AutoAdmin Project at Microsoft Research: Lessons Learned

Nicolas Bruno Surajit Chaudhuri Arnd Christian König Vivek Narasayya
Ravi Ramamurthy Manoj Syamala

Microsoft Corporation

{nicolasb,surajitc,chrisko,viveknar,ravirama,manojsy}@microsoft.com

1 Introduction

The AutoAdmin project was started at Microsoft Research in 1996. The goal of AutoAdmin is to develop technology that makes database systems more self-tuning and self-managing. Initially, we focused mainly on the physical database design problem. In subsequent years we broadened our focus and studied a number of other problems: database monitoring, execution feedback for query optimization, flexible hinting mechanisms for query optimizers, query progress estimation and index defragmentation. One discipline we consistently followed in this project was that of implementing our techniques in the database server (in our case this was Microsoft SQL Server). This helped us better understand the applicability and limitations of our ideas and made it possible for our work to have impact on the Microsoft SQL Server product. In this article, we summarize some of the key technical lessons we have learned in the AutoAdmin project. Thus our discussions center around work done in AutoAdmin and does not cover the extensive related work done in other related projects in the database community (see [21] for an overview of work on self-tuning database systems). We focus our discussion on three problems: (1) physical database design, (2) exploiting feedback from execution for query optimization, and (3) query progress estimation. More details of AutoAdmin can be found on the project website [1].

2 Physical Database Design

The physical database design, together with the capabilities of the execution engine and the optimizer, determines how efficiently a query is executed on a DBMS. The importance of physical design is amplified since today's query optimizers have become more sophisticated to cope with complex decision support queries. Thus, the *choice of the right physical design structures*, e.g., indexes, is crucial for efficient query execution over large databases. The key ideas we developed in the AutoAdmin project formed the basis of the Index Tuning Wizard (ITW) that shipped in Microsoft SQL Server 7.0 [3], the first tool of its kind in a commercial DBMS. In subsequent years, we refined these ideas and incorporated the ability to provide integrated recommendations for indexes, materialized views and partitioning. This led to the Database Engine Tuning Advisor (DTA), a full-fledged application that replaced ITW in Microsoft SQL Server 2005 and all subsequent releases [5]. In the rest of this section, we summarize our experience with the AutoAdmin project along two dimensions: (a) Core learnings that we think are essential ingredients for a high quality, scalable physical design tool. (b) Facets of the problem that we have worked on, but which we think have some open, unresolved technical challenges.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2.1 Core Learnings

One of our early decisions was to have a precise model for the problem of physical database design recommendation. In our approach, and subsequently also adopted also by other DBMS engines, we defined the goodness of a physical design *configuration*, i.e., set of physical design structures, with respect to a given query (or workload) as the *optimizer estimated cost* of the query (or workload) for that configuration. The motivation to adopt the above model was to make sure that the physical design recommendations are always "in-sync" with the query optimizer. Two major challenges immediately became apparent as a consequence of this approach. First, there was a need to enable an efficient "what-if" interface for physical design in the query optimizer. Second, developing techniques for searching the space of alternative physical designs in a scalable manner were crucial.

2.1.1 "What-if" API in the Server

The most important server-side enhancement necessary to enable automated physical database design tools is a scalable "what-if" API. A detailed description of this interface appeared in [16] (see Figure 1). The key aspects of the API are: (1) A "Create Hypothetical Index" command that creates metadata entry in the system catalog which defines the index. (2) An extension to the "Create Statistics" command to efficiently generate the statistics that describe the distribution of values of the column(s) of a hypothetical index via the use of sampling [14, 16]. A related requirement was use of an optimization mode that enabled optimizing a query for a selected subset of indexes (hypothetical or actually materialized) and ignoring the presence of other access paths. This too is important as the alternative would have been repeated creation and dropping of what-if indexes, a potentially costly solution. This is achieved via a "Define Configuration" command followed by an invocation of the query optimizer. The importance of this interface went far beyond just *automated* physical design. Once exposed, the interface also made the manual physical design tuning much more efficient. A DBA who wanted to analyze the impact of a physical design change, could do so without disrupting normal database operations.

