Lecture #05: Cascades

15-799 Special Topics in Databases: Query Optimization (Spring 2025)
`https://15799.courses.cs.cmu.edu/spring2025/`
Carnegie Mellon University
Prepared By: Xueqi Li & Bobby Norwood

1 Introduction

Cascades is an object-oriented, transformation-based (top-down) query optimization framework [3]. It is the third query optimizer project from Graefe (preceding projects were EXODUS [4] and Volcano [5]. One of the first Cascades implementations was the Columbia query optimizer, built by one of Graefe's students as part of their 1998 masters thesis [9].

Like Volcano, Cascades uses a top-down approach (backward chaining) with branch-and-bound search. The primary difference is that Cascades decomposes the search process into *tasks*, which it manages with a stack. Cascades also supports expression rewriting through a direct mapping function (e.g., mapping `WHERE 1=1` to `WHERE true`), reducing the need to perform exhaustive search.

Cascade has four key ideas that distinguish it from Volcano.

1. **Optimization tasks as data structures:** Cascades has dedicated data structures that keep track of patterns to match and transformation rules to apply.
2. **Rules to place property enforcers:** Cascades represents enforcers as rules that change the physical properties of the plan (i.e., no special casing required to implement enforcers).
3. **Ordering of moves by promise:** Cascades dynamically assigns task priorities to find the optimal plan more quickly.
4. **Unified representation of rules and operators:** A single search engine for logical and physical operators enables interleaving logical and physical transformations.

1.1 Expressions and Groups

The most granular unit of a query plan in Cascades is an *expression*. An expression represents some operation in the query with zero or more input expressions. Expressions can logical (e.g., $(A \bowtie B) \bowtie C$), physical (e.g., $(A_{\text{Seq}} \bowtie_{\text{HJ}} B_{\text{Seq}}) \bowtie_{\text{NL}} C_{\text{Idx}}$), or a combination of both. Therefore, the optimizer must be able to quickly determine whether two expressions are equivalent, including logical-logical, physical-physical, and logical-physical expressions. Expressions can specify output properties (e.g., sortedness).

A *group* is a set of logically equivalent logical and physical expressions that produce the same result. This set includes all logical forms of an expression and all physical expressions derived from selecting allowable physical operators for the corresponding logical forms. For example, $A \bowtie B \bowtie C$ can be represented as $\{ABC\}$, as shown in the group below:

Logical Expressions	Physical Expressions
$\{AB\} \bowtie \{C\}$	$\{AB\} \bowtie_{\text{SM}} \{C\}$
$\{BC\} \bowtie \{A\}$	$\{AB\} \bowtie_{\text{HJ}} \{C\}$
$\{AC\} \bowtie \{B\}$	$\{AB\} \bowtie_{\text{NL}} \{C\}$
$\{A\} \bowtie \{BC\}$	$\{BC\} \bowtie_{\text{SM}} \{A\}$

Output: $\{ABC\}$, Properties: None

The optimizer implicitly represents redundant expressions in a group with a placeholder (e.g., $\{ABC\}$). This representation allows the optimizer to avoid explicitly instantiating all the possible expressions in a group. In the example above, the result $\{AB\}$ appears many times in the group's expressions. Instead of repeatedly computing all of $\{AB\}$'s expressions, the optimizer saves and refers to $\{AB\}$ as another group. This technique reduces the number of transformations, storage overhead, and cost estimations.

1.2 Rules

Like the previous systems that we discussed, a rule is a transformation of an expression into a logically equivalent expression. There are two types of rules:

- **Transformation Rule:** logical to logical.
- **Implementation Rule:** logical to physical.

Each rule is represented as a pair of attributes, a *pattern* and a *substitute*. When a logical expression's structure matches a rule's pattern, the optimizer uses the rule's substitute to produce a new expression.

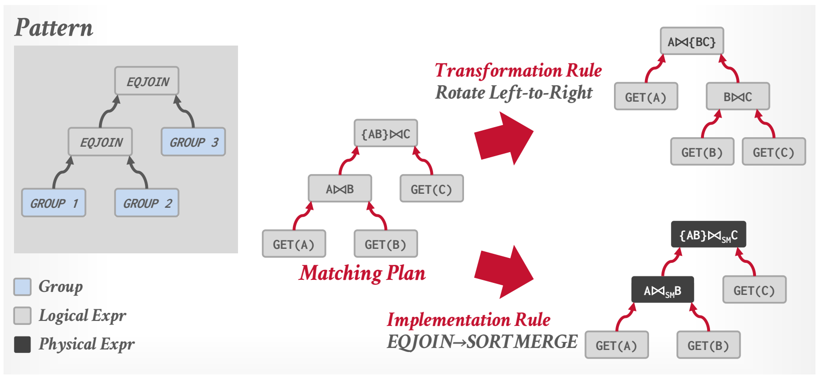


Figure 1: A query plan that matches two different rules.

