# Lecture #14: Optimizer Implementation II

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

## 1 Introduction

In the last class, we learned about different optimizer implementations. We noted that early systems began with Heuristic approaches and then moved into using cost-based join search strategies. Recent systems primarily use two methodologies: stratified and unified search.

1. **Stratified Search** entails rewriting the logical query plan via specific transformation rules. This process involves verifying the suitability of a transformation before its execution, concentrating only on structural changes without accounting for costs. After this initial phase, the approach employs a cost-based search to evolve the logical plan into an efficient physical plan.
2. **Unified Search** combines the procedures of logical-to-logical and logical-to-physical transformations, thus erasing the boundaries between stages. This method leads to the generation of a large number of transformations, with heavy utilization of memoization to minimize redundant computations and enhance efficiency.

These search methodologies can be implemented through two approaches:

1. **Top-down** begins with the end goal of the query and proceeds downwards to identify the optimal plan (e.g. Volcano and Cascades).
2. **Bottom-up** starts from a basic state and methodically builds up towards the desired outcome (e.g. System R and Starburst).

Understanding the distinctions and applications of these search and optimization strategies is essential for developing efficient query execution plans.

## 2 Cascades

Using the Volcano optimizer as a basis, Cascades introduces an object-oriented query optimization framework. While Volcano employs a more traditional bottom-up approach, Cascades opts for a top-down strategy, integrated with a branch-and-bound search, thereby refining the process of exploring possible query plans [3].

Distinct from Volcano, Cascades uses the incremental materialization strategy; this method enables more efficient resource management by progressively constructing potential operator representations within the query plan.

Four core ideas underlie the Cascades framework [13]:

- **Task Structuring**: Optimization is reimagined as matching and rule application tasks, streamlining the enhancement of query plans through a systematic approach.

- **Enforcing Properties**: Cascades ensures plan integrity by only considering sub-plans that meet necessary conditions, like sorted data, hence optimizing output efficiency.
- **Rule Prioritization**: The framework assigns priorities based on the transformation's potential, focusing on high-impact changes to expedite optimal plan formulation.
- **Consistent Application**: By applying rules uniformly across logical and physical spectrums, including WHERE clause conditions, Cascades simplifies and unifies the optimization process.

Incorporating these principles, Cascades systematically explores the query optimization space, halting when reaching a set limit, like maximum transformations, time bounds, or cost thresholds, ensuring plans are both correct and efficiently optimized.

## 2.1 Implementation

In the Cascades framework, an **expression** represents some operation in the query execution plan.

Logical expressions illustrate the structure of a query without specifying how operations are executed, for example, the logical representation of a SQL join operation: $(A \bowtie B) \bowtie C$.

Conversely, physical expressions delineate specific execution strategies, such as $(A_{Seq} \bowtie_{HJ} B_{Seq}) \bowtie_{NL} C_{Idx}$, indicating sequence scans and join methods.

**Groups** are central to Cascades, encompassing sets of logically equivalent expressions that yield identical outputs. These groups merge various logical interpretations and their corresponding physical implementations, ensuring diverse but equivalent solutions are considered under a unified framework.

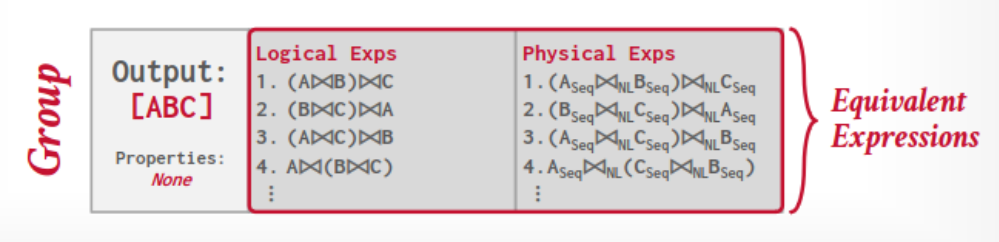


Figure 1: Illustration of a group containing equivalent logical and physical expressions.

To optimize resource usage, Cascades avoids materializing all possible expressions within a group. Instead, it employs **multi-expressions** to represent equivalent expressions implicitly, thereby streamlining transformation processes and reducing both storage and computational burdens.


Figure 2: Concept of multi-expressions in grouping equivalent expressions.