2.1.2 Reducing the Number of Optimizer Calls

The "what-if" API described above is relatively heavyweight (as large as 100s of milliseconds) since it involves an invocation of the query optimizer. Thus, for a physical design tool that is built on top of this API to be scalable, it becomes crucial to employ techniques for identifying when such optimizer calls can be avoided. We identified several such techniques in [15]. The essence of the idea is to identify certain *atomic* configurations for a query such that once the optimizer calls are made for the atomic configurations, the cost of all other configurations for that query can be derived from the results without requiring any additional optimizer calls. Another direction to reduce the number of optimizer calls was described in [12]. The key idea is to introduce a special query optimization mode, which returns not a single physical plan as usual, but a structure that encodes a compact representation of the optimization search space for that query. This structure then allows efficient generation of plans for arbitrary physical design configurations of that query.

Finally, the approach of selecting a set of candidate indexes *per query* in a cost-based manner is also crucial for scalability of a physical database design tool, e.g., [15]. This approach requires the physical design tool to search for the best configuration for each query, potentially using heuristics. In general, this can result in selection of candidates that are sub-optimal. One idea presented in [8] is to instrument the query optimizer itself to generate the candidate set for each query, thus ensuring that the candidate selection step is also more in-sync with the optimizer than in the earlier approach of [15]. Furthermore, this approach can also significantly speed up the candidate selection step since the best configuration for the query can be returned in a single call to the query optimizer.

2.1.3 Importance of Merging

The initial candidate set results in an optimal (or close-to-optimal) configuration for queries in the workload, but often is either too large to fit in the available storage, or causes updates to slow down significantly. Given

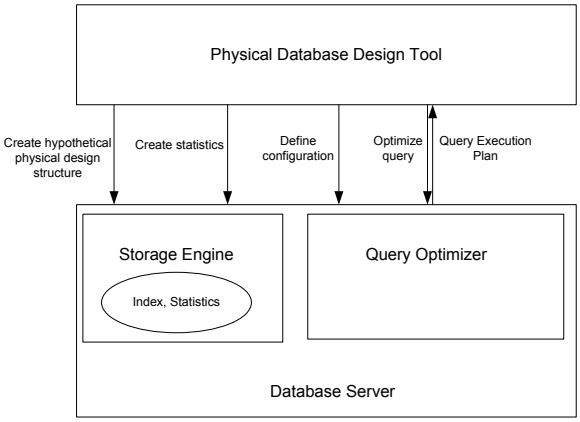

Figure 1: What-if analysis architecture for physical database design

an initial set of candidates for the workload, the merging step augments the set with additional structures that have lower storage and update overhead without sacrificing too much of the querying advantages. The need for merging indexes becomes crucial for decision support queries, where, e.g., different queries are served best by different *covering indexes*, yet the union of these indexes does not fit within the available storage or incurs too high an update cost. Consider a case where the optimal index for query Q_1 is (A, B) and the optimal index for Q_2 is (A, C). A single *merged* index (A, B, C) is sub-optimal for each of the queries but could be optimal for the workload in the case where there is only enough storage to build one index. In general, given a physical design structure S_1 that is a candidate for query Q_1 and a structure S_2 for query Q_2, merging generates a new structure S_{12} with the following properties: (a) Lower storage: $|S_{12}| < |S_1| + |S_2|$. (b) More general: S_{12} can be used to answer both Q_1 and Q_2. Techniques for merging indexes were presented in [17] and merging materialized views in [4].

2.1.4 Integrated Selection of Different Physical Design Structures

Today's DBMSs offer a rich variety of physical design structures to choose from: indexes, partial indexes, compressed indexes, materialized views, range/hash partitioning etc. However, each different physical design structure also introduced unique challenges. For example, two problems were immediately apparent for materialized views. First, the space of materialized views is considerably larger than the space of indexes. We found that we can leverage ideas from frequent itemsets over the workload [4] to prune the candidate sets of tables over which materialized views need to be considered. Second, due to their inherent expressiveness (e.g., they can include a selection condition), materialized views are more susceptible to the problem of *overfitting*, i.e., it is easy to pick a materialized view that is very beneficial to one query but is completely irrelevant for all other queries in the workload. Thus, techniques to identify more widely applicable materialized views are crucial. Our techniques for handling issues arising from other physical design structures such as partitioning and compressed indexes can be found in [6] [25] [26].

