Lecture #06: Transformations

15-799 Special Topics in Databases: Query Optimization (Spring 2025)
https://15799.courses.cs.cmu.edu/spring2025/
Carnegie Mellon University
Prepared By: Mihir Khare

1 Introduction

Previously, we discussed different high-level architectures for query optimizers (e.g., System R, Starburst, Volcano, Cascades). The key factors that determine the quality of the plans that an optimizer generates include its search algorithm, cost model, and transformations. These notes focus on transformations.

In the context of query optimization, transformations change one query plan into a new form that is semantically equivalent to the original. The main restriction on transformations is that they must be logically equivalent (i.e., the new query plan must produce the same result as the original). Transformations exploit relational algebra equivalences in the context of a query, combining that with logical and physical knowledge of the database contents.

The overarching goal of transformations is to lower query execution cost. However, individual transformations do not always have to reduce costs. Many optimizers will make transformations that increase costs in isolation because doing so will unlock additional helpful transformations. For example, converting outer joins into inner joins enables the exploration of additional join orders that can be more efficient.

In this lecture, we cover the following categories of query plan transformations:

- **Access Path**: Determining how to access tuples stored in base relations.
- **Inner Joins**: Determining the best join order.
- **Outer Joins**: Converting certain outer joins to inner joins for easier optimization.
- **Group-By**: Determining whether to push down group-by (i.e., aggregation) operations.

We will also discuss how optimizers make schema-specific optimizations (e.g., star or snowflake schemas)

2 Access Path

Access methods (e.g., indexes, heaps) provide a mechanism for the database engine to access the data stored in base relations [2]. An optimizer chooses access method(s) that minimize the cost of retrieving a query's data from its base relations. The cost of an access method depends on factors such as the selectivity of any predicates, the sort order of the table/index, storage format, and additional information like `INCLUDE` columns or zone maps.

INCLUDE columns: INCLUDE columns are used when creating an index to store the values of additional columns within the leaf nodes. This increases the likelihood of *index-only queries* that can answer a query completely using only the data stored in the index, skipping the need for additional table lookups. Figure 1 shows an index with an INCLUDE column and a query that benefits. However, one limitation of INCLUDE columns is that they cannot be used to guarantee uniqueness.

```
CREATE INDEX idx_foo
          ON foo (a, b)
     INCLUDE (c);

SELECT b FROM foo
 WHERE a = 123
   AND c = 'WuTang';
```

Figure 1: An index with an INCLUDE column and an example query that benefits.

The four main base access methods are as follows:

- **Index Seek**: Retrieving tuples that satisfy a predicate in an index.
- **Index Scan**: Range scan on an index.
- **Index Key Lookup**: Check if a key is present in an index.
- **Table Scan**: Sequential scan through the table heap.

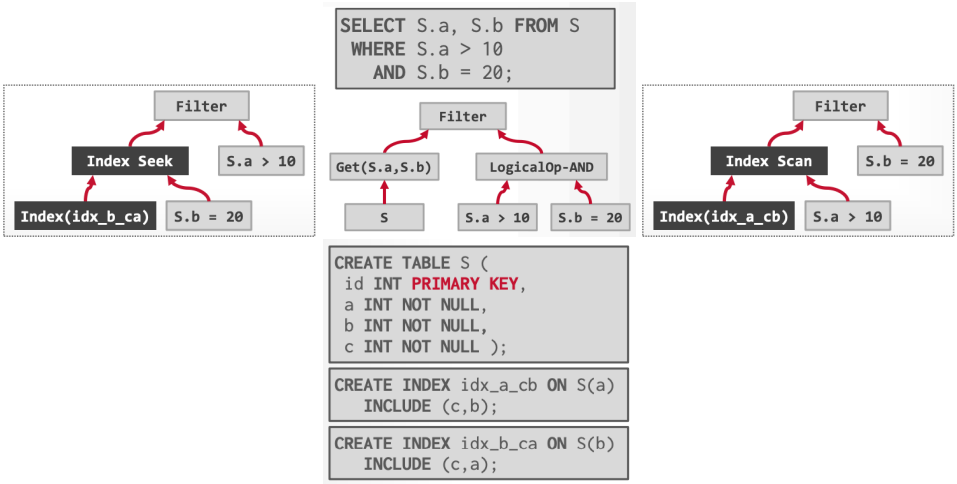

Figure 2: A query plan with two possible access paths for this schema.

