Dynamic Query Evaluation Plans: Some Course Corrections?

Goetz Graefe
SQL Server Development
Microsoft Corporation
goetzg@microsoft.com

Purpose: In database query processing, optimization is commonly regarded as "the hard" part, whether in relational, post-relational, object-oriented, textual-spatial-temporal, federated or web-based database systems. Query execution, on the other hand, is considered a mostly straightforward exercise in algorithm implementation, with the currently "hot" twist to consider CPU caches. There is, however, a third piece to the puzzle, namely physical database design, e.g., the set of useful indexes. The purpose of this brief paper is to tie these three pieces together and to encourage students and researchers to adopt a broader perspective of adaptive query processing. Another purpose is to present some contrarian viewpoints about interesting and worthwhile research topics.

Compilation effort: It is well known that compilation and early binding are good ideas, where they can be applied. Thus, compile-time optimization wins over interpretation, in particular for repetitive query execution, which typically is the bulk of database activity. On the other hand, query optimization based on incomplete information often results in plans that perform poorly in many invocations. The Achilles heel of query optimization is selectivity estimation; thus, missing statistics such as histograms and counts of unique values are a perennial worry for database administrators, unless the DBMS automatically creates, refreshes, and drops statistics as appropriate. The other important source of selectivity estimation errors is lacking compile-time knowledge about run-time parameter values. Traditionally, the only remedy has been run-time optimization. Early research into dynamic or adaptive plans was motivated to address this issue. The basic theme was that the optimized query plan contains, in some reasonably compact form, multiple alternative execution plans, and selects among them at start-up time or even at some few carefully pre-planned points during execution. It turned out, not surprisingly, that execution primitives were relatively easy to design, whereas building effective optimizers for dynamic plans is still much more art than science, and could greatly benefit from more research into practical techniques, which then would greatly facilitate technology transfer into products. After all, the point of dynamic plans is to avoid the effort for frequent re-compilation and re-optimization; if the effort for finding a single dynamic plan is just as large as or even larger than the effort for repeated re-compilation, not much is gained, other than perhaps start-up latency.

Cost estimation and dynamic execution techniques: A lesser source of query optimization errors is fluctuating resource availability and therefore inaccurate cost calculations. For example, the relative costs of index-to-index-and-record-to-record navigation compared to set-oriented sort- and hash-based query plans depends on the available memory, available CPU processing bandwidth, available disk bandwidth for permanent and temporary

Copyright 2000 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

files and tables, etc. Consequently, there have been some, albeit not very many, proposals to address changes in resource availability using dynamic query evaluation plans. However, rather than using dynamic plans with alternative algorithms or alternative plan shapes, more typically resource availability is left to individual algorithms. Thus, there are many proposals and literature on dealing with resources within a single operator or algorithm, in particular memory resources, e.g., hybrid hash join, dynamic destaging, merge optimizations in external merge sort, etc. An interesting algorithm for data-driven join processing in memory-rich parallel and distributed systems is the symmetric hash join using two hash tables. Rather few papers, however, deal with memory management among multiple operators, in nested queries, or in dynamic query plans. In modern commercial query execution engines, there are many adaptive techniques, not only within operators but also among operators within a single query and among multiple concurrent queries. Examples include setting or adjusting the degree of parallelism depending on current load, reordering and merging ranges to optimize repeated probes into an index, sharing scans among multiple queries and user connections, ignoring read-ahead hints if most are buffer hits and therefore pointless, "bailing out" of hash partitioning and resorting to loops- or merge-based algorithms when partitioning is not effective (e.g., due to duplicate key values), etc. In spite of this wealth of adaptability, all of these adaptive techniques are typically ignored in the cost functions of commercial query optimizers, partially because they are difficult to incorporate and partially because a sufficiently strong case for incorporating them has not yet been made. What does that say about techniques as adaptive as dynamic query evaluation plans?

