QUERY PROCESSING IN A RELATIONAL DATABASE MANAGEMENT SYSTEM

GAWANDE BALAJI RAMRAO
Research Scholar, Dept. of Computer Science
CMJ University, Shillong, Meghalaya

ABSTRACT

Database management systems will continue to manage large data volumes. Thus, efficient algorithms for accessing and manipulating large sets and sequences will be required to provide acceptable performance. The advent of object-oriented and extensible database systems will not solve this problem. On the contrary, modern data models exacerbate it: In order to manipulate large sets of complex objects as efficiently as today's database systems manipulate simple records, query processing algorithms and software will become more complex, and a solid understanding of algorithm and architectural issues is essential for the designer of database management software.

This survey provides a foundation for the design and implementation of query execution facilities in new database management systems. It describes a wide array of practical query evaluation techniques for both relational and post-relational database systems, including iterative execution of complex query evaluation plans, the duality of sort- and hash-based set matching algorithms, types of parallel query execution and their implementation, and special operators for emerging database application domains.

Categories and Subject Descriptors: *E.5 [Data]: files, retrieval, searching, sorting; H.2.4 [Database Management]: query processing, query evaluation algorithms, parallelism, systems architecture.*

General Terms: *Software, architecture, algorithms, parallelism, performance.*

Additional Keywords and Phrases: *Relational, Extensible, and Object-Oriented Database Systems; Iterators; Complex Query Evaluation Plans; Set Matching Algorithms; Sort-Hash Duality; Dynamic Query Evaluation Plans; Operator Model of Parallelization; Parallel Algorithms; Emerging Database Application Domains.*

INTRODUCTION

Effective and efficient management of large data volumes is necessary in virtually all computer applications, from business data processing to library information retrieval systems, multimedia applications with images and sound, computer-aided design and manufacturing, real-time process control, and scientific computation. While database management systems are standard tools in business data processing, they are only slowly being introduced to all the other emerging database application areas.

In most of these new application domains, database management systems have traditionally

not been used for two reasons. First, restrictive data definition and manipulation languages can make application development and maintenance unbearably cumbersome. Research into semantic and object-oriented data models and into persistent database programming languages has been addressing this problem and will eventually lead to acceptable solutions. Second, data volumes might be so large or complex that the real or perceived performance advantage of file systems is considered more important than all other criteria, e.g., the higher levels of abstraction and programmer productivity typically achieved with database management systems. Thus, object-oriented database management systems that are designed for non-traditional database application domains and extensible database management systems toolkits that support a variety of data models must provide excellent performance to meet the challenges of very large data volumes, and techniques for manipulating large data sets will find renewed and increased interest in the database community.

The purpose of this paper is to survey efficient algorithms and software architectures of database query execution engines for executing complex queries over large databases. A "complex" query is one that requires a number of query processing algorithms to work together, and a "large" database uses files with sizes from several megabytes to many terabytes, which are typical for database applications at present and in the near future [Dozier 1992; Silberschatz, Stonebraker, and Ullman 1991]. This survey discusses a large variety of query execution techniques that must be considered when designing and implementing the query execution module of a new database management system: algorithms and their execution costs, sorting vs. hashing, parallelism, resource allocation and scheduling issues in complex queries, special

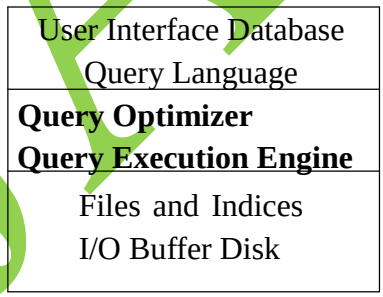

Figure 1. Query Processing in a Database System.

operations for emerging database application domains such as statistical and scientific databases, and general performance-enhancing techniques such as precomputation and compression. While many, although not all, techniques discussed in this paper have been developed in the context of relational database systems, most of them are applicable to and useful in the query processing facility for any database management system and any data model, provided the data model permits queries over "bulk" data types such as sets and lists.

It is assumed that the reader possesses basic textbook knowledge of database query languages, in particular of relational algebra, and of file systems, including some basic knowledge of index structures. As shown in Figure 1, query processing fills the gap between database query languages and file systems. It can be divided into query optimization and query execution. A query optimizer translates a query expressed in a high-level query language into a sequence of operations that are implemented in the query execution engine or the file system. The goal of query optimization is to find a query evaluation plan that minimizes the most relevant performance measure, which can be the database user's wait for the first or last result item, CPU, I/O, and network time and effort (time and effort can differ due to parallelism), the time-space-product of locked database items and their lock duration, memory costs (as maximum allocation or as time-space product), total resource usage, even energy consumption (e.g., for battery-powered laptop systems or space craft), a combination of the above, or some other performance measure. Query optimization is a special form of planning, employing techniques from artificial intelligence such as plan representation, search including directed search and pruning, dynamic programming, branch-and-bound algorithms, etc. The query execution engine is a collection of query execution operators and mechanisms for operator communication and synchronization — it employs concepts from algorithm design, operating systems, networks, and parallel and distributed computation. The facilities of the query execution engine define the space of possible plans that can be chosen by the query optimizer.

SORTING

Sorting is used very frequently in database systems, both for presentation to the user in sorted reports or listings and for query processing in sort-based algorithms such as merge-join. Therefore, the performance effects of the many algorithmic tricks and variants of external sorting

deserve detailed discussion in this survey. All sorting algorithms actually used in database systems use merging, i.e., the input data are written into initial sorted runs and then merged into larger and larger runs until only one run is left, the sorted output. Only in the unusual case that a data set is smaller than the available memory can in-memory techniques such as quicksort be used. An excellent reference for many of the issues discussed here is Knuth [Knuth 1973], who analyzes algorithms much more accurately than we do in this introductory survey.

