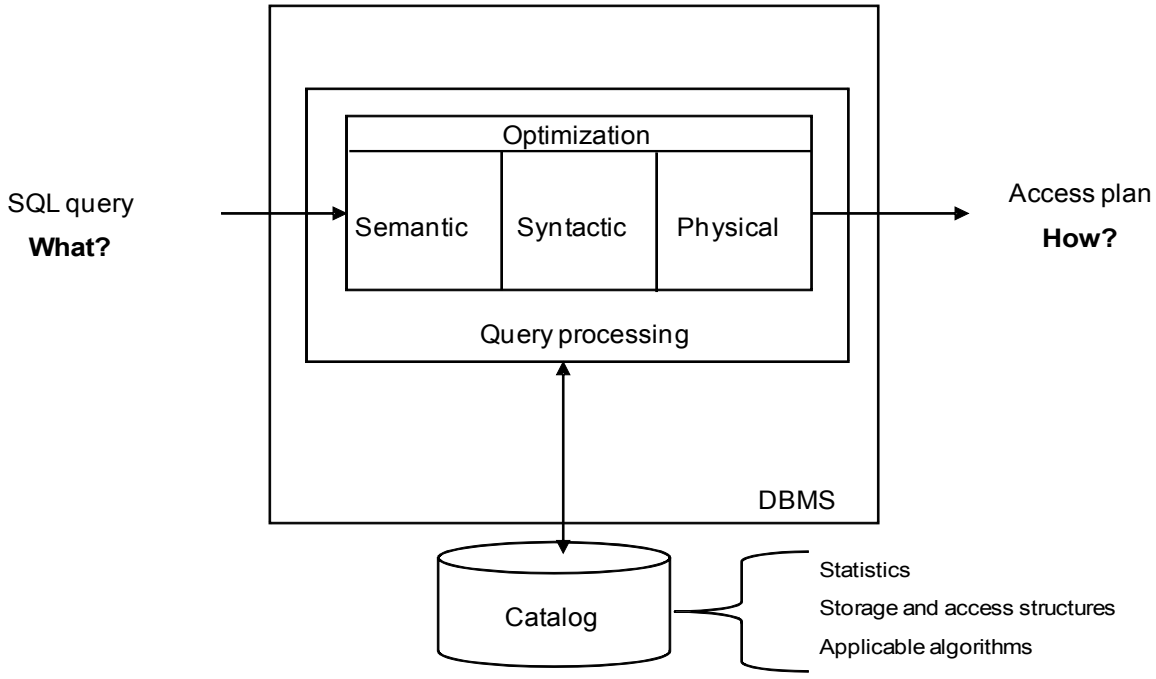
Introduction to query optimization

SQL is a declarative language, by means of which users state the data they want to obtain but not the way it must be retrieved. Given an SQL sentence, the objective of query processing is to retrieve the corresponding data. It follows the following steps:

1. Validation
   - Correct syntax
   - Check permissions
   - Expand views
   - Check table schema
2. Optimization
   a. Semantic
   b. Syntactic
   c. Physical
3. Evaluation (i.e. disk access)

Given a query, there are many strategies that a database management system (DBMS) can follow to process it and produce its answer. All of them are equivalent in terms of their final output but vary in their cost, that is, the amount of time that they need to run. This cost difference can be several orders of magnitude large. Thus all DBMSs have a module that examines "all" alternatives and chooses the plan that needs the least amount of time. This module is called the *query optimizer*.



The optimizer is a module inside the DBMS, whose input is an SQL query and its output is an access plan expressed in a given procedural language. Its objective is to obtain an execution algorithm as good as possible based on the contents of the catalog:

- Contents statistics (for example, number of tuples, size of attributes, etc.)
- Available storage structures (for example, partitions and materialized views)
- Available access structures (for example, B-tree and hash indexes)
- Applicable algorithms (mainly for join and sorting)

In general, a DBMS does not find the optimal access plan, but it obtains an approximation (in a reasonable time). Finding the optimum is computationaly hard (NP-complex in the number of relations involved), resulting in higher costs than just retrieving the data. Therefore, DBMSs use heuristics to approximate it, which means that some times they do not obtain the optimum, but they use to be close. It is important to know how the optimizer works to detect such deviations and correct them, when possible (for example, adding or removing some indexes, partitions, etc.).