**Rules** in Cascades are pivotal for transforming expressions while preserving their logical equivalence, delineated as follows:

1. **Transformation rules** convert one logical expression into another, facilitating alternative but equivalent representations of the same logical operation.
2. **Implementation rules** transition expressions from the logical realm to the physical, defining how operations are executed in practice.

Each rule is characterized by two key components:

1. A **pattern**, which identifies the applicable structural configurations for the rule.
2. A **substitute**, defining the new structure resulting from the rule's application.

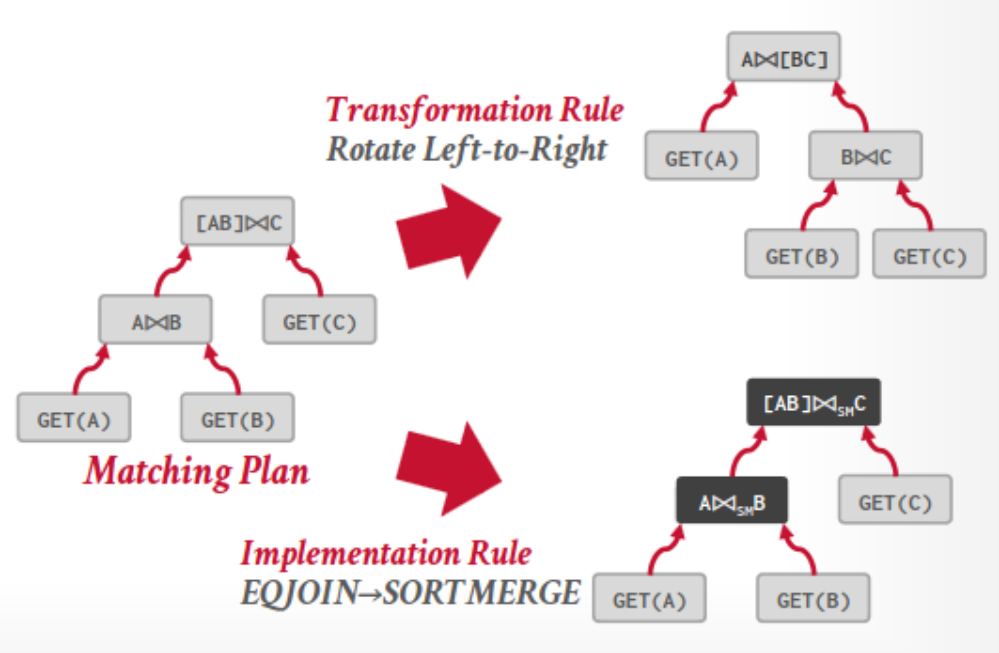


Figure 3: Transformation and implementation rules within the Cascades framework.

The **memo table** is a pivotal component, cataloging all explored alternatives within a compact, graph-like structure for efficient retrieval. This setup aids in memorizing transformation outcomes, detecting duplicate groups, and managing properties and costs efficiently.

Cascades leverages **the principle of optimality** [10], asserting that every sub-plan of an optimal plan must also be optimal. This foundational principle restricts the search space to the most promising expressions, eliminating the need to consider less efficient plans with equivalent physical properties.

## 3 Randomized Search

**Randomized algorithms** perform a random walk over a solution space of all possible (valid) plans for a query. The algorithm continues searching until a cost threshold is reached or the optimizer runs for a length of time. Examples include Postgres' genetic algorithm.

An early paper [4] was able to accomplish this with **Simulated Annealing**. The algorithm can be described as follows: (1) start with a query plan that is generated using the heuristic-only approach (2) compute random permutations of operators (e.g., swap the join order of two tables). In step 2, a permutation is always accepted when there is a change that reduces cost, whereas the optimizer only accepts a change

that increases cost with some probability. It will also reject any change that violates correctness (e.g., sort ordering).

### 3.1 Postgres' Genetic Optimizer

In Postgres' Genetic Optimizer [8], more complicated queries use a genetic algorithm that selects join orderings (GEQO). At the beginning of each round, it generates different variants of the query plan. It then selects the plans that have the lowest cost and permutes them with other plans. The optimizer continues to repeat this process. Notably, the mutator function only generates valid plans.

Since this technique jumps around the search space randomly, the optimizer can avoid local minimums. Also, if no history is kept, there is a low memory overhead. However, when using this strategy, it is difficult to determine why the DBMS may have chosen a plan. Furthermore, the optimizer must do extra work to ensure deterministic query plans.