The query optimizer can potentially exploit available physical design structures in sophisticated ways. Moreover, each of these physical structures presents different tradeoffs in query cost, update cost, and storage space. It is therefore important that the physical design tool handles these complex tradeoffs comprehensively. In particular, we demonstrated in [4] that simple strategies that staged physical design by first picking indexes followed by materialized views (or vice versa) performed much worse than an integrated solution that considered all physical design structures together during the search step. Indeed, DTA in Microsoft SQL Server adopts such an integrated approach across all physical design structures.

2.1.5 Techniques for Handling Large Workloads

One of the key factors that affects the scalability of physical design tools is the *size* of the workload. DBAs often gather a workload by using server tracing tools such as Microsoft SQL Server Profiler, which log all statements that execute on the server over a representative window of time. Thus, the workloads that are provided to physical database design tuning tools can be large [5], and techniques for *compressing* large workloads become essential. A constraint on such compression is to ensure that tuning the compressed workload gives a recommendation with approximately the same quality (i.e., reduction in cost for the entire workload) as the recommendation obtained by tuning the entire workload. One approach for compressing large workloads in the context of physical design tuning is to exploit the inherent templatization in workloads by partitioning queries on their "signature" [18]. Two queries have the same signature if they are identical in all respects except for the constants referenced in the query (e.g., different instances of a stored procedure). The technique picks a subset from each partition using a clustering based method, where the distance function captures the cost and structural properties of the queries. Adaptations of this technique are used in DTA in Microsoft SQL Server 2005. It is also important to note that, as shown in [5], obvious strategies such as uniformly sampling the workload or tuning only the most expensive queries (e.g., top k by cost) suffer from serious drawbacks, and can lead to poor recommendations. We explored the issue of how to choose the number of queries picked from each partition based on *Statistical Selection* techniques in [28].

2.2 Open Challenges

2.2.1 Top-down vs. Bottom-up search

Broadly, the search strategies explored thus far can be categorized as bottom-up [15] or top-down [8] search, each of which has different merits. The bottom-up strategy begins with the empty (or pre-existing configuration) and adds structures in a greedy manner. This approach can be efficient when available storage is low, since the best configuration is likely to consist of only a few structures. In contrast, the top-down approach begins with a globally optimal configuration which, however, could be infeasible if it exceeds the storage bound. The search strategy then progressively refines the configuration until it meets the storage constraints. The top-down strategy has several key desirable properties [8] and it can be very efficient in cases where the storage bound is large. It remains an interesting open issue as to whether hybrid schemes based on specific input characteristics such as storage bound can improve upon the above strategies.

2.2.2 Handling Richer Constraints

Today's commercial physical design tools attempt to minimize execution costs of input workloads for a given a storage constraint. DTA in Microsoft SQL Server can also allow certain constraints on the final configuration, e.g., a certain clustered index must be included [5]. However, this can still be insufficient to handle many real-world situations where even more sophisticated constraints are desirable. For example, we may want to ensure that no individual query in the workload degrades by more than 10% with respect to the existing configuration. As another example, in order to reduce contention during query processing, the DBA may want to specify that no single column of a table should appear in more than say three indexes. In [9] we introduced a constraint language that is simple yet powerful enough to express many such important scenarios; and show how to efficiently incorporate such constraints into the search algorithm. There are several interesting questions that still need to be addressed, e.g., how these constraints should be exposed to users of physical design tools, and analysis of dependencies or correlations among the constraints.

2.2.3 Exploiting Test Servers for Tuning

The process of tuning a large workload can impose significant overhead on the server being tuned since the physical design tool needs to potentially make many "what-if" API calls to the query optimizer component. In

enterprises there are often test servers in addition to the production server(s). A test server can be used for a variety of purposes including performance tuning, testing changes before they are deployed on the production server, etc. It is therefore important to provide the ability to automatically exploit a test server, if available, to tune a database on a production server *without* having to copy the data or pose "what-if" optimizer calls on the production server. The key observation that enables this functionality is that the query optimizer relies fundamentally on the database metadata and statistics when generating a plan for a query. This observation can be exploited to enable tuning on the test server by migrating only the database "shell" automatically to the test server and then tuning the workload on the test server. The resulting recommendation is guaranteed to be identical to the recommendation that would have been obtained if the tuning had been done entirely on the production server. One important challenge in this approach is that the multi-column statistics required for tuning a workload are typically not known a priori [5]. Thus, there is still some load imposed on production servers for collecting statistics during tuning. Techniques for further alleviating this load on production servers are important.