Figure 1 shows a query plan that matches the pattern of two different rules. The transformation rule rotates the plan left-to-right, whereas the implementation rule realizes the matching equijoins as sort-merge joins.

Transformation rules can only be applied if the output has the same properties as the input (e.g., sort order). The optimizer also needs to maintain metadata that tracks whether it has applied a given rule to avoid reapplying the same rule unnecessarily (e.g., commutativity).

Unlike Volcano, Cascades does not require special casing to implement enforcers. Recall that enforcers are used to guarantee certain plan properties (e.g., sort order). Cascades represents enforcers as rules that insert physical operators to change the physical properties of a plan.

1.3 Tasks

Cascades breaks down optimization into smaller and more manageable pieces (*tasks*) to allow for more flexible and efficient exploration of the search space. A task is a fine-grained unit of work that represents an operation in the query optimization process. This decomposition allows the optimizer to learn the best order for invoking its tasks, based on *promise*. It also allows for parallel task execution (e.g., Orca does this).

1.4 Promises

A task's promise is the estimated benefit of a move on a given expression relative to other tasks. For example, when comparing two possible join orderings, the optimizer estimates the cardinality of both tables and assigns higher promise to a join order swap rule if the outer table is larger than the inner table. Note that while cost may be a good promise metric if it is available, promise is distinct from cost. Additionally, the optimizer must still ensure that tasks execute in the order defined by their dependencies.

2 Cascades: Task Search

2.1 Task Flow

Cascades has six tasks, roughly grouped around Optimize, Explore, and Rewrite.

1. **Optimize Group**: The task is the entry point to any group. This task finds the best physical plan for a given group (set of expressions).
2. **Optimize Expression**: This task finds the best physical plan for a given expression. For both Optimize Group and Optimize Expression, the set of equivalent expressions must first be generated.
3. **Explore Group**: This task generates the logical expressions for a given group. Exploring a group essentially creates an Explore Expression task for each expression in the group.
4. **Explore Expression**: This task applies rules to create all equivalent expressions to the input expression. The application of each rule is itself another task.
5. **Apply Rule**: This task applies a rule to a specific expression and creates additional tasks if necessary.
6. **Optimize Inputs**: This task takes an expression and creates a new Optimize Group task for each of its children.

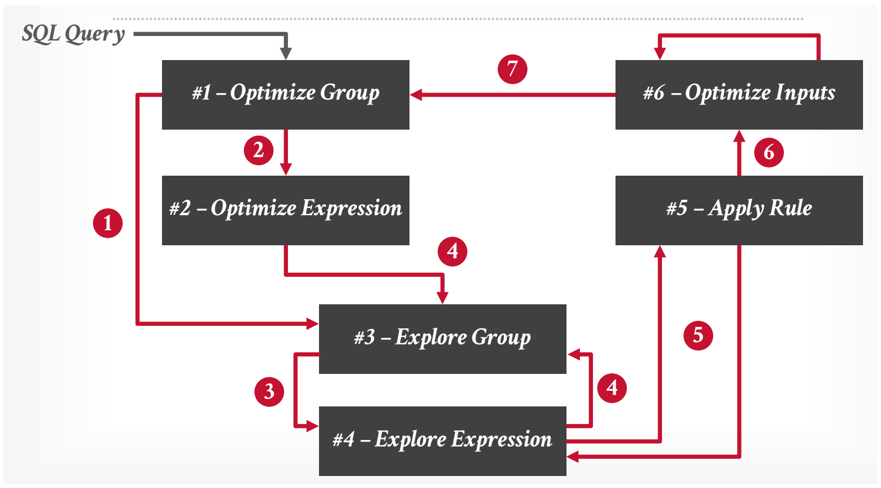


Figure 2: An example of the task flow in Cascades.

We walk through the task search process in Figure 2.

