Chapter 7: Query Optimization

Introduced by Joe Hellerstein

Selected Readings:

Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. *ICDE*, 1993.

Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. *SIGMOD*, 2000.

Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, Miso Cilimdzic. Robust Query Processing Through Progressive Optimization. *SIGMOD*, 2004.

Query optimization is one of the signature components of database technology—the bridge that connects declarative languages to efficient execution. Query optimizers have a reputation as one of the hardest parts of a DBMS to implement well, so it's no surprise they remain a clear differentiator for mature commercial DBMSs. The best of the open-source relational database optimizers are limited by comparison, and some have relatively naive optimizers that only work for the simplest of queries.

It's important to remember that no query optimizer is truly producing "optimal" plans. First, they all use estimation techniques to guess at real plan costs, and it's well known that errors in these estimation techniques can balloon—in some circumstances being as bad as random guesses [7]. Second, optimizers use heuristics to limit the search space of plans they choose, since the problem is NP-hard [6]. One assumption that's gotten significant attention recently is the traditional use of 2-table join operators; this has been shown to be theoretically inferior to new multi-way join algorithms in certain cases [12].

Despite these caveats, relational query optimization has proven successful, and has enabled relational database systems to serve a wide range of bread-and-butter use cases fairly well in practice. Database vendors have invested many years into getting their optimizers to perform reliably on a range of use cases. Users have learned to live with limitations on the number of joins. Optimizers still, for the most part, make declarative SQL queries a far better choice than imperative code for most uses.

In addition to being hard to build and tune, serious query optimizers also have a tendency to grow increasingly complex over time as they evolve to handle richer workloads and more corner cases. The research literature on database query optimization is practically a field unto itself, full of technical details—many of which have been discussed in the literature by researchers at mature vendors like IBM and Microsoft who work closely with product groups. For this book, we focus on the big picture: the main architectures that have been considered for query optimization and how have they been reevaluated over time.

Volcano/Cascades

We begin with the state of the art. There are two reference architectures for query optimization from the early days of database research that cover most of the serious optimizer implementations today. The first is Selinger et al.'s System R optimizer described in Chapter 3. System R's optimizer is textbook material, implemented in many commercial systems; every database researcher is expected to understand it in detail. The second is the architecture that Goetz Graefe and his collaborators refined across a series of research projects: Exodus, Volcano, and Cascades. Graefe's work is not covered as frequently in the research literature or the textbooks as the System R work, but it is widely used in practice, notably in Microsoft SQL Server, but purportedly in a number of other commercial systems as well. Graefe's papers on the topic have something of an insider's flavor—targeted for people who know and care about implementing query optimizers. We chose the Volcano paper for this book as the most approachable representative of the work, but aficionados should also read the Cascades paper [5]—not only does it raise and address a number of detailed deficiencies of Volcano, but it's the latest (and hence standard) reference for the approach. Recently, two open-source Cascades-style optimizers have emerged: Greenplum's Orca optimizer is now part of the Greenplum open source, and Apache Calcite is an optimizer that can be used with multiple backend query executors and languages, including LINQ.

Graefe's optimizer architecture is notable for two main reasons. First, it was expressly designed to be extensible. Volcano deserves credit for being quite forward-looking—long predating MapReduce and the big data stacks—in exploring the idea that dataflow could be useful for a wide range of data-intensive applications. As a result, the Graefe optimizers are not just for compiling SQL into a plan of dataflow iterators. They can be parameterized for other in-

put languages and execution targets; this is a highly relevant topic in recent years with the rise of specialized data models and languages on the one hand (see Chapter 2 and 9), and specialized execution engines on the other (Chapter 5). The second innovation in these optimizers was the use of a top-down or goal-oriented search strategy for finding the cheapest plan in the space of possible plans. This design choice is connected to the extensibility API in Graefe's designs, but that is not intrinsic: the Starburst system showed how to do extensibility for Selinger's bottom-up algorithm [9]. This "top-down" vs "bottom-up" debate for query optimization has advocates on both sides, but no clear winner; a similar top-down/bottom-up debate came out to be more or less a tie in the recursive query processing literature as well [13]. Aficionados will be interested to note that these two bodies of literature–recursive query processing and query optimizer search–were connected directly in the Evita Raced optimizer, which implemented both top-down and bottom-up optimizer search by using recursive queries as the language for implementing an optimizer [1].

Adaptive Query Processing

By the late 1990's, a handful of trends suggested that the overall architecture of query optimization deserved a significant rethink. These trends included:

- Continuous queries over streaming data.
- Interactive approaches to data exploration like Online Aggregation.
- Queries over data sources that are outside the DBMS and do not provide reliable statistics or performance.
- Unpredictable and dynamic execution environments, including elastic and multitenant settings and widely distributed systems like sensor networks.
- Opaque data and user-defined functions in queries, where statistics can only be estimated by observing behavior.

In addition, there was ongoing practical concern about the theoretical fact that plan cost estimation was often erratic for multi-operator queries [7]. As a result of these trends, interest emerged in adaptive techniques for processing queries, where execution plans could change mid-query. We present two complementary points in the design space for adaptive query processing; there is a long survey with a more comprehensive overview [4].