2.2.4 Different Modes of Tuning

Due to the combinatorial nature of the search space for physical design, the time taken for high quality physical design for a database with a large schema and for a given large workload can be significant. This issue limits the DBA's ability to run the tuning tool frequently to capture impact of changing workload and changing data. One possibility is to develop the ability to determine whether the current physical design configuration is suboptimal a-priori, i.e., *before* running an expensive tuning tool. In [11] we introduced a low-overhead procedure (called *alerter*) that reports lower and upper bounds on the reduction in the optimizer-estimated cost of the workload that would result if the tuning tool were to run then.

However, a qualitatively different approach to physical design recommendation would be one that is fully automated. In a fully automated model, no workload is explicitly provided (unlike the model introduced in Section 2.1) and the design module will continuously track drifts in workload and data characteristics. Furthermore, its decision to make any changes would also take into account cost and the latency involved in implementing the changes to the physical design (e.g., such considerations would discourage building structures with marginal incremental value). In [10] we present algorithms that are always-on and continuously modify the current physical design, reacting to changes in the workload. Two key design considerations were to ensure that our technique adds very little overhead and suggests changes to physical design rather conservatively since no DBAs are in the loop. While our technique takes a significant first step, additional research is needed to realize the vision of fully automated physical database design for database systems.

3 Exploiting Execution Feedback for Query Optimization

An important problem that has been explored by the database community is studying how to effectively exploit statistics obtained during query execution to mitigate the well-known problem of accurate cardinality estimation during query optimization. The feedback information can be integrated with the existing set of database statistics in two basic ways (see Figure 2). In the AutoAdmin project, we introduced the self-tuning histogram [23] approach in which the feedback information is folded back to the existing histograms. We also studied the feedback cache approach (pioneered by the LEO project [31]) in which the feedback information is maintained in a cache. In this section, we briefly summarize some of the key lessons learned in both approaches and outline some interesting avenues for future work.

3.1 Self-Tuning Histograms

Self-tuning histograms, first proposed in [2], use execution feedback to refine their structure in such a way that frequently queried portions of data are represented in more detail compared to data that is infrequently

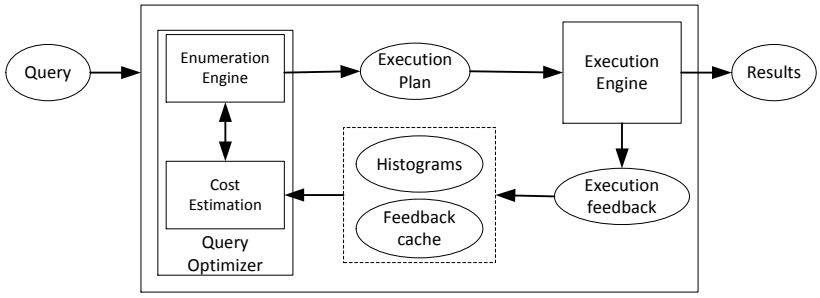

Figure 2: Architecture for incorporating execution feedback into query optimization using (a) histograms or (b) via a feedback cache.

queried. This is especially attractive for multi-dimensional histograms, since the space requirements of traditionally built histograms grow exponentially with the number of dimensions. We learnt that it is possible to design the structure of multi-dimensional histograms to be "execution feedback aware" and by doing so achieve greater accuracy [7] compared to using a traditional grid-like structure.

The main challenge in building self-tuning histograms is to be able to balance the trade-off between accuracy and monitoring overhead needed for obtaining execution feedback. Naturally, tracking errors between estimation and execution feedback at the granularity of every histogram bucket improves accuracy but such monitoring can also raise execution overhead sharply. One approach that shows promise (taken in ISOMER [30]) uses the same multi-dimensional structure introduced in [7] but restricts monitoring to a coarser level, as in [2]. It then uses the maximum entropy principle to reconcile the observed cardinalities and to approximate the data distribution.

3.2 Feedback-Cache Approach

The Feedback-Cache approach was proposed in the LEO project [31]. The main idea is to cache the cardinality values of expressions that are seen during query execution and reuse them during query optimization. In our investigations in the AutoAdmin project, we focused on two key issues: (1) the effectiveness of this approach for the important special case of a query that is repeatedly run. (2) the generality of this approach - in particular, can it be leveraged for parameters beyond cardinalities? The key lessons we learned are summarized below.

