Query Optimization

YANNIS E. IOANNIDIS

Computer Sciences Department, University of Wisconsin, Madison ⟨yannis@cs.wisc.edu⟩

Given a query, there are many access plans that a database management system (DBMS) can follow to process it and produce its answer. All plans 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*.

Query optimization is a large area within the database field and has been surveyed extensively [Jarke and Koch 1984; Mannino et al. 1988]. This short paper emphasizes optimization of a single select-project-join query in a centralized relational DBMS.

An abstraction of the query optimization process, divided into a *rewriting* and a *planning* stage, is shown in Figure 1.[1] The functionality of each module in Figure 1 is described in the following.

Rewriter. This module applies transformations to a given query and produces equivalent queries intended to be more efficient, for example, standardization of the query form, replacement of views by their definition, flattening out of nested queries, and the like. The transformations performed by the Rewriter 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.

Planner. This is the main module of the ordering stage. It employs a *search strategy* that explores the space of access plans determined by the *Algebraic Space* and the *Method-Structure Space* modules for each query produced in the previous stage. It compares these plans based on estimates of their cost derived by the *Cost Model* and the *Size-Distribution Estimator* modules and selects the overall cheapest one to be used to generate the answer to the original query.

There are several types of search strategies that the Planner may employ for its exploration. By far the most important one is based on dynamic programming. It was first proposed in the context of the System R prototype [Selinger et al. 1979] and is currently used (in various forms) by essentially all commercial systems. It constructs all alternative access plans by iterating on the number of relations joined so far, always pruning plans known to be suboptimal. The memory requirements and running time of dynamic programming 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 really large queries, which appear in various novel database applications, several other algorithms

[1] Figure 1 is essentially a modular architecture of a query optimizer. Although one could build an optimizer based on this architecture, in real systems, the modules shown do not always have such clear-cut boundaries.

This work was partially supported by the National Science Foundation under Grants IRI-9113736 and IRI-9157368 (PYI Award), by Lockheed as part of an MDDS contract, and by grants from IBM, DEC, HP, AT&T, Informix, and Oracle.

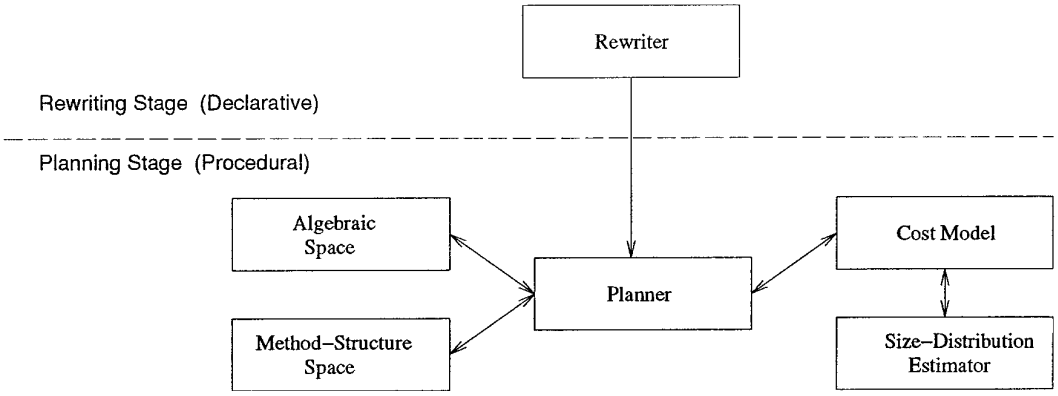

Figure 1. Query optimizer architecture.

have been proposed. Of these, randomized algorithms, for example, simulated annealing, iterative improvement, and two-phase optimization, appear very promising [Ioannidis and Kang 1990; Swami and Gupta 1988].

Algebraic Space. This module determines the orderings of the necessary operators to be considered by the Planner for each query sent to it. These are usually represented in relational algebra as formulas or in tree form. For a complicated query, the number of all orderings may be enormous. 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., selections and projections are processed on the fly); never forming unnecessary cross products; and never having an intermediate result as the inner operand of a join (i.e., it should always be a database relation).

Method-Structure Space. This module determines the implementation choices that exist for the execution of each operator ordering specified by the Algebraic Space. These choices are related to the available join methods for each join (e.g., nested loops, merge scan, and hash join), if supporting data structures are built on the fly, 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. Given an algebraic formula or tree from the Algebraic Space, this module produces all corresponding complete access plans that specify the implementation of each algebraic operator and the use of any indices.

Cost Model. 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 (as prescribed by the Method-Structure Space), there is a formula that gives an (often approximate) cost for it.

Size-Distribution Estimator. 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. Several techniques have been proposed in the literature to estimate query result sizes and frequency distributions [Mannino et al. 1988], for example, sampling or using polynomial or statistical approximations. Most commercial DBMSs, however, base their es-

timation on *histograms*. A histogram is formed by partitioning the domain of an attribute into *buckets* and assuming a uniform distribution within each bucket (i.e., all attribute values in the bucket having the same frequency). Commercial systems typically use *equi-width* histograms [Kooi 1980], in which buckets are associated with equal-sized ranges of the domain of the attribute. Although not yet used commercially, several other histogram types have been proposed that produce better estimates, for example, *equi-depth* [Kooi 1980; Piatetsky-Shapiro and Connell 1984] and *serial/end-biased* [Ioannidis and Poosala 1995].

Despite all the work that has been done on query optimization, in every single module of the architecture of Figure 1, 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, global, parametric, dynamic, nested, rule-based, object-oriented, heterogeneous, recursive, and aggregate query optimization, as well as query optimizer generators, optimization with materialized views, optimization with expensive selection predicates, and query optimizer validation. Despite its age, query optimization remains an exciting field.

REFERENCES

IOANNIDIS, Y. AND KANG, Y. 1990. Randomized algorithms for optimizing large join queries. In *Proceedings of the 1990 ACM-SIGMOD Conference on the Management of Data* (Atlantic City, NJ, May) 312–321.

IOANNIDIS, Y. AND POOSALA, V. 1995. Balancing histogram optimality and practicality for query result size estimation. In *Proceedings of the 1995 ACM-SIGMOD Conference on the Management of Data* (San Jose, CA, May) 233–244.

JARKE, M. AND KOCH, J. 1984. Query optimization in database systems. *ACM Comput. Surv. 16,* 2 (June), 111–152.

KOOI, R. P. 1980. *The optimization of queries in relational databases.* Case Western Reserve University, Ph.D. Thesis, Sept.

MANNINO, M. V., CHU, P., AND SAGER, T. 1988. Statistical profile estimation in database systems. *ACM Comput. Surv. 20,* 3, (Sept.), 192–221.

PIATETSKY-SHAPIRO, G. AND CONNELL, C. 1984. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the 1984 ACM-SIGMOD Conference on the Management of Data* (Boston, MA, June), 256–276.

SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. 1979. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Symposium on Management of Data,* (Boston, MA, June), 23–34.

SWAMI, A. AND GUPTA, A. 1988. Optimization of large join queries. In *Proceedings of the 1988 ACM-SIGMOD Conference on the Management of Data,* (Chicago, IL, June), 8–17.