Nested iteration: While most resource issues have relatively little impact on optimal plan choices (even if they affect the ranking among different plans of fairly similar costs), one issue that is crucial in practice but usually ignored in academic research is the effect of buffer hits and faults in complex query plans. In particular nested SQL queries as well as nested iteration as operator in both the logical and physical query algebra have been usually neglected in research. Interesting issues arise if multiple invocations of the same nested computation affect each other, e.g., the first invocation warms up the I/O buffer for subsequent ones. Other interesting issues arise if different nested computations compete for resources, e.g., I/O buffer or memory for sort and hash operations within inner queries. Nested computations are very important in practice, both because queries are authored using nested SQL queries and because nested iteration based on index-to-index navigation often is the best execution plan. Therefore, nested computations could be a very fruitful research topic, both execution and optimization, and could probably also benefit from more dynamic and adaptive techniques than those in use today. However, we will need a conceptual model of nested queries that is substantially simpler that nested SQL, e.g., based on algebra expressions with a table of parameter values.

Indexed views: Perhaps the most exciting recent development in database query processing has been the commercial use of materialized views. In a way, materialized views take the concept of early binding a step further than compile-time optimization. The benefit can be tremendous, as many OLAP tools and applications demonstrate every day with sub-second response times even for very large volumes of detail data. In effect, when materialized views work really effectively, complex query processing is reduced to index navigation very similar to OLTP processing, except for differences in the update load. Fancy techniques for large queries, e.g., shared scans, bitmap indexes, hash joins, and parallel query processing might once again be of little importance, just as they were when databases were used only for business operations (OLTP), not data analysis and business intelligence. There are many issues with respect to building redundant indexes on materialized views as well as to coherency, updates, invalidation, re-computation, incremental updates, etc.; probably the simplest policy (certainly from the perspectives of the application developer and the end user) is to treat materializations of view results similar to indexes, meaning instant updates within the original transaction, or even simply to index views in addition to tables, supporting both non-clustered and clustered indexes (the latter contain all columns in the table or the view). By doing so, the semantics of views is unchanged whether they are materialized and indexed or not; thus, indexing a view is (correctly) purely an issue of physical database design and performance, not one

of logical database design and correctness.

Caching recent queries: Another exciting technology that is only starting to take off is closely related to materialized views: caching the results of recent actual queries in order to exploit them in future queries. Clearly, some OLAP tools achieve amazing performance using this technique today already, in particular if it is used both on the server and on the desktop, i.e., both for local and distributed queries. The issues with respect to indexing as well as invalidation or incremental maintenance are the same as for materialized (or indexed) views. The hard part, as with materialized views, is the policy deciding which query result to keep and to maintain for how long.

Performance metrics: The goal of indexing and result caching is to improve system performance. However, there are different definitions of performance. An excellent illustration can be found in the arguments going back and forth whether the overall performance in the TPC-D benchmark should be based on the arithmetic or geometric mean of the individual query times. For example, is it better to improve one query from 10 seconds to 1 second (9 seconds difference or factor 10) or to improve another query from 10 minutes to 5 minutes (300 seconds difference but only a factor of 2)? A more important performance goal than high performance, however, is consistent and predictable performance. The vast majority of database servers in production is running on CPUs much slower than the fastest available CPUs, often by a factor 2. Given that, clearly top performance is not the top priority; if it were, CPUs would be replaced much more rapidly. In other words, predictability versus risk is a more important dimension than fast versus slow, within the limits of common sense.

Predictable performance: Predictable performance has several aspects. One of them is that the same request or very similar requests always take the same time. For example, once a user has seen a certain query complete in 2 minutes, it is a source of great annoyance (and support calls) if that query usually takes 5 minutes. One lesson from this observation is that parallel query processing might be a very bad idea indeed, because it may well lead to unfulfilled expectations. Another aspect of predictability is that a "regular" person must be able to predict the performance. In other words, fancy adaptive techniques that make a chosen dynamic query plan hard to predict or to understand are a disservice to developers and users, unless these adaptive techniques work so well that developers and users never have much interest in looking at query plans, just as little as they have nowadays in understanding the inner workings of virtual memory.