### 3.2 Dynamic Programming Optimizer

Another approach is to use dynamic programming as a way to enumerate through query plans for the query optimizer by modeling the query as a hypergraph. A brief overview of the algorithm is for the optimizer to iterate through connected sub-graphs and incrementally add new edges to other nodes to complete the query plan and then use rules to determine which nodes the traversal is allowed to visit and expand [5].

## 4 Real-World Implementations

Real-world implementations referenced in class can be noted as follows:

**Standalone**:

- Wisconsin OPT++ (1990s)
- Portland State Columbia (1990s)
- Greenplum Orca (2010s)
- Apache Calcite (2010s)

**Integrated**

- Microsoft SQL Server (1990s)
- Tandem NonStop SQL (1990s)
- CockroachDB (2010s)

We primarily focus on Microsoft SQL Server. SQL server, Orca, CockroachDB, and Snowflake are all implementations of Cascades.

### 4.1 Microsoft SQL Server

Introduced in 1995, the first Cascades framework has become integral to numerous Microsoft database products. Distinctively, its transformations are implemented in C++ without a DSL, favoring procedural over rule-based methods for scalar and expression transformations.

The DBMS employs a structured pipeline of four transformation stages, each escalating in complexity to methodically refine and constrain the search space [7].

1. **Simplification/Normalization**: Tree transformations including subquery removal, outer-to-inner join conversion, predicate pushdown, and pruning of empty results.
2. **Pre-Exploration**: Initializes cost-based search, addressing trivial plans, projection normalization, and initial cardinality estimations.
3. **Exploration**: Advances through a phased cost-based search, from simple to more intricate plans, depending on the remaining optimization time budget.
4. **Post-Optimization**: Implements final engine-specific adjustments.

MSSQL further employs two key strategies for optimized query performance and consistent results. The first is that timeouts are set based on the number of transformations to ensure consistent optimization, regardless of system load. The second is that the system pre-fills the memo table with strategic join orders based on heuristic analysis (e.g. base order from SQL query).

### 4.2 Apache Calcite

Apache Calcite [1] is a standalone extensible query optimization framework for data processing systems. It includes support for pluggable query languages, cost models, and rules. It also does not distinguish between logical and physical operators and provides physical properties as annotations. Calcite was originally developed as a part of LucidDB.

### 4.3 Greenplum Orca

Greenplum Orca [11] is a standalone Cascades implementation in C++. The query optimizer was originally written for Greenplum. It was later extended to support HAWQ. A DBMS can integrate Orca by implementing an API to send the catalog, statistics, and logical plans to the optimizer, in turn receiving physical plans. Orca is also able to support multi-threaded search.

Since when Orca was developed, most DBMSs were not running in the cloud, Orca supports Remote Debugging by automatically dumping the state of the optimizer (with inputs) whenever an error occurs. The dump is enough to put the optimizer back in the same state later for further debugging, which reduces the amount of engineering effort necessary for debugging.

Orca is also able to verify optimizer accuracy. It does this by executing two plans for each query rather than one, then it automatically checks whether the estimate costs of both plans follow their expected order relative to each other post-execution. In other words, if Plan 1 was expected to run faster than Plan 2, Orca verifies this is what happened post-execution while updating the cost-model statistics as necessary.

### 4.4 CockroachDB

Written in 2018, CockroachDB [12] employs yet another custom Cascades implementation. All of its transformation rules are written in a custom DSL (OptGen) and then converted into Go-lang. Additionally, one can embed Go logic within a rule to perform more complex analyses and modifications. Also, CockroachDB considers scalar expression (predicate) transformations together with relational operators.

## 5 Unnesting Subqueries

SQL allows a nested `SELECT` subquery to exist (almost) anywhere in another query. Places where such a subquery can exist include in `SELECT`, `FROM`, `WHERE`, `LIMIT`, and `HAVING` clauses. Once the inner subquery is

completed, the results are passed to the outer query. Such nesting enables more expressive queries without having to use additional queries to prepare intermediate results.

There are two types of subqueries: uncorrelated subqueries and correlated subqueries.