The cost function typically refers to machine resources such as disk space, disk input/output, buffer space, and CPU time. In current centralized systems where the database resides on disk storage, the emphasis is on minimizing the number of disk accesses.

Even though its implementation could not be that modular, we can study the optimization process as if executed in three sequential steps (i.e. Semantic, Syntactic and Physical).

Semantic Optimization

Consists of transforming (i.e. rewriting) the SQL sentence into an equivalent one with a lower cost, by considering:

- Integrity constraints
- Logics properties (i.e. transitivity, etc.)

This module applies transformations to a given query and produces equivalent queries intended to be more efficient, for example, standardization of the query form, flattening out of nested queries, and the like. It aims to find incorrect queries: incorrect form or contradictory (having an incorrect form means that there is a better way to write them from the point of view of performance, while being contradictory means that their result is going to be the empty set). The transformations performed depend only on the declarative, that is, static, characteristics of queries and do not take into account the actual query costs for the specific DBMS and database concerned.

Replicating clauses over one side of an equality to the other is a typical example of transformation performed at this phase of the optimization. This may look a bit naive, but it allows to use indexes over both attributes. For example, if the optimizer finds "a=b AND a=5", it will transform it into "a=b AND a=5 AND b=5", so that if there is and index over "b", it can also be used.

Another example of optimization is removing disjunctions. For example, we may transform the disjunction of two equalities over the same attribute into an "IN". Reducing this way the number of clauses in the selection predicate would also reduce its evaluation cost. However, such transformation is not easily detected in complex predicates and can only be performed in the most simple cases.

In general, semantic optimization is really poor in most (if not all) DBMSs, and only useful in simple queries. Therefore, even though SQL is considered a declarative language, the way we write the sentence can hide some optimizations and affect its performance. Thus, a simple way to optimize a query (without any change in the physical schema of the database) may be just rewriting it in a different way.

Syntactic Optimization

Consists of translating the sentence from SQL into a sequence of algebraic operations in the form of syntactic tree. There may be many sequences of algebraic operations corresponding to the same SQL query. The optimizer will choose the one with minimum cost, by means of heuristics. These sequences are usually represented in relational algebra as formulas or in tree form. The interpretation of the tree is as follows:

- Nodes
    - Root: Result
    - Internal: Algebraic Operations
    - Leaves: Tables
- Edges: direct usage