2.1 Single Access Method

When choosing access paths, the simplest option is to pick a single access method for each base relation in a given expression. We can do this by generating multiple alternatives and picking the one that we predict will do the least amount of work. Available alternatives depend on the query, database logical schema, and DBMS implementation. For example, if a query's predicates reference data that is not covered by a *partial index*, then the index does not need to be explored. Table scans are always the fallback access method: while they are often the worst choice in a row store, they are sometimes the only choice in a column store.

Partial Indexes: Partial indexes only contain the subset of a table's data that meets some predicate. For example, a partial index on `R` may only include tuples that satisfy `R.a LIKE '%.txt'`.

Figure 2 shows an example of a query on a table with two indexes that can satisfy one of the two predicates. Note that the INCLUDE columns on each index provide the necessary data for the other predicate within the index itself. Without these INCLUDE columns, the DBMS would need to retrieve the corresponding tuples from the table heap to perform the filter, which can be expensive.

2.2 Multiple Access Methods

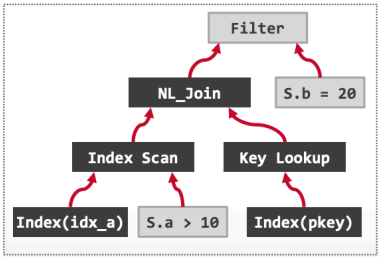

Figure 3: A query plan with multiple access methods on a relation.

In general, the optimizer can pick more than one access method for a query. For example, Figure 3 shows a plan that combines an index scan with a primary index key lookup to obtain matching tuples without performing a table scan. The optimizer can represent multiple access methods on the same base relation as a logical self-join on the target relation. It can then convert the self-join into a regular join or use a specialized multi-index scan physical operator (e.g., PostgreSQL's Bitmap Scan).

The benefit of using multiple access methods depends on factors such as the query (e.g., selectivity of predicates) and physical data layout (e.g., Figure 3 assumes a clustered index or an index-organized table).

3 Inner Joins

Inner join transformations generate different join orderings for the query to determine the cheapest join order. Although inner equi-joins are the most common, the optimizer must still handle cases such as anti-joins.

3.1 Join Enumeration

Exhaustive join enumeration has a space complexity of $O(3^n)$ and a computational complexity of $O(4^n)$. Some of the generated expressions are duplicates (e.g., naively applying commutativity and associativity rules), which Cascades avoids exploring through the use of its memo table.

A query optimizer can leverage behavior about the behavior of rules to avoid unnecessary transformations: by applying rules in a specific order and maintaining a summary about the derivation history of each operator (e.g., with a bitmap [4]), the optimizer can determine which rules to disable to reduce the search space. Figure 4 shows an example of transformation rules with the rules that they disable.

R1: Commutativity
→ $X \bowtie_0 Y \blacktriangleright Y \bowtie_1 X$
→ Disable R1, R2, R3, R4 on new operator $\bowtie_1$.

R2: Right Associativity
→ $(X \bowtie_0 Y) \bowtie_1 Z \blacktriangleright X \bowtie_2 (Y \bowtie_3 Z)$
→ Disable R2, R3, R4 on new operator $\bowtie_2$.

R3: Left Associativity
→ $X \bowtie_0 (Y \bowtie_1 Z) \blacktriangleright (X \bowtie_2 Y) \bowtie_3 Z)$
→ Disable rules R2, R3, R4 on new operator $\bowtie_3$.

R4: Exchange
→ $(W \bowtie_0 X) \bowtie_1 (Y \bowtie_2 Z) \blacktriangleright (W \bowtie_3 Y) \bowtie_4 (X \bowtie_5 Z)$
→ Disable all rules R1, R2, R3, R4 on $\bowtie_4$.

