ACM SIGMOD Special Interest Group on Management Of Data Blog
Association For Computing Machinery



# Is Query Optimization a "Solved" Problem?

April 10, 2014 Guy Lohman 8 comments

*Is Query Optimization a "solved" problem? If not, are we attacking the "right" problems? How should we identify the "right" problems to solve?*

I asked these same questions almost exactly 25 years ago, in an extended abstract for a Workshop on Database Query Optimization that was organized by the then-Professor Goetz Graefe at the Oregon Graduate Center [Grae 89a]. Remarkably and quite regrettably, most of the issues and critical unsolved problems I identified in that brief rant remain true today. Researchers continue to attack the wrong problems, IMHO: ***they attack the ones that they can***, i.e., that they have ideas for, rather than the ones that ***they should***, i.e., that are critical to successfully modeling the true cost of plans and choosing a good one. Perhaps more importantly, that will avoid choosing a disastrous plan! At the risk of repeating myself, I'd like to re-visit these issues, because I'm disappointed that few in the research community have taken up my earlier challenge.

---

The root of all evil, the Achilles Heel of query optimization, is the estimation of the size of intermediate results, known as cardinalities. Everything in cost estimation depends upon how many rows will be processed, so the entire cost model is predicated upon the cardinality model. In my experience, the cost model may introduce errors of at most 30% for a given cardinality, but the cardinality model can quite easily introduce errors of ***many orders of magnitude***! I'll give a real-world example in a moment. With such errors, the wonder isn't "Why did the optimizer pick a bad plan?" Rather, the wonder is "Why would the optimizer ever pick a decent plan?"

"Well," you say, "we've seen lots of improvements in histograms, wavelets, and other statistics since 1989. Surely we do better now." There's been no shortage of such papers, it's true, but the wealth of such papers precisely illustrates my point. Developing new histograms that improve selectivity estimation for individual local predicates of the form "Age BETWEEN 47 AND 63" by a few percent doesn't really matter, when other, much larger errors that are introduced elsewhere in cardinality estimation dwarf those minor improvements. It's simple engineering, folks. If I have to review one more such paper on improved histograms for local predicates, I'll scream (and reject it)! It ***just doesn't matter***! What we have now is good enough.

What still introduces the most error in cardinality estimation is (*a*) host variables and parameter markers, (*b*) the selectivity of join predicates, and, even more significantly, (*c*) how we combine selectivities to estimate the cardinality. Amazingly, these three topics also have enjoyed the least research attention, or at least the fewest number of papers attempting to solve them, unless I've missed some major contributions lately. I'll visit each of these topics in turn, describing the fundamental causes of errors, and why those errors can easily reach disastrous proportions, illustrated by war stories from real customer situations.

**Host variables and parameter markers**
Host variables, parameter markers, and special registers occur in SQL queries because applications on top of the DBMS, not humans, invoke most queries, unlike in academic papers. Such applications typically get the constants used in predicates from a "fill in the blank" field on a web page, for example. The SQL predicate then looks like "Age BETWEEN :hv1 AND :hv2". At compile time, the optimizer has no clue what :hv1 and :hv2 are, so it

**Recent Posts**
- Exploratory search: New name for an old hat?
- Is Query Optimization a "Solved" Problem?

**Recent Comments**
- James Parker on Is Query Optimization a "Solved" Problem?
- Ruslan Fomkin on Is Query Optimization a "Solved" Problem?
- Radim Baca on Is Query Optimization a "Solved" Problem?
- Stephen Dillon on Is Query Optimization a "Solved" Problem?
- Alexandr Savinov on Is Query Optimization a "Solved" Problem?

**Archives**
- June 2014
- April 2014
- February 2014
- December 2013
- October 2013
- September 2013
- June 2013
- May 2013
- March 2013
- December 2012
- November 2012
- August 2012
- July 2012
- May 2012
- April 2012
- February 2012

**Search Posts**

cannot look them up in those wonderful histograms that we all have polished. Instead, it must make a wild guess on the average or likely values, which could be off significantly from one execution to the next, or even due to skew. A war story illustrates this point.