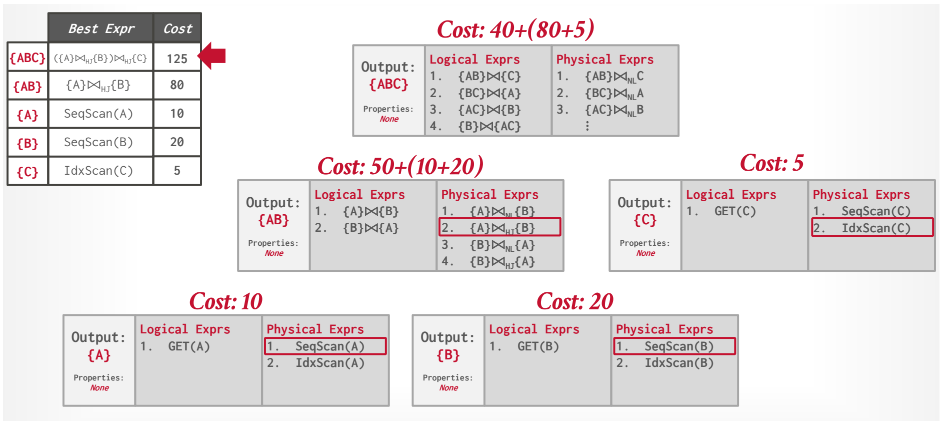
1. The SQL query creates an Optimize Group task. If its corresponding group has not been explored yet, we defer the Optimize Group task and start an Explore Group task, which generates all equivalent logical expressions. This allows the Optimize Group task to proceed.
2. The Optimize Group task creates an Optimize Expression task for each expression in the group. The Optimize Expression task tries to optimize expressions (e.g., it may be able to immediately produce constants). It may also create expressions that need to be explored with Explore Expression tasks.
3. Explore Group expands the group with new expressions, creating Explore Expression tasks.
4. The exploration process feeds into itself. As groups are explored, they may produce new logical expressions. These logical expressions may be converted into physical expressions that then generate new groups and logical expressions.
5. Apply Rule tasks perform logical-to-logical and logical-to-physical expression conversions.
6. The Optimize Inputs task descends further into the query plan by examining the current group's inputs.
7. Each input to the current group generates a new Optimize Group task, restarting the process.

Cascades uses a LIFO stack of tasks to perform these actions on groups and expressions according to the flow shown in Figure 2. The stack maintains the required ordering of tasks while ensuring that expressions are derived after the best plans of their input expressions are derived and allowing independent task optimizations to occur. To reduce out of memory errors, the actual tasks are stored on the heap.

2.2 Memo Table

To maximize efficiency and avoid redundant computation, the optimizer stores all previously explored alternatives in a compact graph structure or hash table known as the *memo table*. Equivalent operator trees and their corresponding plans are stored together in groups. The memo table provides a way to inspect the optimizer's search progress (e.g., for duplicate group detection).

Cascade's search relies on the *Principle of Optimality* that assumes that every sub-plan of an optimal plan is itself optimal. This allows the optimizer to restrict the search space to a smaller set of expressions. For example, if the optimizer knows the optimal sub-plan for $\{A\}$, it does not need to search $\{A\}$ when optimizing $\{A\} \bowtie \{B\}$: it can use the optimal plan that was already found.



	Best Expr	Cost
{ABC}	$(\{A\}\bowtie_{HJ}\{B\})\bowtie_{HJ}\{C\}$	125
{AB}	$\{A\}\bowtie_{HJ}\{B\}$	80
{A}	SeqScan(A)	10
{B}	SeqScan(B)	20
{C}	IdxScan(C)	5

Figure 3: An example of a memo table.

Figure 3 contains an example of a memo table in the upper left, where each group is mapped to its optimal expression and cost. For example, the optimal expression of $\{C\}$ is IdxScan(C), which will be reused for any expression involving $\{C\}$.

3 Optimizations

The Promise mechanism allows Cascades to potentially apply beneficial transformations earlier in the search process. The flexible architecture of Cascades also enables other optimizations to reduce search times.

3.1 Simplification Rules

Because some rules simplify the logical plan and almost always reduce the cost, Cascades expresses them as *simplification rules*: instead of creating alternative expressions, such rules replace the expression with the transformed expression, eliminating the need to retain unnecessary state. These rules are equivalent to Starburst's rewrite rules [6].

3.2 Macro Rules

Macro rules apply multiple transformations in a single rule to reduce the complexity of the search space. Although this goes against the spirit of Volcano's simple and independent rules, Microsoft shows that this technique works well in practice [2].

3.3 Parallel Search

If tasks are independent, the optimizer can execute them in parallel on multiple threads. However, this requires sharing the memo table and ensuring that internal data structures are thread-safe. Orca [7] may be the only multi-threaded Cascades optimizer implementation.

4 Cascades Implementations

There are many implementations of the Cascades optimizer:

Standalone:

- Wisconsin OPT++ (1990s)
- Portland State Columbia (1990s)
- Greenplum Orca (2010s)
- CMU optd (2025)

Integrated

- Microsoft SQL Server (1990s)
- Tandem NonStop SQL (1990s)
- Clustrix (2000s)
- CockroachDB (2010s)
- Snowflake (2010s)
- Databricks (2010s)

4.1 Microsoft SQL Server

Microsoft SQL Server started its Cascades implementation in 1995. Derivatives of its optimizer are used in many Microsoft database products[1]. The transformations are written in C++ (i.e., there is no domain-specific language). It expresses scalar and expression transformations as procedural code instead of declarative rules.