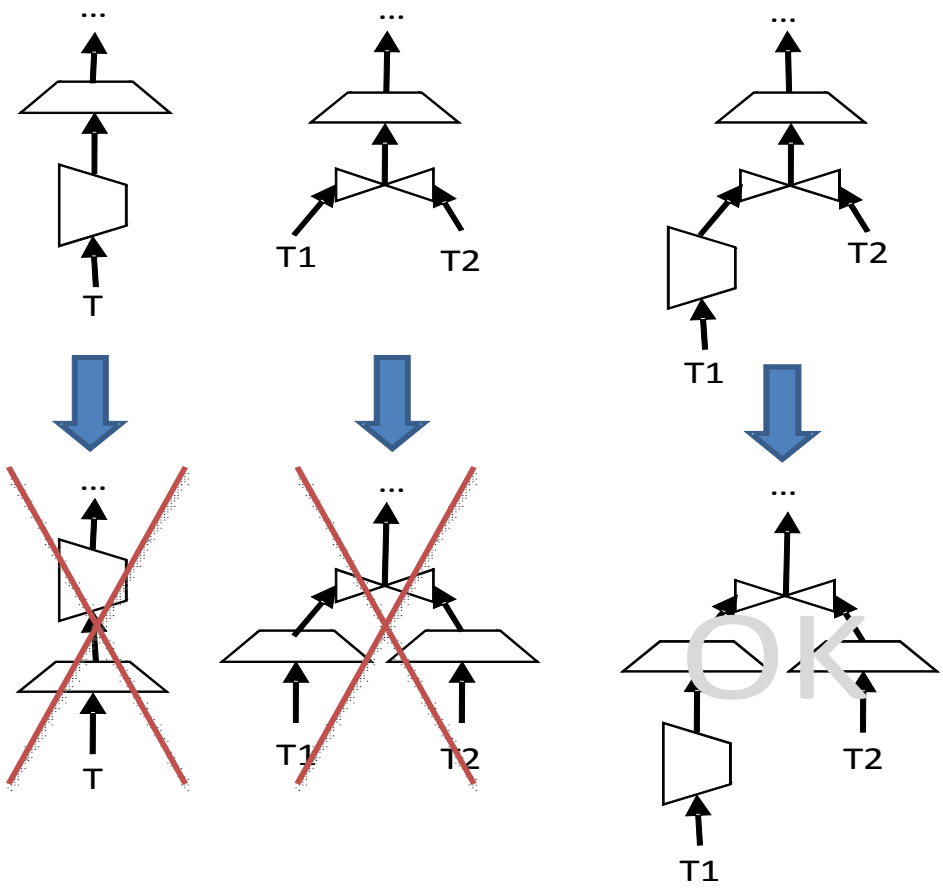
This module determines the orderings of the necessary operators to be considered by the optimizer for each query sent to it. The objective is to reduce the size of data passing from the leaves to the root (i.e. the output of the operations corresponding to the intermediate nodes in the tree), which can be mainly done in two different ways: (i) reduce the number of attributes as soon as possible, and (ii) reduce the number of tuples as soon as possible. To do this, we use the equivalence rules in the slides:

I. Splitting/grouping selections
II. Commutating the precedence of selection and join
III. Commutating the precedence of selection and set operations (i.e. Union, Intersection and Difference)
IV. Commutating the precedence of selection and projection, when the selection attribute is projected
V. Commutating the precedence of selection and projection, when the section attribute is not projected
VI. Commutating the precedence of projection and join, when the join attributes are projected
VII. Commutating the precedence of projection and join, when some join attribute is not projected
VIII. Commutating the precedence of projection and union. Notice that intersection and difference do not commutate. For example,
R[A,B]={[a,1]} S[A,B]={[a,2]}
R[A]-S[A]=∅
(R-S)[A]={[a]}
IX. Commutating join branches
X. Associating join tables

We use the equivalence rules to apply two heuristics that usually drive to the best access plan: (i) execute projections as soon as possible, and (ii) execute selections as soon as possible (notice that since they are heuristics, some times this does not result in the best option). We will follow the algorithm:

1. Split the selection predicates into simple clauses (usually, the predicate is firstly transformed into Conjunctive Normal Form – CNF)

   (x OR y) AND (z OR t OR …) AND ...
2. Lower selections as much as possible
3. Group consecutive selections (simplify them if possible)
4. Lower projections as much as possible (do not leave them just on a table, except when one branch leaves the projection on the table and the other does not)



5. Group consecutive projections (simplify them if possible)

The last two equivalence rules (i.e. IX and X) would be used to generate alternative trees. However, for the sake of understandability, we will assume this is not part of the syntactic optimization algorithm, but done during the next step.

It is also part of the syntactic optimization to detect disconnect parts of the query, if any, and simplify tautologies ($R \cap \varnothing = \varnothing$, $R - R = \varnothing$, $\varnothing - R = \varnothing$, $R \cap R = R$, $R \cup R = R$, $R \cup \varnothing = R$, $R - \varnothing = R$). Detecting disconnect tables (those with no join condition in the predicate) in a query can easily be done. However, in this case no error uses to be thrown. Instead, a cartesian product is performed by most (if not all) DBMSs. Moreover, in some cases, the tree is transformed into just a Directed Acyclic Graph (DAG) by fusing nodes if they correspond to exactly the same relational operation with the same parameters (for example, the same selection operation in two different subqueries of the same SQL sentence).