One of our major ISVs retrofitted a table with a field that identified which subsystem each row came from. It had 6 distinct values, but 99.99% of the rows had the value of the founding subsystem, i.e., when there was only one. A predicate on this subsystem column was added to every query, with the value being passed as a host variable. Not knowing that value a priori, DB2's optimizer used the average value of 1/|distinct values| = 0.167, though that predicate's true selectivity was usually 0.9999 (not selective at all) and once in a blue moon was 0.0001 (extremely selective).

There has been some work on this so-called Parametric Query Optimization (PQO), though it's sometimes attacking the problem of other parameters unknown at compilation time (e.g. the number of buffer pages available) or limited to discrete values [Ioan 97]. One of my favorites is a fascinating empirical study by Reddy and Haritsa [Redd 05] of plan spaces for several commercial query optimizers as the selectivity of multiple local predicates are varied. It demonstrated (quite colorfully!) that regions in which a particular plan is optimal may not be convex and may even be disconnected! Graefe suggested keeping a different plan for each possible value of each host variable [Grae 89b], but with multiple host variables and a large number of possible values, Graefe's scheme quickly gets impractical to optimize, store, and decide at run-time among the large cross-product of possible plans, without grouping them into regions having the same plan [Stoy 08].

Version 5 of DB2 for OS/390 (shipped June 1997) developed a practical approach to force re-optimization for host variables, parameter markers, and special registers by adding new SQL bind options REOPT(ALWAYS) and REOPT(ONCE). The latter re-optimizes the first time that the statement is executed with actual values for the parameters, and assumes that these values will be "typical", whereas the former forces re-optimization each time the statement is run. Later, a REOPT(AUTO) option was added to autonomically determine if re-optimization is needed, based upon the change in the estimated filter factors from the last re-optimization's plan.

**Selectivity of join predicates**

The paucity of innovation in calculating join predicate selectivities is truly astounding. Most extant systems still use the techniques pioneered by System R for computing the selectivity of a join as the inverse of the maximum of the two join-column cardinalities [Seli 79], which essentially assumes that the domain of one column is a subset of the other. While this assumption is valid for key domains having referential integrity constraints, in general the overlap of the two sets may vary greatly depending upon the semantics of the joined domains. Furthermore, the common practice of pre-populating a dimension table such as "Date" with 100 years of dates into the future can incorrectly bias this calculation when the fact table initially has only a few months of data. Statistics on join indexes [Vald 87] would give much more precise selectivities for join predicates, if we were willing to suffer the added costs of maintenance and lock contention of updates to these join indexes. However, join indexes would not solve the **intersection problem** typical of star schemas.

For example, suppose a fact table of Transactions has dimensions for the Products, Stores, and Dates of each transaction. Though current methods provide accurate selectivity estimates for predicates local to each dimension, e.g., ProductName = 'Dockers' and StoreName = 'San Jose' and Date = '23-Feb-2013', it is impossible to determine the effect of the ***intersection*** of these predicates on the fact table. Perhaps the San Jose store had a loss-leader sale on Dockers that day that expired the next day, and a similar sale on some other product the next day, so that the individual selectivities for each day, store, and product appear identical, but the actual sales of Dockers on the two days would be significantly different. It is the **interaction** of these predicates, through join predicates in this case, that research to date doesn't address. This leads naturally to the final and most challenging problem in our trifecta.

**Correlation of columns**

With few exceptions ([Chri 83], [Vand 86]), query optimizers since [Seli 79] have modeled selectivities as probabilities that a predicate on any given row will be satisfied, then multiplied these individual selectivities together. Effectively, this assumes that Prob{ColX = Value1, ColY = Value2} = Prob{ColX = Value1} * Prob{ColY = Value2}, i.e., that the two predicates are probabilistically independent, if you recall your college probability. Fortunately for the database industry, this assumption is often valid. ***However, occasionally it is not***.