Execution feedback raises the possibility of automatically improving the plans for queries that are identical or are similar to queries have been executed in the past. Interestingly, we found that there are natural cases even for the identical query case where the query can continue to be stuck with a bad execution plan despite repeated executions. Intuitively, the reason (see [23] for a detailed example) is that accurate cardinality information for the expression whose inaccurate cardinality estimation is leading to the bad plan may not be available by executing the current bad plan. Thus, "passively" monitoring the plan to obtain execution feedback may be insufficient. An interesting direction of work is identifying lightweight *proactive* monitoring mechanisms that enable a much larger set of expression cardinalities to be obtained from executing the plan. In [23], we identified a few such mechanisms both for single table expressions as well as certain key-foreign joins. Since these mechanisms do impose some additional overheads on execution, we outline a flexible architecture for execution feedback where the overhead incurred can be controlled by a DBA. An important problem that deserves more careful study is identifying a robust way to characterize which expressions are likely to be essential for obtaining a good plan. This is important to ensure that proactive monitoring mechanisms can be used effectively.

Another important conclusion we drew was the fact that the feedback cache approach in general can be extended to record and reuse accurate values of other key parameters that are used by the cost estimation module of the optimizer. We studied one particular parameter - DistinctPageCount: The number of distinct pages that contain tuples that satisfy a predicate (this is important in costing an index seek plan). We studied novel mechanisms – leveraging techniques such as probabilistic counting and page sampling – to obtain the DistinctPageCount parameter [22] using execution feedback. Our initial results indicate that getting the estimates right for this important parameter can lead to a significant improvement in query performance.

In summary, execution feedback is an interesting approach to potentially mitigate some of the estimation errors that affect query optimizers. While certain aspects of execution feedback have already impacted commercial products (e.g., self-tuning single column histograms in Sybase and a simple form of the feedback cache in Oracle), in our opinion effectively using execution feedback to improve cardinality estimates and other important parameters of the cost model remains largely an open problem.

4 Query Progress Estimation

Query progress estimation [19, 20, 27] can be useful in database systems: e.g., to help a DBA decide to whether or not to kill a query to free up resources, or for the governor in dynamic resource management. The first challenge in doing such estimation was to define the right *model* of the total work done by a query; we require a measure that is able to reflect progress at a fine granularity and can be computed at very low overhead as the query is executing. These constraints rule out most obvious models such as using the fraction of result tuples output by the query or the fraction of operators in the plan that have completed. Our model exploits the fact that modern query execution engines are based on the demand-driven iterator model of query processing. We used the total number of GetNext() calls issued over all operators in the execution plan as the measure of work done by the query. Accordingly, we can define the progress of a query at any point as the fraction of the total GetNext() calls issued so far. This measure provides a model to reason about progress independent of the specifics of the processing engine and in our experiments was shown to be highly correlated with the query's execution time itself (see [27], Section 6.7).

A key challenge in progress estimation becomes estimating the total number of GetNext() calls that will be issued. Therefore, the accuracy of progress estimation is tied to the accuracy of cardinality estimation. In general, the problem of providing robust progress estimates for complex queries is hard [20]. In particular, any nontrivial guarantee in a worst-case sense is not possible, even for the restricted case of single join queries (assuming the database statistics include only single column histograms).

To alleviate this, we proposed a number of different progress estimators [19, 20], which are more resilient to certain types of cardinality estimation errors and leverage execution feedback to improve upon initial estimates of GetNext() calls. Unfortunately, no single one of these estimators generally outperforms the others for arbitrary queries in practice; each estimator performs well for certain types of queries and data distributions and less so for others. Consequently, we proposed a framework based on statistical learning techniques [27] that selects among a set of progress estimators one best suited to the specific query, thereby increasing the overall accuracy significantly.

Acknowledgments: We are very grateful for the contributions of Sanjay Agrawal, who was a core member of the Autoadmin Project for several years and instrumental in shipping the Index Tuning Wizard and Database Tuning Advisor as part of Microsoft SQL Server. Raghav Kaushik played a pivotal role in our work on Query Progress Estimation. Finally, we would like to thank all the interns and visitors who have contributed immensely to this project.

References

[1] AutoAdmin Project `http://research.microsoft.com/en-us/projects/autoadmin/default.aspx`

[2] Aboulnaga, A. and Chaudhuri, S. Self-Tuning Histograms: Building Histograms Without Looking at Data. Proceedings of ACM SIGMOD, Philadelphia, 1999.