In order to ensure that the sort module interfaces well with the other operators, e.g., file scan or merge-join, sorting should be implemented as an iterator, i.e., with *open*, *next*, and *close* procedures as all other operators of the physical algebra. In the Volcano query processing system (which is based on iterators), most of the sort work is done during *open-sort* [Graefe 1990a; Graefe 1993c]. This procedure consumes the entire input and leaves appropriate data structures for *next-sort* to produce the final, sorted output. If the entire input fits into the sort space in main memory, *open-sort* leaves a sorted array of pointers to records in I/O buffer memory which is used by *next-sort* to produce the records in sorted order. If the input is larger than main memory, the *open-sort* procedure creates sorted runs and merges them until only one final merge phase is left. The last merge step is performed in the *next-sort* procedure, i.e., when demanded by the consumer of the sorted stream, e.g., a merge-join. The input to the sort module must be an iterator, and sort uses *open*, *next*, and *close* procedures to request its input; therefore, sort input can come from a scan or a complex query plan, and the sort operator can be inserted into a query plan at any place or at several places.

HASHING

For many matching tasks, hashing is an alternative to sorting. In general, when equality matching is required, hashing should be considered because the expected complexity of set algorithms based on hashing is $O(N)$ rather than $O(N \log N)$ as for sorting. Of course, this makes intuitive sense if hashing is viewed as radix sorting on a virtual key [Knuth 1973].

Hash-based query processing algorithms use an in-memory hash table of database objects to perform their matching task. If the entire hash table (including all records or items) fits into memory, hash-based query processing algorithms are very easy to design, understand, and implement, and outperform sort-based alternatives. Note that for binary matching operations, such as join or intersection, only one of the two inputs must fit into memory. However, if the required hash table is larger than memory, *hash table overflow* occurs and must be dealt with.

There are basically two methods for managing hash table overflow, namely *avoidance* and *resolution*. In either case, the input is divided into multiple partition files such that partitions can be processed independently from one another and the concatenation of the results of all partitions is the result of the entire operation. Partitioning should ensure that the partitioning files are of roughly even size, and can be done using either hash-partitioning or range-partitioning, i.e., based on keys estimated to be quantiles. Usually, partition files can be processed using the original hash-based algorithm. The maximal partitioning *fan-out* F, i.e., number of partition files created, is determined by the memory size M divided over the cluster size C minus one cluster for the partitioning input, i.e., $F = \lfloor M / C - 1 \rfloor$, just like the fan-in for sorting.

(IJAER) 2012, Vol. No. 4, Issue No. VI, December ISSN: 2231-5152

In hash table overflow avoidance, the input set is partitioned into *F* partition files before any in-memory hash table is built. If it turns out that fewer partitions than have been created would have been sufficient to obtain partition files that will fit into memory, bucket tuning (collapsing multiple small buckets into larger ones) and dynamic destaging (determining which buckets

should stay in memory) can improve the performance of hash-based operations [Kitsuregawa, Nakayama, and Takagi 1989; Nakayama, Kitsuregawa, and Takagi 1988].

REFERENCES

Atkinson and Buneman 1987: M. P. Atkinson and O. P. Buneman, Types and Persistence in Database Programming Languages, *ACM Computing Surveys 19*, 2 (June 1987), 105.

Babb 1979: E. Babb, Implementing a Relational Database by Means of Specialized Hardware, *ACM Trans. on Database Sys. 4*, 1 (March 1979), 1.

Guttman 1984: A. Guttman, R-Trees: A Dynamic Index Structure for Spatial Searching, *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984, 47. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan- Kaufman, San Mateo, CA, 1988.

Haas et al. 1982: L. M. Haas, P. G. Selinger, E. Bertino, D. Daniels, B. Lindsay, G. Lohman, Y. Masunaga, C. Mo- han, P. Ng, P. Wilms, and R. Yost, *R*: A Research Project on Distributed Relational DBMS*, IBM Res. Divi- sion, San Jose CA, October 1982.

Lyytinen 1987: K. Lyytinen, Different Perspectives on Information Systems: Problems and Solutions, *ACM Com- puting Surveys 19*, 1 (March 1987), 5.

Mackert and Lohman 1989: L. F. Mackert and G. M. Lohman, Index Scans Using a Finite LRU Buffer: A Validated I/O Model, *ACM Trans. on Database Sys. 14*, 3 (September 1989), 401.

Richardson and Carey 1987: J. E. Richardson and M. J. Carey, Programming Constructs for Database System Implementation in EXODUS, *Proc. ACM SIGMOD Conf.*, San Francisco, CA, May 1987, 208.

Sun et al. 1993: W. Sun, Y. Ling, N. Rishe, and Y. Deng, An Instant and Accurate Size Estimation Method for Joins and Selections in a Retrieval-Intensive Environment, *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993, 79.

Tansel and Garnett 1992: A. U. Tansel and L. Garnett, On Roth, Korth, and Silberschatz's Extended Algebra and Calculus for Nested Relational Databases, *ACM Trans. on Database Sys. 17*, 2 (June 1992), 374.

Wong and Katz 1983: E. Wong and R. H. Katz, Distributing a Database for Parallelism, *Proc. ACM SIGMOD Conf.*, San Jose, CA, May 1983, 23.

Yang and Larson 1987: H. Yang and P. A. Larson, Query Transformation for PSJ-queries, *Proc. Int'l. Conf. on Very Large Data Bases*, Brighton, England, August 1987, 245.