My favorite example, which occurred in a customer database, is Make = 'Honda' and Model = 'Accord'. To simplify somewhat, suppose there are 10 Makes and 100 Models. Then the independence (and uniformity) assumption gives us a selectivity of 1/10 * 1/100 = 0.001. But since only Honda makes Accords, by trademark law, the real selectivity is 0.01. So we will *under-estimate the cardinality by an order of magnitude*. Such optimistic errors are much worse than pessimistic over-estimation errors, because they cause the optimizer to think that certain operations will be cheaper than they really are, causing nasty surprises at run time. The only way to avoid such errors is for the database administrator to be aware of the semantic relationship (a functional dependency, in this case) between those two columns and its consequences, and to collect ***column group statistics***, as DB2 and other database products now allow.

To identify these landmines in the schema automatically, Stillger et al. [Stil 01] developed the LEarning Optimizer (LEO), which opportunistically and automatically compared run-time actual cardinalities to optimizer estimates, to identify column combinations exhibiting such correlation errors. Ilyas et al. [Ilya 04] attacked the problem more pro-actively in CORDS (CORrelation Detection by Sampling), searching somewhat exhaustively for such correlations between any two columns in samples from the data before running any queries. And Markl and colleagues [Mark 05], [Mark 07] have made ground-breaking advances on a consistent way to combine the selectivities of conjuncts in partial results.

All great progress on this problem, but ***none yet solves the problem of redundant predicates*** that can be inadvertently introduced by the query writer who typically believes that "more is better", that providing more predicates helps the DBMS do its job better – it's American as Apple Pie! Let me illustrate with one of my favorite war stories.

At a meeting of the International DB2 User's Group, a chief database administrator for a major U.S. insurance company whom I'd helped with occasional bad plans asked me to conduct a class on-site. I suggested it include an exercise on a real problem, unrehearsed. After my class, she obliged me by presenting two 1-inch stacks of paper, each detailing the EXPLAIN of a plan for a query. I feared I was going to embarrass myself and fail miserably under the gun. The queries differed in only one predicate, she said, but the original ran in seconds whereas the one with the extra predicate took over an hour.

I instinctively examined first the cardinality estimates for the results of the two, and the slower one had a cardinality estimate 7 orders of magnitude less than the fast one. When asked what column the extra predicate was on, my host explained that it was a composite key constructed of the first four letters of the policy-holder's last name, the first and middle initials, the zip code, and last four digits of his/her Social Security Number. Did the original query have predicates on all those columns? Of course! And how many rows were there in the table? Ten million. Bingo! I explained that that predicate was completely redundant of the others, and its selectivity, $1/10^7$, when multiplied by the others, underestimated the cardinality by 7 orders of magnitude, wreaking havoc with the plan. It took me maybe 5 minutes to figure this out, and I was immediately dubbed a "genius", but it really was straightforward: the added predicate might help the run-time, especially if they had an index on that column, but it totally threw off the optimizer, which couldn't possibly have detected that redundancy without LEO or CORDS.

**So c'mon, folks, let's attack problems that really matter, those that account for optimizer disasters, and stop polishing the round ball.**

*Disclaimer: The postings on this site are my own and don't necessarily represent IBM's positions, strategies or opinions.*

References

[Chri 83] S. Christodoulakis, "Estimating record selectivities", Info. Systems 8,2 (1983) pp. 105-115.

[Grae 89a] G. Graefe, editor, Workshop on Database Query Optimization, CSE Technical Report 89-005, Oregon Graduate Center, Portland, OR, 30 May 1989.

[Grae 89b] G. Graefe, "The stability of query evaluation plans and dynamic query evaluation plans", Proceedings of ACM-SIGMOD, Portland, OR (1989).

