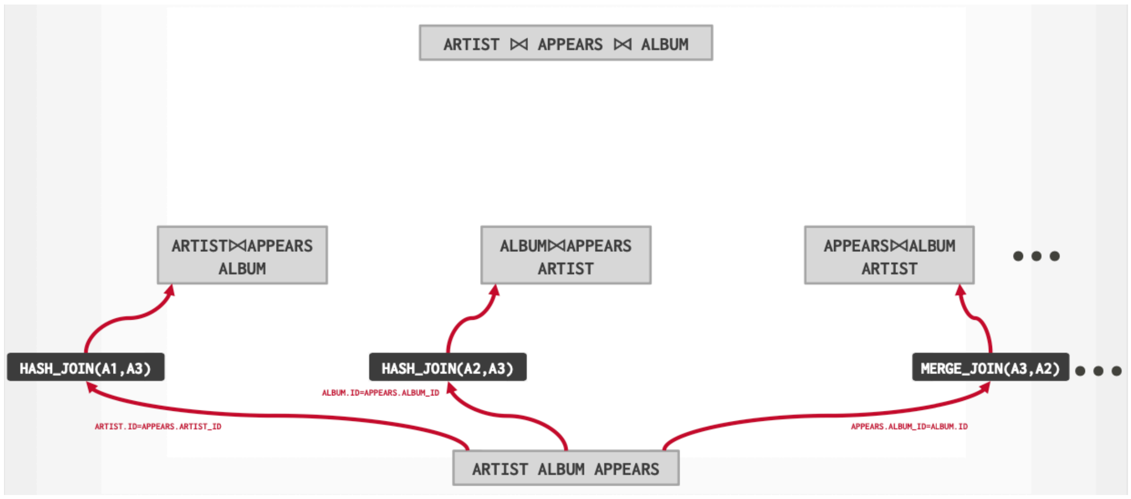


Figure 3: The bottom-up search process

5 Stratified Search

When developing an optimizer generator, one option for the search strategy is **stratified search**. With this, optimization is split into multiple distinct stages that each use a subset of the transformation rules. For instance, it is common to perform logical-to-logical transformations in the first stage and logical-to-physical transformations in the second stage, similar to heuristics + cost-based search.

Examples:

- **Starburst**
- **CockroachDB**

5.1 Starburst

The Starburst optimizer [3] is essentially an implementation of the System R optimizer as an optimizer generator. It has a heuristic phase and a cost-based search phase, just like the System R optimizer. The cost-based search is also performed in the same bottom-up manner.

6 Unified Search

Another search strategy for an optimizer generator is a **unified search**. With this, the entire optimization process happens in a single (search) phase where both logical-to-logical and logical-to-physical transformations are applied. This approach removes the need to apply transformations using only heuristics, instead folding them into the search process.

Examples:

- **Volcano**
- **Cascades**
- **optd**
- **OPT++**
- **SQL Server**

6.1 Volcano

The Volcano optimizer [2] uses unified search with a **top-down** approach (see Figure 4) based on branch-and-bound search. Starting from the root node, the system would invoke transformation rules to expand the plan and traverse the search space. To improve efficiency, the system would **memoize** the best costs of subplans found so far and bound a particular branch if its cost exceeded the memoized cost of equivalent subplans.