Eddies

The work on eddies, represented by our second paper, pushed hard on the issue of adaptivity: if query "re-planning" has to occur mid-execution, why not remove the architectural distinction between planning and execution entirely? In the eddies approach, the optimizer is encapsulated as a dataflow operator that is itself interposed along other dataflow edges. It can monitor the rates of dataflow along those edges, so it has dynamic knowledge of their behavior, with whatever history it cares to record. With that ongoing flow of information, it can dynamically control the rest of the aspects of query planning via dataflow routing: the order of commutative operators is determined by the order tuples are routed through operators (the focus of the first eddies paper that we include here) the choice of physical operators (e.g. join algorithms, index selection) is determined by routing tuples among multiple alternative, potentially redundant physical operators in the flow [14, 3] the scheduling of operators is determined by buffering inputs and deciding which output to deliver to next [15]. As an extension, multiple queries can be scheduled by interposing on their flows and sharing common operators [10]. Eddies intercept the ongoing dataflow of query operators while they are in flight, pipelining data from their inputs to their output. For this reason it's important that eddy routing be implemented efficiently; Deshpande developed implementation enhancements along these lines [2]. The advantage of this pipelined approach is that eddies can adaptively change strategies in the middle of executing a pipelined operator like a join, which is useful if a query operator is either very long-lived (as in a streaming system) or a very poor choice that should be abandoned long before it runs to completion. Interestingly, the original Ingres optimizer also had the ability to make certain query optimization decisions on a per-tuple basis [18].

Progressive Optimization

The third paper in this section from IBM represents a much more evolutionary approach, which extends a System R style optimizer with adaptivity features; this general technique was pioneered by Kabra and DeWitt [8] but receives a more complete treatment here. Where eddies focused on intra-operator reoptimization (while data is "in motion"), this work focuses on inter-operator reoptimization (when data is "at rest"). Some of the traditional relational operators including sorting and most hash-joins are blocking: they consume their entire input before producing any output. This presents an opportunity after input is consumed to compare observed statistics to optimizer predictions, and reoptimize the "remainder" of the query plan using traditional query optimization technique. The downside of this approach is that it does no reoptimization while operators are consuming their inputs, making it inappropriate for continuous queries over streams, for pipelining operators like symmetric hash join [17] or for long-running relational queries that have

poorly-chosen operators in the initial parts of the plan – e.g. when data is being accessed from data sources outside the DBMS that do not provide useful statistics [11, 16].

It's worth noting that these two architectures for adaptivity could in principle coexist: an eddy is "just" a dataflow operator, meaning that a traditional optimizer can generate a query plan with an eddy connecting a set of streaming operators, and also do reoptimization at blocking points in the dataflow in the manner of our third paper.

Discussion

This brings us to a discussion of current trends in dataflow architectures, especially in the open source big data stack. Google MapReduce set back by a decade the conversation about adaptivity of data in motion, by baking blocking operators into the execution model as a fault-tolerance mechanism. It was nearly impossible to have a reasoned conversation about optimizing dataflow pipelines in the mid-to-late 2000's because it was inconsistent with the Google/Hadoop fault tolerance model. In the last few years the discussion about execution frameworks for big data has suddenly opened up wide, with a quickly-growing variety of dataflow and query systems being deployed that have more similarities than differences (Tenzing, F1, Dremel, DryadLINQ, Naiad, Spark, Impala, Tez, Drill, Flink, etc.) Note that all of the motivating issues for adaptive optimization listed above are very topical in today's big data discussion, but not well treated.

More generally, I would say that the "big data" community in both research and open source has been far too slow to focus on query optimization, to the detriment of both the current systems and the query optimization field. To begin with, the "hand-planned" MapReduce programming model remained a topic of conversation for far longer than it should have. It took a long time for the Hadoop and systems research communities to accept that a declarative language like SQL or LINQ is a good general-purpose interface, even while maintaining low-level MapReduce-style dataflow programming as a special-case "fast path". More puzzling is the fact that even when the community started building SQL interfaces like Hive, query optimization remained a little-discussed and poorly-implemented topic. Maybe it's because query optimizers are harder to build well than query executors. Or maybe it was fallout from the historical quality divide between commercial and open source databases. MySQL was the open source de facto reference for "database technology" for the preceding decade, with a naive heuristic optimizer. Perhaps as a result, many (most?) open source big data developers didn't understand—or trust—query optimizer technology.

In any case, this tide is turning in the big data community. Declarative queries have returned as the primary interface to big data, and there are efforts underway in essentially all the projects to start building at least a 1980's-era optimizer. Given the list of issues I mention above, I'm confident we'll also see more innovative query optimization approaches deployed in new systems over the coming years.

References

[1] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. *Proceedings of the VLDB Endowment*, 1(1):1153–1165, 2008.

[2] A. Deshpande. An initial study of overheads of eddies. *ACM SIGMOD Record*, 33(1):44–49, 2004.

[3] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, 2004.

[4] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[5] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[6] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS)*, 9(3):482–502, 1984.

[7] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD*, 1991.

[8] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[9] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *SIGMOD*, 1988.

[10] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[11] J. Melton, J. E. Michels, V. Josifovski, K. Kulkarni, and P. Schwarz. Sql/med: a status report. *ACM SIGMOD Record*, 31(3):81–89, 2002.

[12] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms:[extended abstract]. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

[13] R. Ramakrishnan and S. Sudarshan. Top-down vs. bottom-up revisited. In *Proceedings of the International Logic Programming Symposium*, pages 321–336, 1991.

[14] V. Raman, A. Deshpande, and J. M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*. IEEE, 2003.

[15] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *SIGMOD*, pages 275–286. ACM, 2002.

[16] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. *ACM SIGMOD Record*, 27(2):130–141, 1998.

[17] A. N. Wilschut and P. M. Apers. Dataflow query execution in a parallel main-memory environment. In *Parallel and Distributed Information Systems, 1991., Proceedings of the First International Conference on*, pages 68–77. IEEE, 1991.

[18] E. Wong and K. Youssefi. Decompositiona strategy for query processing. *ACM Transactions on Database Systems (TODS)*, 1(3):223–241, 1976.