[Ioan 97] Y. Ioannidis et al., "Parametric query optimization", VLDB Journal, 6(2), pp. 132 – 151, 1997.

[Ilya 04] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, A. Aboulnaga: CORDS: Automatic Discovery of Correlations

and Soft Functional Dependencies. SIGMOD Conference 2004: 647-658.

[Mark 05] V. Markl, N. Megiddo, M. Kutsch, T. Minh Tran, P. J. Haas, U. Srivastava: Consistently Estimating the Selectivity of Conjuncts of Predicates. VLDB 2005: 373-384.

[Mark 07] V. Markl, P. J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, T. Minh Tran: Consistent selectivity estimation via maximum entropy. VLDB J. 16(1): 55-76 (2007)

[Redd 05] N. Reddy and J. R. Haritsa. “Analyzing plan diagrams of database query optimizers”, VLDB, 2005.

[Seli 79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, “Access path selection in a Relational Database Management System”, Procs. Of ACM-SIGMOD (1979), pp. 23-34.

[Stil 01] M. Stillger, G. M. Lohman, V. Markl, M. Kandil: LEO – DB2′s LEarning Optimizer. VLDB 2001: 19-28.

[Stoy 08] J. Stoyanovich, K. A. Ross, J. Rao, W. Fan, V. Markl, G. Lohman: ReoptSMART:A Learning Query Plan Cache. Technical report.

[Vald 87] P. Valuriez, “Join indices”, ACM Trans. on Database Systems 12, 2 (June 1987), pp. 218-246.

[Vand 86] B.T. Vander Zanden, H.M. Taylor, and D. Bitton, “Estimating block accesses when attributes are correlated, Procs. of the 12th Intl. Conf. on Very Large Data Bases (Kyoto, Sept. 1986), pp. 119-127.

**Blogger’s Profile:**
*Dr. Guy M. Lohman is Manager of Disruptive Information Management Architectures in the Advanced Information Management Department at IBM Research Division’s Almaden Research Center in San Jose, California, where he has worked for 32 years. He currently manages the Blink research project, which contributed BLU Acceleration to DB2 for Linux, UNIX, and Windows (LUW) 10.5 (GA’d 2013) and the query engine of the IBM Smart Analytics Optimizer for DB2 for z/OS V1.1 and the Informix Warehouse Accelerator products (2007-2010). Dr. Lohman was the architect of the Query Optimizer of DB2 LUW and was responsible for its development from 1992 to 1997 (versions 2 – 5), as well as the invention and prototyping of Visual Explain, efficient sampling, the DB2 Index Advisor, and optimization of XQuery queries in DB2. Dr. Lohman was elected to the IBM Academy of Technology in 2002 and made an IBM Master Inventor in 2011. He was the General Chair for ACM’s 2013 Symposium on Cloud Computing and is the General Co-Chair of the 2015 IEEE International Conference on Data Engineering (ICDE). Previously, he was on the editorial boards of the “Very Large Data Bases Journal” and “Distributed and Parallel Databases”. He is the author of over 75 papers in the refereed academic literature, and has been awarded 39 U.S. patents. His current research interests involve disruptive machine architectures for Business Intelligence, advanced data analytics, query optimization, self-managing database systems, information management appliances, database compression, and autonomic problem determination.*



Databases

## 8 comments

**James Parker**
April 17, 2014 at 9:43 pm

I found this an interesting and useful paper, given that this is not my area of expertise (main-memory DBMS). However, I wonder to what extent query optimization ought to also consider the working set of rows already in memory when determining a query plan as well. Such an approach might even improve performance when paging of the underlying in-VM rowss is possible, by keeping them in the working set(s) of the DBMS process(es).