Index tuning: Several database vendors offer (or include in their main database products) tools for index tuning, including materialized views and their indexes. These tools perform a complex analysis of the workload and can create indexes as well as drop indexes due to update costs or to space constraints. Clearly, the current versions of these tools have left room for improvement, in particular in the expense of the analysis and in the immediacy, i.e., how fast they adapt a physical database design to an initial workload and to changes in the workload. Moreover, most of these tools are based on the assumptions that database tuning is an activity that is separate from processing requests, requires a trained individual such as a DBA, and takes a substantial amount of system effort, e.g., for analyzing query plans for various proposed index configurations. Due to their computational overhead, they typically do not lend themselves to complete automation.

Indexing tuning heuristics: It turns out, however, that fairly simple indexing schemes often work amazingly well. For example, clustered indexes on primary keys and non-clustered indexes on foreign keys are a pretty good starting point. Since they rarely change and are important in many applications, non-clustered indexes on all columns with "date" or "time" data types often help. Finally, columns that appear in "=" predicates probably benefit from indexes, which actually would include indexes on primary and foreign keys if not already indexed. Indexes on "in" predicates as well as selective "between" predicates typically are worthwhile, too. To refine these rules, one can add the primary key and all foreign keys to all indexes, just to make index-to-index navigation

faster. Finally, one can avoid any non-clustered indexes on very small tables, say less than a page, and avoid using more space for indexes than for the main data.

Queries building indexes: With these simple indexing rules, many query sets, e.g., TPC-D, work pretty well (definitely for query processors supporting index intersection) – certainly within the same factor of 2 many people are willing to forgo by not keeping up with the latest hardware. An interesting question is whether these indexes can be created quickly and quietly. For example, if an equality predicate is found that is not supported by an existing index, how can one be built with minimal overhead? A separate index creation has to tread lightly, e.g., it must not "hog" the CPU or the disk, and it must not prevent concurrent user activity. In other words, online index creation is just as important a feature for fully automated "low-end" data management systems as for carefully monitored high-end 24x7 systems that strive for "five nines" or better (99.999% availability, or 5 minutes down time per year, including planned down time). If, on the other hand, an equality predicate without suitable index results in a full scan, can we design adaptive plans that scan, sort, merge join, and leave behind the appropriate index for the next execution? Can that still work even if there are concurrent updates that the index must reflect but the query result must not? Can the new index be fully exploited when the same query is run again, without re-compilation and re-optimization, by means of a dynamic query plan that exploits an index if it exists or creates one if it doesn't?

Heuristics for indexing views: While fairly obvious and immediate rules might give satisfactory results for indexes on tables, this is not the case for materialized (indexed) views. There are some very effective algorithms and heuristics for pre-aggregations, both in the literature and in products, but many queries used in data analysis, in particular if dimensions and hierarchies are used, can benefit not only from pre-aggregations but also from joins, outer joins, nested queries, semi joins, set operations, etc. Crystallizing simple and effective heuristics and their integration into query processing could substantially benefit the use of database systems, in particular relational database systems, in data analysis and data mining.

Research for orders of magnitude: Finally, a brief word of caution. A performance- or throughput-oriented software technique that results in a performance improvement measured as a percentage might very well be important for a vendor under pressure to win today's benchmark, but it isn't a worthy research goal or result. An improvement measured by a small factor (say factor 3) is laudable and useful, but not a breakthrough – improvements in hardware technology will give us the same improvement (fairly predictably!) in just one or two years, the time it takes to publish a journal paper about the new software technique or to ship it in a product. A typical example for small-factor performance improvement is research into algorithms exploiting CPU caches, typically by adapting techniques proven for disks and disk pages, e.g., B-trees, external sorting, and horizontal and vertical partitioning. In order to be truly a breakthrough, a performance improvement has to be measured in orders of magnitude. Materialized views are one such technique. Dynamic query plans, on the other hand, so far have not achieved this level of success on a broad scale. Is it possible to achieve it? Can we achieve consistent and predictable orders-of-magnitude performance improvements for relational and post-relational database systems by combining dynamic query plans with on-the-fly indexing and materialized views?