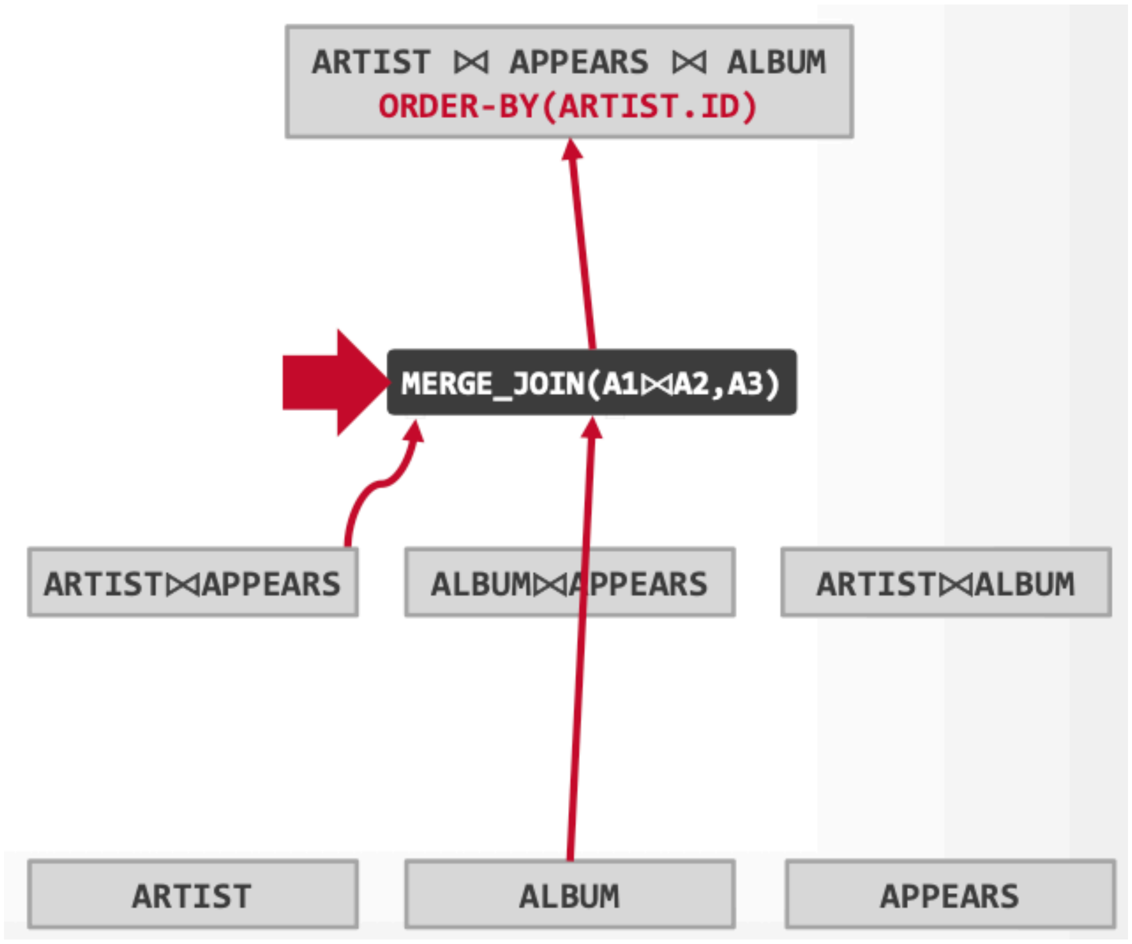


Figure 4: The top-down search process

Volcano also had the notion of physical properties, like System R. However, since its search was top-down instead of bottom-up, these physical properties were managed using **enforcer rules** instead. Enforcer rules constrained child nodes to only use transformations that resulted in the physical properties required by their parents.

6.2 Cascades

The Cascades optimizer [1] also performs top-down search, but in a way that is more sophisticated than the Volcano optimizer. Instead of trying all transformation rules immediately upon arriving at a node, it can prioritize some rules and defer other rules to later. It achieves this by abstracting the children of a node away with placeholders so that they do not need to be materialized immediately. More discussion on the Cascades optimizer can be found in **Lecture #14: Optimizer Implementation II**.

7 Search Termination Strategies

As enumerating all possible plans may take too long, we need a way to determine whether we have reached a good enough plan and stop the search. Below are some common strategies for terminating the search.

- **Wall-clock Time**: Stop after the optimizer runs for some length of time.
- **Cost Threshold**: Stop when the optimizer finds a plan that has a lower cost than some threshold.
- **Exhaustion**: Stop when there are no more enumerations of the target plan. Usually done per sub-plan.
- **Transformation Count**: Stop after a certain number of transformations have been considered.

References

[1] G. Graefe. The cascades framework for query optimization. *IEEE Data(base) Engineering Bulletin*, 18: 19–29, 1995. URL `https://api.semanticscholar.org/CorpusID:260706023`.

[2] G. Graefe and W. McKenna. The volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218, 1993. doi: 10.1109/ICDE.1993.344061.

[3] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, page 18–27, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912683. doi: 10.1145/50202.50204. URL `https://doi.org/10.1145/50202.50204`.

[4] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, page 23–34, New York, NY, USA, 1979. Association for Computing Machinery. ISBN 089791001X. doi: 10.1145/582095.582099. URL `https://doi.org/10.1145/582095.582099`.