**Stephen Dillon**
April 20, 2014 at 2:48 pm

James,

I too was contemplating the role of an in-memory DBMS (IMDBS) and these query optimization points. Just how much of an impact will they contribute to a system such as VoltDB? This is actually something I’ve had discussions with other architects about to address their own proprietary main memory DBMS.

My initial thoughts are that a commercial IMDBS may be impacted less, than a traditional RDBMS, by such optimizations when you consider the significant impact main-memory operations already provide queries. By commercial, I really mean one that is professionally developed and properly addresses latching, locking, durability, et al. That does not mean proper or better optimization is not needed for an IMDBS. Nobody has proven these optimizations yet with an IMDBS to my knowledge. I just believe that as compared to a traditional RDBMS, an IMDBS may not benefit as much. I also believe that these modern main-memory DBs such as memSQL or VoltDB allow “some” bad code to hide under the guise of fast execution plans that have benefited from zero disk reads. Now for someone’s proprietary IMDBS well these optimizations could certainly provide more benefits considering how less efficient it is to a commercial product.

**Ruslan Fomkin**
April 25, 2014 at 7:54 am

I haven’t tested, but in my opinion the query optimization has similar effect on in-memory and disk-based databases. I see two reason for this:
(1) Data in disk-based database are cached in main-memory. Thus no disk access in both DBMS types. The main difference is cardinality units: pages vs cache lines.
(2) The main bottleneck for query execution is memory wall nowadays. Thus suboptimal query plans in IMDBMS can still perform by an order of magnitude worse then optimal one.

**James Parker**
April 26, 2014 at 7:17 am

Stephen,

I think you misunderstood me; I was not thinking directly of main memory (AKA in-memory) DBMSs, but rather about the memory hierarchy (memory vs. “disk”) and where specific data resides at the time of a query. In a traditional DBMS, some data is known to be in memory, including which rows from which tables — a query optimizer might favor joins using tables with most or all rows in memory over those which are not, as a part of query optimization. Of course the effort must take into account paging, if it may occur; however paging can often be managed programmatically, e.g., by “locking” pages in memory.

The similar issue in a main memory DBMS would be where in the CPU cache hierarchy data resides; this is generally of much lower utility, since machine architectures generally do not expose ways to manage these caches programmatically (although I have speculated as to how hardware and OS architects might provide useful features toward this end, and would almost certainly need to be combined with processor scheduling and interrupt handling).

**spaghettidba**
April 18, 2014 at 10:19 am

Thanks for the interesting writing.
The recently released SQL Server 2014 incorporates a new cardinality estimator that (partially) addresses the predicate independance issue. The new algorithm used is know as

“exponential backoff” and it helps reducing the estimation error with multiple predicates. You can read about it here: http://www.sqlperformance.com/2014/01/sql-plan/cardinality-estimation-for-multiple-predicates

**Lothar Flatz**

April 18, 2014 at 5:54 pm

After 15 years of tuning in the field I have come accross all of these issues.
This I have my own examples i.e. Correlation of columns: mobile phone service provider. Primary key of table units consist of customer number and phone number. Selectivity of join predicates: Road assistence search for service centers. Selectivity differs largely depending on country of accident. Wrote a paper on it. Hope it gets accepted. 😃

**Alexandr Savinov**

April 18, 2014 at 6:24 pm

Thanks for the interesting post and relevant works. One problem in query optimization I am currently dealing with is optimizing column operations for analytical queries. It is highly important for a novel system for analytical data integration (ConceptMix – http://conceptoriented.com) where all operations with data are described in terms of an algebra of functions (a function represents a column). Essentially, all manipulations are reduced to manipulating functions (and inverse functions) and the problem is to translate queries from the concept-oriented expression language to these functional operations.

**Radim Baca**

April 22, 2014 at 1:07 pm

Thank you for an inspirational article. I have a question regarding the “redundant predicates problem”: can the CORDS method solve the problem? From my perspective this problem is just a result of high correlation of attributes, which is addressed by CORDS. Thanks