Lecture #04: Volcano

15-799 Special Topics in Databases: Query Optimization (Spring 2025)

https://15799.courses.cs.cmu.edu/spring2025/

Carnegie Mellon University

Prepared By: Melody Hu

1 Introduction

In the late 1900s, *optimizer generators* emerged as a framework for abstracting away the functionality of an optimizer. The framework provides an API that allows database management system (DBMS) developers to avoid the complexity of implementing query optimizers from scratch. Instead, DBMS developers write the rules for optimizing queries, which the optimizer generator then translates into optimizer source code that is compiled and linked with the rest of the DBMS. By doing so, these optimizer generators separate (1) the search strategy from the data model and (2) the transformation rules and logical operators from physical rules and physical operators, ensuring modularity and flexibility. This design also decouples the optimizer's pattern matching mechanism and transformation rules from its search strategy, allowing the search strategy to be changed without modifying the rest of the optimizer.

One of the first optimizer generators was developed in 1987 as part of the EXODUS extensible DBMS project by Goetz Graefe, David DeWitt, and their team at the University of Wisconsin-Madison. Graefe would further improve on the extensibility, efficiency, and expressiveness of the EXODUS Optimizer Generator by creating the Volcano Optimizer Generator in 1994.

2 EXODUS

The EXODUS optimizer generator takes a rule-based approach that separates concerns between the optimization logic and data structures. It translates its algebraic transformation rules into executable optimizer code that performs a bottom-up breadth-first search (*forward chaining*): starting from the query plan roots, it triggers all rules that match those operators and adds their conclusion to the known facts, repeating this process until the full query is produced. The generated optimizer avoids exhaustive search by using dynamic programming and branch-and-bound pruning. It also modifies the search priorities of its rules based on past experience. The initial implementation of EXODUS's optimizer generator used Prolog for its pattern matching and search capabilities, but because their Prolog interpreter was too slow and did not support configuring its depth-first search at runtime, this was replaced with a C implementation.

2.1 Query Optimizer Pipeline

As Figure 1 shows, the optimizer generator is a stand-alone library that can generate query optimizer source code for different DBMSs. The model description file is metadata that includes operator definitions, access method definitions, rules for transforming query plans, and rules that tracked the correspondence between operators and methods. The users of the optimizer generator could also provide their own operator implementations (e.g., to support new operators).

EXODUS has *transformation rules* (logical to logical) and *implementation rules* (logical to physical). Transformation rules are algebraic rules of expression equivalence (e.g., commutativity, associativity) and include hints to prevent reapplying the same rule to the output to avoid infinite loops. The implementation rules map one or more logical operators to one or more physical operators, and vice versa (bidirectional).

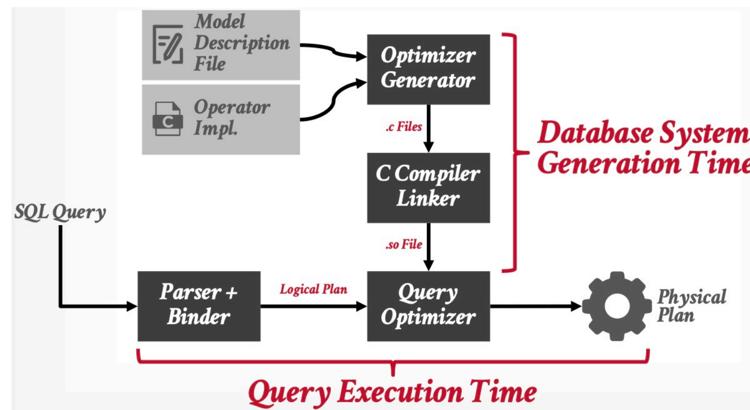

Figure 1: EXODUS's Query Optimizer Pipeline

2.2 Search Algorithm

EXODUS's search algorithm maintains a priority queue of transformations (**OPEN**) to apply on the remaining access plans for a query. These access plans are stored in an in-memory hash table (**MESH**). At the beginning of each round, the optimizer chooses the transformation rule that it estimates would provide the largest cost improvement (i.e., the most *promise*).

Each transformation rule is associated with an expected cost factor f. The promise of a rule is calculated using the pre-transformation cost and the expected cost factor. If C is the cost of the query plan before a transformation with an expected cost factor of f, the query plan cost after applying the transformation is $C \times f$. Expected cost factors are learned from past experiences through averaging historical data, indirect adjustment, and propagation adjustment. The paper shows that various averaging methods are all statistically valid constructs with minor performance differences (e.g., geometric sliding average, geometric mean, arithmetic sliding average, arithmetic mean). Adjustments occur after an advantageous transformation is observed. Indirect adjustment lowers the expected cost factor of the last two rules applied to account for rules that enable subsequent beneficial transformations. Propagation adjustment reduces the expected cost factor if cost advantages are realized when reanalyzing the parent nodes after a transformation. Intuitively, rules that are good heuristics should have $f < 1$, whereas rules that are neutral on average will have $f = 1$.

As stated above, the round begins with the optimizer picking the transformation rule with the most promise from OPEN. Next, it applies the transformation rule to the correct node(s) in MESH. It then immediately applies the implementation rules to convert logical operators into physical operators and eagerly performs cost analysis for new nodes. It then adds newly enabled transformations to OPEN before moving on to the next round. The process stops once OPEN is empty.