Physical Optimization

Given a Syntactic Tree, this module produces all corresponding complete access plans that specify the implementation of each algebraic operator and the use of any indices. Specifically, it consists of generating the execution plan of a query considering:

- Physical structures
- Access paths
- Algorithms

This is the main module of the query processor. It employs a *search strategy* that explores the space of access plans determined by the Algebraic Tree produced in the previous stage. It compares these plans based on estimates of their cost and selects the overall cheapest one to be used to generate the answer to the original query. First of all, to do it, we transform the Syntactic Tree into a Process Tree. This is the tree associated to the Syntactic Tree that models the execution strategy. The interpretation of the tree is as follows:

- Nodes
    - Root: Result
    - Internal: Intermediate tables generated by a physical operation
    - Leaves: Tables (or Indexes)
- Edges: direct usage

To reduce the size of the space that the search strategy must explore, DBMSs usually impose various restrictions. Typical examples include: never generating unnecessary intermediate relations (i.e., intermediate selections and projections are processed on the fly). The difference between both trees is that now the nodes represent real steps that will be executed (for example, most projections disappear by fusing them with the previous operation in the data flow of just being removed if they lay on a table). In a real optimizer, selections that are not on a table would also be fused with the previous operation, however we will assume they are not. Moreover, grouping and sorting operations are also added now to the Process Tree.

There are four steps in the physical optimizer:

1. Alternatives generation
2. Intermediate results cardinality and size estimation
3. Cost estimation for each algorithm and access path
4. Choose the best option and generate the access plan

During step 1, this module determines the implementation choices that exist for the execution of each operator ordering specified by the Algebraic Tree. These choices are related to the available join methods for each join (e.g., nested loops, merge join, and hash join), if/when duplicates are eliminated, and other implementation characteristics of this sort, which are predetermined by the DBMS implementation. They are also related to the available indices for accessing each relation, which are determined by the physical schema of each database.

We would also construct at this point all alternative trees by iterating on the number of relations joined so far (using equivalence rules IX and X). The memory requirements and running time grow exponentially with query size (i.e., number of joins) in the worst case. Most queries seen in practice, however, involve less than 10 joins, and the algorithm has proved to be very effective in such contexts. For a complicated query, the number of all orderings may be enormous.

During step 2, this module estimates the sizes of the results of (sub)queries and the frequency distributions of values in attributes of these results, which are needed by the cost model. Most commercial DBMSs, however, base their estimation on assuming a uniform distribution (i.e., all attribute values having the same frequency).

During step 3, this module specifies the arithmetic formulas used to estimate the cost of access plans. For every different join method, different index-type access, and in general for every distinct kind of step that can be found in an access plan, there is a formula that gives an (often approximate) cost for it.

Despite all the work that has been done on query optimization there are many questions for which we do not have complete answers, even for the most simple, single-query, relational optimizations. Moreover, several advanced query optimization issues are active topics of research. These include parallel, distributed, semantic, object-oriented, and aggregate query optimization, as well as optimization with materialized views, optimization with expensive selection predicates, and query optimizer validation.

Bibliography

[GV89] Georges Gardarin and Patrick Valduriez. “Relational Databases and Knowledge Bases”. Addison-Wesley, 1998.
[Ioa96] Yannis Ioannidis. "Query Optimization". ACM Computing Surveys, vol. 28, num. 1, March 1996.
[LTN07] Sam Lightstone, Toby Teorey and Tom Nadeau. "Physical Database Design". Morgan Kaufmann, 2007.
[SOO+02] Jaume Sistac, Marta Oliva, Santiago Ortego and Ramón Segret. "Sistemes de Gestió de Bases de Dades". Editorial UOC, 2002.