[3] Agrawal S., Chaudhuri S., Kollar L., and Narasayya V. Index Tuning Wizard for Microsoft SQL Server 2000. `http://msdn.microsoft.com/en-us/library/Aa902645(SQL.80).aspx`

[4] Agrawal, S., Chaudhuri, S. and Narasayya, V. Automated Selection of Materialized Views and Indexes for SQL Databases. In Proceedings of the VLDB, Cairo, Egypt, 2000.

[5] Agrawal, S. et al. Database Tuning Advisor for Microsoft SQL Server 2005. VLDB 2004.

[6] Agrawal, S., Narasayya, V., and Yang, B. Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In Proceedings of ACM SIGMOD Conference 2004.

[7] Bruno,N., Chaudhuri, S., Gravano, L. STHoles: A Multidimensional Workload-Aware Histogram. SIGMOD 2001.

[8] Bruno, N., and Chaudhuri, S. Automatic Physical Design Tuning: A Relaxation Based Approach. Proceedings of the ACM SIGMOD, Baltimore, USA, 2005.

[9] Bruno, N., and Chaudhuri, S. Constrained physical design tuning. VLDB J. 19(1): 21-44 (2010).

[10] Bruno, N., and Chaudhuri, S. An Online Approach to Physical Design Tuning. ICDE 2007.

[11] Bruno N. and Chaudhuri S. To Tune or not to Tune? A Lightweight Physical Design Alerter. VLDB 2006.

[12] Bruno, N., Nehme, R. Configuration-parametric query optimization for physical design tuning. SIGMOD 2008.

[13] Chaudhuri, S. An Overview of Query Optimization in Relational Systems. PODS 1998.

[14] Chaudhuri, S., Motwani, R., and Narasayya V. Random Sampling for Histogram Construction: How much is enough? In Proceedings of ACM SIGMOD 1998.

[15] Chaudhuri, S. and Narasayya, V. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. VLDB 1997.

[16] Chaudhuri, S. and Narasayya, V. AutoAdmin "What-If" Index Analysis Utility. SIGMOD 1998.

[17] Chaudhuri S. and Narasayya V. Index Merging. In Proceedings of ICDE, Sydney, Australia 1999.

[18] Chaudhuri, S., Gupta, A., and Narasayya, V. Compressing SQL workloads. SIGMOD 2002.

[19] Chaudhuri, S., Narasayya, V., Ramamurthy, R. Estimating Progress of Long Running SQL Queries. SIGMOD 2004.

[20] Chaudhuri, S., Narasayya, V., Ramamurthy, R.. When Can We Trust Progress Estimators for SQL Queries?. SIGMOD 2005.

[21] Chaudhuri, S. and Narasayya, V., Self-Tuning Database Systems: A Decade of Progress., VLDB 2007.

[22] Chaudhuri, S., Narasayya, R.Ramamurthy. Diagnosing Estimation Errors in Page Counts using Execution Feedback. ICDE 2008.

[23] Chaudhuri, S., Narasayya, V., Ramamurthy, R. A Pay-As-You-Go Framework for Query Execution Feedback. VLDB 2008.

[24] Graefe, G. The Cascades framework for query optimization. Data Engineering Bulletin, 18(3), 1995.

[25] Idreos, S., Kaushik, R., Narasayya, V., Ramamurthy, R. Estimating the compression fraction of an index using sampling. ICDE 2010: 441-444.

[26] Kimura, H., Narasayya, V., Syamala, M. Compression Aware Physical Database Design. PVLDB 4(10): 657-668 (2011).

[27] König, A., Ding, B. , Chaudhuri, S., Narasayya, V., A Statistical Approach Towards Robust Progress Estimation VLDB 2012.

[28] König, A. and Nabar, S. Scalable Exploration of Physical Database Design. ICDE 2006.

[29] Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T. Access Path Selection in a Relational Database. SIGMOD 1979.

[30] Srivastava, U., Haas, P., Markl, V., Kutsch, M., Tran, T. ISOMER: Consistent Histogram Construction Using Query Feedback. ICDE 2006.

[31] M. Stillger, G. Lohman, V. Markl, M. Kandil. LEO - DB2's Learning Optimizer. VLDB 2001.