Although their optimizer is based on Cascades, the DBMS applies transformations in multiple stages (which is more similar to Starburst) with increasing scope based on the complexity of the query. This approach leverages domain knowledge to reduce the search space efficiently.

4.1.1 Multi-Stage Optimization in Microsoft SQL Server

Their multi-stage optimization [1] consists of the following steps:

1. **Simplification / Normalization**: This step conducts tree-to-tree transformations that apply basic rules like sub-query removal, outer joins to inner joins, predicate push-down, and empty result pruning. The rules in this category should have broad applicability (always apply).
2. **Pre-Exploration**: This step initializes the cost-based search. If required statistics are not in the system catalog, the system pauses the query optimizer to gather statistics[2]. Examples of rules at this stage include trivial plan short-circuit, projection normalization, statistics identification/collection, initial cardinality estimates, and join collapsing.
3. **Exploration**: This step conducts multi-stage cost-based search. Stage 1 generates the trivial plan. If there is still optimization time available, the optimizer goes into stage 2 for quick plan, and lastly to stage 3 for full plan. The quick plan and full plan may execute in parallel.

[1]Initially, each product maintained its own fork of the optimizer. Later in the semester, we will read about Microsoft Fabric.
[2]In data warehouses (e.g., data exists as Parquet files on S3), obtaining statistics is prohibitively expensive.

4. **Post-Optimization**: This step performs engine-specific transformations. For example, if the query is running on the distributed version of SQL Server, the optimizer converts the query plan from the last stage to a distributed query plan.

4.1.2 Additional Microsoft SQL Server Optimizations

Microsoft SQL Server has additional optimizations that help to generate a better query plan. One optimization is setting timeouts based on the number of transformations instead of wall-clock time. This criteria ensures that systems do not generate different query plans when under heavy load. Another optimization is pre-populating the memo table with potentially useful join orderings based on heuristics that consider relationships between tables and syntactic appearance in the query.

4.2 Greenplum Orca

Greenplum Orca [7] is a standalone Cascades implementation in C++. Orca may be the only Cascades implementation that currently supports multi-threaded search. A DBMS integrates Orca by implementing an API to send its catalog, statistics, and logical plans, after which it can retrieve physical plans. Orca was open-sourced in the 2010s but became closed-sourced in 2024 after Broadcom's acquisition.

4.3 CockroachDB

CockroachDB has a custom Cascades implementation [8] written in 2018 where all transformation rules are written in a custom DSL (OptGen). These rules then undergo codegen into Golang. The rules support embedding Go logic to perform more complex analysis and modifications.

References

[1] N. Bruno and C. Galindo-Legaria. The Cascades Framework for Query Optimization at Microsoft (Nico Bruno + Cesar Galindo-Legaria). https://youtu.be/pQe1LQJiXN0, 2020. [Accessed 01-02-2025].

[2] B. Ding, V. Narasayya, and S. Chaudhuri. Extensible Query Optimizers in Practice. *Foundations and Trends® in Databases*, 14(3-4):186–402, 2024. ISSN 1931-7883. doi: 10.1561/1900000077. URL http://dx.doi.org/10.1561/1900000077.

[3] G. Graefe. The cascades framework for query optimization. *IEEE Data(base) Engineering Bulletin*, 18: 19–29, 1995. URL https://15721.courses.cs.cmu.edu/spring2020/papers/20-optimizer1/graefe-ieee1995.pdf.

[4] G. Graefe and D. J. DeWitt. The exodus optimizer generator. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, page 160–172, 1987. ISBN 0897912365. doi: 10.1145/38713.38734. URL https://doi.org/10.1145/38713.38734.

[5] 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.

[6] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. *SIGMOD Rec.*, 21(2):39–48, June 1992. ISSN 0163-5808. doi: 10.1145/141484.130294. URL https://doi.org/10.1145/141484.130294.

[7] 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.

[8] R. Taft. CockroachDB's Query Optimizer (Rebecca Taft, Cockroach Labs). https://www.youtube.com/watch?v=wHo-VtzTHx0, 2020. [Accessed 01-02-2025].

[9] Y. Xu. Efficiency in the Columbia Database Query Optimizer. 1998. URL https://15721.courses.cs.cmu.edu/spring2020/papers/20-optimizer2/xu-columbia-thesis1998.pdf.