An **uncorrelated subquery** does not reference any attributes from the (calling) outer query. The DBMS logically executes it once and then reuses the result for all tuples in the outer query. An example of such a query is as follows:

```
SELECT name
    FROM students
WHERE score =
    (SELECT MAX(score) FROM students);
```

On the other hand, a **correlated subquery** refers to one or more attributes from outside of the subquery (i.e., the outer query). The DBMS logically evaluates the subquery on each tuple in the outer query since the result can change from one tuple to another. An example of such a query is as follows:

```
SELECT name, major
    FROM students AS s1
WHERE score =
    (SELECT MAX(s2.score)
    FROM students AS s2
    WHERE s2.major = s1.major);
```

## 5.1 Hueristic Rewriting

Almost every DBMS that supports nested subqueries uses heuristics that identify specific query plan patterns to decorrelate nested subqueries. The goal is to move the subquery up a level so that the DBMS can execute it as a join. This is called **hueristic rewriting** [2].

These transformed queries are more efficient since a subquery does not have to run for every outer query. Additionally, deciding to decorrelate can be a cost-based decision, which means if using a nested strategy is more optimal, we don't lose out. It's also easy to control decorrelation by enabling/disabling rules.

However, it's hard to write rules for all possible correlation scenarios, which means we aren't able to decorrelate all nested subqueries. This also means that changing a small part of a query can make these rules ineffective. Furthermore, when using a large amount of heuristic rules, these rules become difficult to maintain, since one has to verify adding an additional rule does not cause conflicts with prior rules. This makes handling all edge cases very difficult.

## 5.2 German-Style Unnesting Subqueries

To flatten correlated joins, the key idea in the paper *Unnesting Arbitrary Subqueries* [6] is introducing the dependent join operator. This new dependent join relational algebra operator denotes a correlated subquery, and its job is to evaluate the right-hand side (RHS) of the join for every tuple on the left-hand side (LHS), combining the results from every execution and returning them as its output. The Germans then came up with a general-purpose method to eliminate all dependent joins by manipulating the query plan until the RHS of a plan no longer depends on the LHS. The optimizer is then able to convert dependent joins to regular joins. In the best-case scenario, queries can switch from a $O(n^2)$ nested-loop join to a $O(n)$ hash

join. This technique doesn't have a name in the paper, but in class, we call this **German-Style Unnesting Subqueries**.

The overall procedure is summarized as follows [9]:

1. Push the dependent join down into the RHS of the plan.
2. Introduce duplicate elimination scan since we only need to execute the RHS once for every unique combination of correlated columns.
3. Push the dependent join as far into the plan as possible.
4. Convert the dependent join operator into a cross-product.
5. Convert the cross-product into an inner join.
6. Remove duplicate elimination scan.
7. Remove the filter above the new join.

## 6 Conclusion

Only HyPer, Umbra, and DuckDB correctly unnest correlated sub-queries. All the optimizer strategies we discussed assume that the optimizer has one shot at choosing a plan. But what happens if the DBMS discovers that the cost estimates don't match reality when it starts processing data?

## References


[1] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 221–230, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3190662. URL https://doi.org/10.1145/3183713.3190662.

[2] C. Duta, D. Hirn, and T. Grust. Compiling pl/sql away, 2019.

[3] G. Graefe. The cascades framework for query optimization, 1995.

[4] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, page 9–22, New York, NY, USA, 1987. Association for Computing Machinery. ISBN 0897912365. doi: 10.1145/38713.38722. URL https://doi.org/10.1145/38713.38722.

[5] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 539–552, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581026. doi: 10.1145/1376616.1376672. URL https://doi.org/10.1145/1376616.1376672.

[6] T. Neumann and A. Kemper. Unnesting arbitrary queries. In *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, pages 383–402. Gesellschaft für Informatik e.V., Bonn, 2015. ISBN 978-3-88579-635-0.

[7] C. G.-L. Nico Bruno, 2020. https://youtu.be/pQe1LQJiXN0.

[8] PostgreSQL Documentation, 1996. https://www.postgresql.org/docs/9.4/geqo-pg-intro.html.

[9] M. Raasveldt, 2020. https://youtu.be/ajpg_pMX620.

[10] L. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H.-M. Wu, and B. Vance. Exploiting upper and lower bounds in top-down query optimization. In *Proceedings 2001 International Database Engineering and Applications Symposium*, pages 20–33, 2001. doi: 10.1109/IDEAS.2001.938068.

[11] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca: a modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 337–348, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323765. doi: 10.1145/2588555.2595637. URL https://doi.org/10.1145/2588555.2595637.

[12] R. Taft, 2020. https://youtu.be/wHo-VtzTHx0.

[13] Y. Xu. Efficiency in the columbia database query optimizer, 1998.