Query Optimization

Ilaria Battiston *

Winter Semester 2020-2021

*All notes are collected with the aid of material provided by T. Neumann. All images have been retrieved by slides present on the TUM Course Webpage.

Contents

1 Introduction

Queries involve multiple steps to be performed. All techniques obviously depend on the physical characteristic of hardware, however there are algorithms and data structures able to help making them faster.

Query optimization is specifically important since SQL-like languages are declarative, hence do not specify the exact computation. Furthermore, the same query can be performed in multiple ways, and choosing the correct one is not trivial (depending on size, indices and more).

For instance, looking at cardinality of joins can either cause a Cartesian product of a hundred or several thousands, depending on the order in which they are executed. Often the human-written sequential order of instructions is not the most efficient one.

1.1 Query Processing

Query processing consists in:

1. Taking a text query as input;
2. Compiling and optimizing;
3. Extract the execution plan;
4. Executing the query.

Roughly, it can ve differentiated between compile time and runtime system. Most systems strongly separate those phases, for example using so-called prepared queries.

```
SELECT S.NAME
FROM STUDENTS S
WHERE S.ID = ?
```

Compilation takes time and limits the number of queries which can be ran in parallel. This kind of query can be executed over and over providing a value `Q1(123)` without the compilation overhead.

Embedded SQL is another technique to optimize compilation time in the case of large periods of time between the two phases. The programming language compiler takes care of the SQL part as well, optimizing it.

Specifically, the steps executed in compile time are:

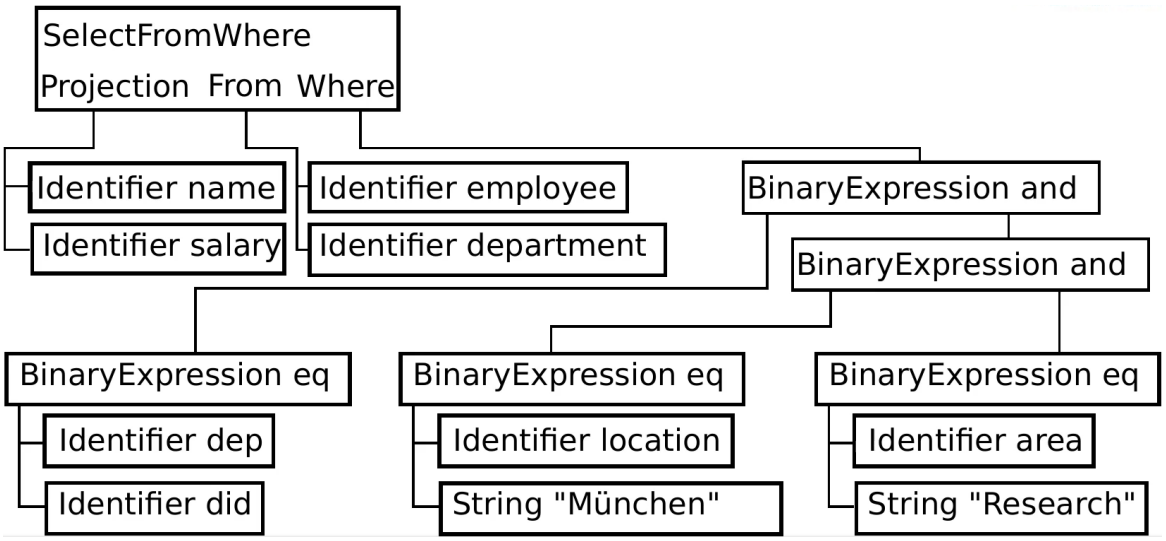
1. Parsing, AST production (abstract syntax to understand the structure);
2. Schema lookup, variable binding, type inference (semantic analysis about relations and columns, syntax check);
3. Normalization, factorization (bringing the query in abstract form, avoiding computing the same thing twice, evaluating expressions);
4. Unnesting, deriving predicates, resolution of views (the plan generator can finally construct a cost-based model);
5. Construction of execution plan;
6. Review, pushing joins and refining the plan in general;

7. Production of imperative plan (code generation).

Rewrite I involves steps 1-3, while 4 and 5 compose rewrite II.

Example (with views):

```
SELECT name, salary
FROM employee, department
WHERE dep = did
AND location = "Munich"
AND area = "Research"
```



Finally, the execution tree is built and polished, with the join operation on top and selects on nodes, reducing the amount of tuples to be joined. In other cases, such as regular expressions which are hard to evaluate, filtering can be done later.

The executable plan is a set of tuples containing variables or constants along and their type, with the main function loading query parameters, operations and allocated resources.