2.3 Flaws

Some flaws of the EXODUS system include:

- In MESH, the lookup key is a combination of logical operators and the physical operators that they got converted to. If a logical operator can be converted to different physical operators, each combination would be a separate entry in MESH (i.e., logical operators are duplicated).
- Enforcing certain properties required embedding logic into the cost functions (i.e., adding operators may require modifying the cost model).
- EXODUS immediately applies implementation rules and performs cost analysis after invoking a logical transformation rule. It may be more efficient to defer this step until more logical transformations have been explored.

3 Volcano

Volcano is a general purpose cost-based query optimizer that is based on algebraic equivalence rules. It treats physical properties of data as first-class entities during planning and allows easy additions of new operations and equivalence rules. It has a similar rule compilation pipeline to EXODUS. However, it uses a top-down depth-first search (*backward chaining*): it starts from the query result and works backward to determine what operators to add to the query plan. It also uses branch-and-bound pruning to improve the efficiency of its search.

3.1 Design Goals

In an effort to ensure extensibility, the design goals of Volcano [1] are:

1. Interoperable with existing DBMSs
2. Computational and storage efficiency
3. Extensible physical properties
4. Extensible search guidance and pruning
5. Flexible cost model that supports parameterized queries

3.2 Components

Volcano uses two distinct algebras (logical and physical) to map logical expressions to physical expressions. *Logical expressions* represent user queries as a directed tree of one or more logical operators (e.g., select, project, join). *Physical expressions* represent query evaluation plans using physical algorithms (e.g., sort-merge join, hash join, nested loop join).

Volcano has three operator types: logical, physical, and enforcer. Logical operators have properties (e.g., schema, cardinality) and a property function that is used to query for available properties. Physical operators also have properties (e.g., sort-order, partitioning) and a property function, however, they additionally have cost functions and applicability functions (i.e., whether the operator satisfies the required properties of its logical operator(s)). Enforcer operators are "virtual" physical operators injected into a plan to ensure that physical properties are satisfied (e.g., sort order).

Rules in Volcano are pattern matching functions with an action function to permute a query plan. These patterns support parameterized conditionals based on operator types (e.g., match any scan). All the rules are independent; the system relies on the search engine to find useful combinations of them. The rules also support auxiliary functions that perform additional analysis after a rule's pattern matches (e.g., to examine the structure of the query plan). Volcano prioritizes faster search through rule compilation over the runtime flexibility of rule interpretation (e.g., Starburst).

In Volcano, cost is represented as an abstract data type (ADT) that supports basic arithmetic and comparison functions. DBMS implementers must provide the cost function calculation for each physical operator and enforcer; logical operators have no cost. Volcano does not use the expected cost factor or cost adjustments from EXODUS.

3.3 Search Engine

Volcano's search engine uses a top-down, goal-oriented control strategy with dynamic programming, known as *directed dynamic programming* [1]. This approach starts with the final result of the query and works backward to determine the optimal operators required to achieve that result (using depth-first search).

Similar to EXODUS, Volcano maintains a hash table ("lookup table") with expressions and equivalence classes. It always checks whether a logical or physical operator already exists in the lookup table to avoid reapplying the same transformations. Unlike EXODUS, the lookup table also allows the optimizer to reuse cost calculations without re-analyzing subtrees.

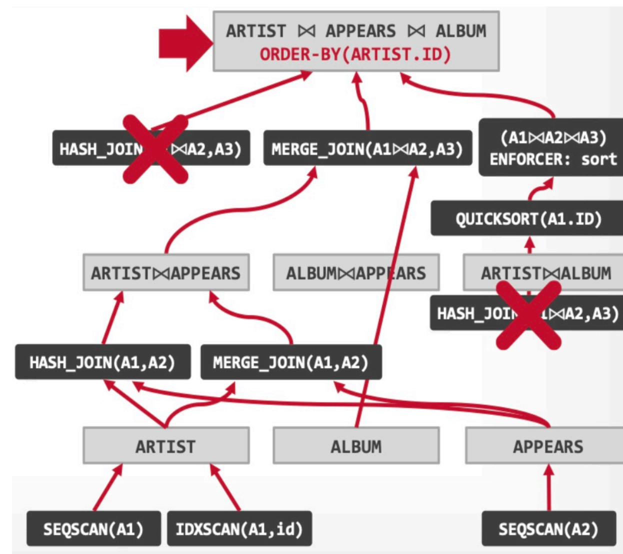

Figure 2: Volcano's Search Process

During the search process, the search engine starts with a logical plan of the final query result. It then invokes rules to create new nodes and traverses the tree to find the plan with the lowest cost, pruning any nodes that exceed the cost limit or violate certain properties (e.g. sort order) to reduce the search space. In particular, the search process is split into two distinct stages:

1. **Generation Phase**: apply transformation rules to generate all possible logical expression alternatives
2. **Cost Analysis Phase**: apply implementation rules to generate physical operators

An example of the search process is shown in Figure 2.

3.4 Advantages & Disadvantages

Some advantages of the Volcano system include:

- It compiles declarative rules to generate transformations.
- It offers better extensibility with an efficient search engine and reduces redundant calculations with the use of a hash table that doesn't include duplicate operators.

Some limitations of Volcano that eventually led to the development of the Cascades optimizer (1995) include:

- All equivalence classes are completely expanded to generate all possible logical operators before the optimization search down the search tree, which increases the search space and leads to unnecessary computations.
- It is not easy to modify compiled transformation rules.

References

[1] 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.