Figure 4: Join enumeration rules.

3.2 Predicate Pushdown / Pullup

Another important aspect of query optimization is the placement of predicates. The optimizer can move any predicate that does not reference attributes in a join result to occur before or after the join, based on the predicate's selectivity and computational cost.

For example, if a predicate is cheap to evaluate and highly selective, it is beneficial to evaluate it before the join. By reducing the size of the join's inputs, we reduce the work that the join performs. However, if a predicate is expensive to evaluate and there is a join with a highly selective predicate of its own, it may be better to pull the former predicate above the join.

3.3 Physical Operators

The selection of physical join operators depends on the join predicates and the data layout requirements for the inputs and outputs. For example, hash joins can only be used for equi-joins, merge join require input data to be sorted on join keys and produces a sorted output, and nested loop joins are a fallback option.

The optimizer can also select runtime operator parameters as part of this process. For example, a hash join requires picking a hash table size, which can be done by using the query optimizer's estimates.

4 Outer Joins

Like inner joins, different orderings for outer joins can have wildly different costs. But unlike inner joins, different orderings of outer joins can retain different non-matching tuples (i.e., produce different results). Our goal is to transform outer joins while preserving the same non-matching tuples and NULL values for attributes.

4.1 Redundancy Rule

The *redundancy rule* [3] replaces certain outer joins with inner joins. The key insight is that if the query contains a NULL-rejecting predicate, then the outer join can be converted into an inner join that produces the same result. For example, suppose we have `S LEFT OUTER JOIN R` and a later predicate filters on `R.a > 10`. Then the NULL values from `R` would not match, allowing the optimizer to convert the outer join into an inner join that is easier to optimize.

5 Group-By

When a query computes an aggregation on the result of joining two or more relations, it may better to first perform the aggregation before the join, and then join the tables on the aggregation result. There are two cases [1]: Complete Group-By Pushdown and Partial Group-By Pushdown.

Complete Group-By Pushdown: In some cases, we are able to move the entire aggregation operator below the join. Some of these cases include (1) all aggregate functions in a group-by operator only use columns from one of the joined tables and (2) the primary key of a joined table is a subset of the columns referenced in the group-by. The columns referenced in a pushed down group-by operator is the union of columns in the original group-by and equi-join columns of the other table.

Partial Group-By Pushdown: Even if the optimizer cannot push down the entire aggregation, it can still create a new group-by operator that computes a portion of the aggregation. Partial aggregation reduces the cardinality of the input relation to the join.

6 Schema-Specific Transformations

The optimizer can also detect common database design patterns and invoke transformations that are optimized for those patterns. One important pattern is the **star schema**, which features a large fact table connected to denormalized dimension tables. The **snowflake schema** extends the star schema by normalizing the dimension tables (i.e., breaking them down into subdimensions). Star and snowflake schemas are popular in workloads such as business intelligence, analytics, and data warehousing.

If the optimizer notices that a query is joining a fact table with multiple dimension tables (i.e., a star or snowflake query), it can transform it into a left-deep (or right-deep) join tree and order dimension tables from most to least selective immediately. In doing so, it avoids wasting time exploring bushy plans or alternative join orderings. Note that additional transformations can still be done, but this provides a good starting point.

References

[1] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 354–366, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 1558601538.

[2] B. Ding, V. Narasayya, and S. Chaudhuri. *Extensible Query Optimizers in Practice*, volume 14. Foundations and Trends® in Databases, December 2024. URL `https://www.microsoft.com/en-us/research/publication/extensible-query-optimizers-in-practice/`.

[3] C. Galindo-Legaria and A. Rosenthal. How to extend a conventional optimizer to handle one- and two-sided outerjoin. In *[1992] Eighth International Conference on Data Engineering*, pages 402–409, 1992. doi: 10.1109/ICDE.1992.213169.

[4] A. Pellenkoft, C. A. Galindo-Legaria, and M. L. Kersten. The complexity of transformation-based join enumeration. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, page 306–315, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. ISBN 1558604707.