Usually query planners are much more complicated and have practical difficulties: for instance, a long list of AND/OR predicates (machine-generated in the order of hundreds of thousands) can make the binary tree recursion crash due to insufficient space in the stack.

1.2 Query Optimization

Possible goals of query optimization include minimizing response time, resource consumption, time to first tuple (producing the first tuple as quick as possible, for instance with search results) or maximizing throughput. This can be expressed as a cost function: most systems aim to minimize response time, having resources as constraints.

Algebraic optimization is a branch using relational algebra to find the cheapest expression equivalent to the original. However, finding the cheapest is a practically impossible problem: it is hard to test

for equivalence (numerical overflow, undecidable), the set of expressions is potentially huge and some algorithms are NP-hard (actual search space is limited and smaller than the potential one).

There are ways to transform numerical expressions in algebraic ones, yet they might be expensive: calculus is faster to evaluate than algebra.

Optimization approaches can be:

- Transformative, taking an algebraic expression and iteratively making small changes, not efficient in practice;
- Constructing, starting from small expressions and joining them, obtaining larger sets, usually the preferred approach.

1.3 Query Execution

Query execution is the last step, the one directly benefiting from optimization. In reality, operators can perform extremely specialized operations, treating data as bags (sets with duplicates) or streams.

1.3.1 Relational and physical algebra

Relational algebra for query optimization includes the operators of projection, selection and join, plus some additional ones such as mapping and grouping.

However, this does not imply an implementation, which can have a great impact, hence less abstract operators are needed due to stream nature of data (and not sets):

- Sort, according to criteria;
- Temp, which materializes the input stream making further reads cheaper and releasing memory afterwards;
- Ship, sends the input stream to a different host (for distributed computing).

Furthermore, there are different kind of joins having different characteristics:

- Nested Loop Join, the textbook implementation which is very slow (quadratic) but supports all predicates;
- Blockwise Nested Loop Join, reading chunks of the first column in memory and then the second one once (requires memory, can be improved using hashing). The left side is read $\frac{|L|}{|M|}$ times, which asymptotically is the same as above but in practice much faster;
- Sort Merge Join, scanning each column only once, but requiring sorted input ($O(n \log n)$ plus linear runtime assuming no duplicates on at least one side);
- Hybrid Hash Join, partitioning columns and joining them in memory (linear time, but working only for equi-joins).

Hash Join is usually the fastest, but with very large amount of data sorting might be more efficient. Nested Loop Join works well when one of the sides has only one tuple.

Other operators also have different implementations, such as aggregation sorting and hashing, whose speed depends on the algorithm (quick sort, heap sort etc.) and can rely or not on memory.

Physical algebra is hence the output of the query optimizer, since it fixes the exact execution runtime and does not stop to the general approach like relational algebra. It mainly helps to select which operator to use before running the query, so that optimization leads to more efficient plans.

2 Textbook Query Optimization

Textbook query optimization involves techniques to perform a rough first optimization, which however is quite simple. There is a series of steps to translate raw SQL into logical and physical plans, each of them transforming input in a more optimal form.

The output is going to be executable, but still to be improved by non-trivial methods.

2.1 Algebra and tuples

Plain relational algebra is not sufficient itself: it needs to be revisited ensuring correctness (producing the same result) within a formal model. The most relevant problem to tackle is deciding whether two algebraic expressions are the same, but this in difficult in practice.

For instance, performing a selection before a join might be correct (and faster) in case the considered criterion is equality, but can give a different result than selecting after an outer join.

To remedy this issue, it is possible to guarantee that two expressions are equivalent, not accepting false positing yet allowing false negatives.

A formal definition of tuple is an unordered mapping from attribute names to values of a domain. A schema consists in a set of attributes with domain $A(t)$.

Tuple operations are:

- Concatenation, attaching one tuple to another regardless of ordering (union);
- Projection, producing a notation $t.a$ in which it is possible to access single values or multiple $t_{|\{a,b\}}$, getting a subset of the schema.

A set of tuples with the same schema forms a relation. Sets naturally do not comply with real data, since they not allow duplicates, but are used for simplicity.

In most cases, sets and bags can be used interchangeably, but the optimizer considers different semantics: logical algebra operates on bags, physical algebra on streams and sets are only considered after an explicit duplicate elimination.

Set operations are the classic ones of union, intersection and difference, yet are subject to schema constraints. On bags, operations are performed on frequencies.

There are also free variables, which first must be bounded to be evaluated: they are essentials for predicates and algebra expressions, such as dependent joins.

It is important to note that projection removes duplicates within sets, while keeping them in bags.

There are equivalences for selection and projection useful to derive whether a different ordering produces the same output. For instance, applying selection twice is the same as applying it once with two criteria plus an AND. Commutative property also holds.

2.2 Canonical Query Translation

The canonical query translation transforms SQL into algebra expressions. The first approach involves some restrictions: it assumes no duplicates without aggregation and set operations.

The first step is translating the FROM clause:

$$F = \begin{cases} R_1 & k = 1 \\ ((\dots(R_1 \times R_2) \times \dots) \times R_k)) & \text{else} \end{cases}$$

In short all relations are joined through a cross product. The next step is translating the WHERE clause:

$$W = \begin{cases} F & \text{there is no WHERE clause} \\ \sigma_p(F) & \text{otherwise} \end{cases}$$

The SELECT clause is translated starting from the projection $a_1, \dots, a_n$ or $*$. The expression is constructed:

$$S = \begin{cases} W & \text{if the projection is ALL} \\ \prod_{a_1, \dots, a_n}(W) & \text{otherwise} \end{cases}$$

GROUP BY can also be translated, even though it is not part of the canonical translation. Let $g_1, \dots, g_n$ be the attributes in the clause and agg the aggregations within SELECT:

$$G = \begin{cases} W & \text{there is no GROUP BY clause} \\ \Gamma_{g_1, \dots, g_m:agg}(W) & \text{otherwise} \end{cases}$$

HAVING is basically the same as WHERE, with the filter predicate on top of G.

2.3 Logical Query Optimization

Once obtained the relational algebra, equivalences span the potential search space and new expressions are derived thanks to them. Of course equivalence can be applied in both ways, hence it is relevant to decide which one works better, and conditions have to be checked as well. This, however, makes the search more expensive since there are plenty of alternatives.

To speed the process up, sometimes some equivalences are ignored, even the simplest ones (for instance when choosing the join algorithm).

Query plans can only be compared if there is a cost function, often needing details which are not available merely through relational algebra (what kind of join is being used): logical query optimization is still a heuristic and requires additional steps, since it is not enough to determine the runtime.

Most algorithms, therefore, use the following strategy:

- Organization of equivalences into groups;
- Directing equivalences, deciding the preferred side and rewriting rules to apply them sequentially to the initial expression, trying to reduce the size of intermediate results.

For example, a projection on the output of a join can be preferred to a join of a projection. It is important to keep in mind that tuples are being removed in the process, and this only applies in certain circumstances (regular expressions, high selectivity of join).

The rule of thumb is simply to eliminate the most tuples during the intermediate step, to then perform computationally expensive operation with the smallest amount of data.

To summarize, the phases are:

- Breaking up conjunctive selection predicates, since simpler predicates can be moved around easier;
- Pushing selections down, reducing the number of tuples early;
- Introducing joins, which are cheaper than cross product (linear time);
- Determining join order;
- Introducing and pushing down projections, removing redundant attributes.

Some SQL queries have limitations: selections sometimes cannot be pushed down, since there might be no join predicate between tables. Choosing a different join order allows further push down.

2.4 Physical Query Optimization

Physical query optimization adds execution information to the plan, allowing actual cost calculation and optimizing over data structures, access path and operator implementation.

Data may be sorted or materialized, introducing results which can be reused and deciding where to store them.

First of all, the access path is selected: lookup can be done through index or table scan, depending on the selectivity (fraction of the data satisfying the clause): in general, above 10% a table scan is recommended.

Scanning a table might be efficient since tuples are stored adjacent in memory; using index, instead, involves traversing a tree multiple times starting from the root.

Sometimes it is useful to just store in cache the output of a view, but that also depends on the query plan: intermediate results should actually be reused.

Operator selection is replacing a logical operator with a physical one, according to semantic restrictions (most operators require equi-join).

A blockwise nested loop join is generally better than a natural join; sort merge join and hash join are better than both. In general, hash join is the best if not reusing sorts. This process must be performed for all operators: sort join requires ordered tuples, distributed databases need local data and there are multiple ways to model the properties (hashing).

Sort merge join might outperform the hash join if the amount of data is much larger than the available memory.

Materializing, on the other side, is quite relevant for nested loop joins: the first pass is expensive, but the afterwards ones are way cheaper, making it essential for multiple consumers.

2.5 Join Ordering

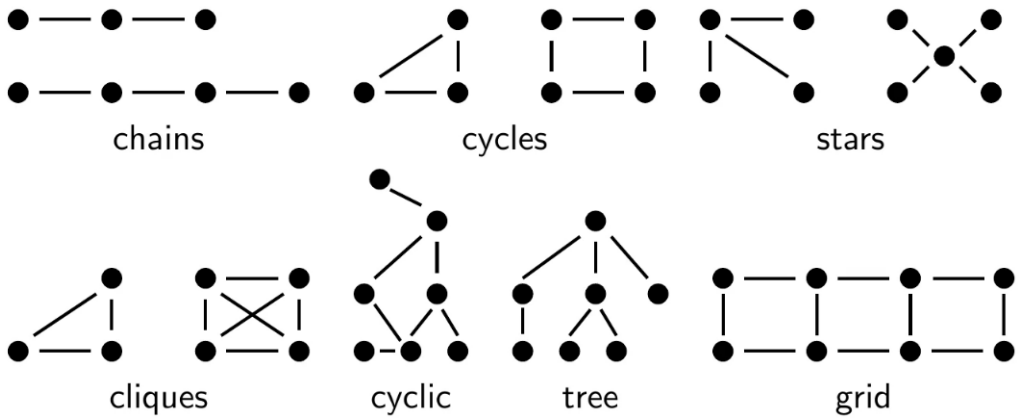
Join ordering focuses on conjunctive queries with simple predicates of the type $a_1 = a_2$ where the latter can be either an attribute or a constant (commonly algorithms assume join between attributes).

Relations may include selections or complex building blocks, however for simplicity filtering is ignored; having operators other than equality might cause differences within the query planner.

Ordering basically means choosing which relation to be joined first, placing entities in a graph and adding an edge whenever a predicate from a node is joined to another.

This kind of schema is defined as a query graph, in which edges consist in predicates and self loops represent equality with a constant. Usually cycles are pushed down, since algorithms only assume attributes.

Based on the query graph it is possible to obtain an overview of the complexity of the problem: there are different shapes which are treated differently.



1. Chains are the simplest kind of query, fairly common in practice;
2. Cycles (cyclic) are a chain with a closing edge, the easiest example of cycles;
3. Stars are mostly used in data warehouse, in which the center table has large dimension and the ones outside are relatively small, quite different to solve;
4. Cliques are instances in which every relation is joined with all the others, and are the hardest to optimize causing the worst runtime;
5. Trees are acyclic queries even if the level of nesting can be high;
6. Grids are also fairly hard and interesting for research.

Joins are represented with join trees, binary trees with operators as inner nodes and relations as leaves. The most common type is unordered (not distinguish left from right) without cross product, however algorithms might produce other variants.

There furthermore are different kinds of trees:

- Left-deep tree, in which joins only happen on the left side, easy to represent and implement through hash tables ($n!$ trees with cross products);
- Right-deep tree ($n!$);
- Zig-zag tree, a combination of the previous ($n!2^{n-2}$);

- Bushy tree, a full binary tree (non-linear, harder to find optimal solutions but can be the most efficient in some cases, $n!C(n-1) = \frac{(2n-2)!}{(n-1)!}$ where C represents a Catalan number).

It is relevant to notice that the number of leaf combinations and unlabeled trees grows exponentially, and increases even more with a flexible structure. However, nodes can often be swapped from left to right.

Another important information about joins is their selectivity:

$$f_{i,j} = \frac{\left|R_i \bowtie_{p_{i,j}} R_j\right|}{|R_i \times R_j|}$$

This depends on whether the attributes are a key, and gives an estimation of the result cardinality with the aid of assumptions and statistics.

Given a join tree, the cardinality can be computed recursively as the productory of the selectivity function multiplied by the size of both relations. This allows easy calculations only requiring base cardinalities and independence of predicates:

$$C_{out}(T) = \begin{cases} 0 & T \text{ is a leaf} \\ |T| + C_{out}(T_1) + C_{out}(T_2) & T = T_1 \bowtie T_2 \end{cases}$$

This formula sums up the sizes of intermediate results, which are the ones causing more works. There are basic specific cost functions for joins, to be summed to the cost of single relations.

Algorithms are mainly designed for left-deep trees, and some of the cost functions do not work in practice, for instance in the case of cross products. Therefore, those indicators are mainly theoretical and work under strict assumptions. However, join ordering is a main factor regardless of the chosen cost methods.

A cost function is called symmetric if $C_{impl}(e_1 \bowtie^{impl} e_2) = C_{impl}(e_2 \bowtie^{impl} e_1)$. Commutativity can be ignored.

An ASI (Adjacent Sequence Interchange) is a set of properties classifying functions. Both these concepts can be exploited during optimization.