Self-Tuning Database Systems: A Decade of Progress

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

ABSTRACT

In this paper we discuss advances in self-tuning database systems over the past decade, based on our experience in the AutoAdmin project at Microsoft Research. This paper primarily focuses on the problem of automated physical database design. We also highlight other areas where research on self-tuning database technology has made significant progress. We conclude with our thoughts on opportunities and open issues.

1. HISTORY OF AUTOADMIN PROJECT

Our VLDB 1997 paper [26] reported our first technical results from the AutoAdmin project that was started in Microsoft Research in the summer of 1996. The SQL Server product group at that time had taken on the ambitious task of redesigning the SQL Server code for their next release (SQL Server 7.0). Ease of use and elimination of knobs was a driving force for their design of SQL Server 7.0. At the same time, in the database research world, data analysis and mining techniques had become popular. In starting the AutoAdmin project, we hoped to leverage some of the data analysis and mining techniques to automate difficult tuning and administrative tasks for database systems. As our first goal in AutoAdmin, we decided to focus on physical database design. This was by no means a new problem, but it was still an open problem. Moreover, it was clearly a problem that impacted performance tuning. The decision to focus on physical database design was somewhat ad-hoc. Its close relationship to query processing was an implicit driving function as the latter was our area of past work. Thus, the paper in VLDB 1997 [26] described our first solution to automating physical database design.

In this paper, we take a look back on the last decade and review some of the work on Self-Tuning Database systems. A complete survey of the field is beyond the scope of this paper. Our discussions are influenced by our experiences with the specific problems we addressed in the AutoAdmin project. Since our VLDB 1997 paper was on physical database design, a large part of this paper is also devoted to providing details of the progress in that specific sub-topic (Sections 2-6). In Section 7, we discuss briefly a few of the other important areas where self-tuning database technology have made advances over the last decade. We reflect on future directions in Section 8 and conclude in Section 9.

VLDB '07, September 23-28, 2007, Vienna, Austria.

2. AN INTRODUCTION TO PHYSICAL DATABASE DESIGN

2.1 Importance of Physical Design

A crucial property of a relational DBMS is that it provides physical data independence. This allows physical structures such as indexes to change seamlessly without affecting the output of the query; but such changes do impact efficiency. Thus, together with the capabilities of the execution engine and the optimizer, the physical database design determines how efficiently a query is executed on a DBMS.

The first generation of relational execution engines were relatively simple, targeted at OLTP, making index selection less of a problem. The importance of physical design was amplified as query optimizers became sophisticated to cope with complex decision support queries. Since query execution and optimization techniques were far more advanced, DBAs could no longer rely on a simplistic model of the engine. But, the choice of right index structures was crucial for efficient query execution over large databases.

2.2 State of the Art in 1997

The role of the workload, including queries and updates, in physical design was widely recognized. Therefore, at a high level, the problem of physical database design was - for a given workload, find a *configuration*, i.e. a set of indexes that minimize the cost. However, early approaches did not always agree on what constitutes a *workload*, or what should be measured as *cost* for a given query and configuration.

Papers on physical design of databases started appearing as early as 1974. Early work such as by Stonebraker [63] assumed a parametric model of the workload and work by Hammer and Chan [44] used a predictive model to derive the parameters. Later papers increasingly started using an explicit workload [40],[41],[56]. An explicit workload can be collected using the tracing capabilities of the DBMS. Moreover, some papers restricted the class of workloads, whether explicit or parametric, to single table queries. Sometimes such restrictions were necessary for their proposed index selection techniques to even apply and in some cases they could justify the goodness of their solution only for the restricted class of queries.

All papers recognized that it is not feasible to estimate goodness of a physical design for a workload by actual creation of indexes and then executing the queries and updates in the workload. Nonetheless, there was a lot of variance on what would be the model of cost. Some of the papers took the approach of doing the comparison among the alternatives by building their own cost model. For columns on which no indexes are present, they built histograms and their custom cost model computed the selectivity of predicates in the queries by using the histograms.

Another set of papers, starting with [40], used the query optimizer's cost model instead of building a new external cost model. Thus the goodness of a configuration for a query was measured by the optimizer estimated cost of the query for that configuration. In this approach, although histograms still needed to be built on columns for which no indexes existed, no new cost model was necessary. This approach also required metadata changes to signal to the query optimizer presence of (fake) indexes on those columns. A concern in this approach is the potential impact on performance on the server and therefore there was a need to minimize the number of optimizer calls [40,41]. Some of the techniques to reduce optimizer calls introduced approximations, and thus led to lack of full fidelity with the optimizer's cost model.

The hardness result for selecting an optimal index configuration was shown by Shapiro [60]. Therefore, the challenge was similar to that in the area of query optimization – identifying the right set of heuristics to guide the selection of physical design. One set of papers advocated an approach based on rule-based expert systems. The rules took into account query structures as well as statistical profiles and were "stand-alone" applications that recommended indexes. A tool such as DEC RdbExpert falls in this category. Rozen and Shasha [56] also used an external cost model but their cost model was similar to that of a query optimizer. They suggested a search paradigm that used the best features of an individual query (using heuristics, without optimizer calls) and restricting the search to the union of those features. The latter idea of using best candidates of individual queries as the search space is valuable, as we will discuss later.

The "stand-alone" approaches described above suffered from a key architectural drawback as pointed out by [40], the first paper to propose an explicit workload model and also to use the query optimizer for estimating costs. This paper argued that the optimizer's cost model must be the basis to evaluate the impact of a physical design for a query. It also proposed building database statistics for non-existent indexes and making changes to system catalog so that optimizers can estimate costs for potential physical design configurations. Despite its key architectural contributions, there were several important limitations of this approach as will be discussed shortly.

3. REVIEW OF VLDB 1997 PAPER

3.1 Challenges

The AutoAdmin project started considering the physical design problem almost a decade after [40]. During this decade, tremendous progress was made on the query processing framework. The defining application of this era was decision-support queries over large databases. The execution engine supported new logical as well as physical operators. The engines used indexes in much more sophisticated ways; for example, multiple indexes per table could be used to process selection queries using index intersection (and union). Indexes were also used to avoid accessing the base table altogether, effectively being used for sequential scans of vertical slices of tables. These are known as "covering indexes" for queries, i.e., when a covering index for a query is present, the query could avoid accessing the data file. Indexes were used to eliminate sorts that would otherwise have been required for a GROUP BY query. The optimization technology was able to handle complex queries that could leverage these advances in execution engine. The workload that represented usage of the system often consisted of many queries and stored procedures coming from a variety of applications and thus no longer limited to a handful of queries.

While this new era dramatically increased the importance of the physical database design problem, it also exposed the severe limitations of the past techniques. The "expert system" based approach was no longer viable as building an external accurate model of index usage was no longer feasible. Therefore, the approach taken in [40] to use the optimizer's cost model and statistics was the natural choice. However, even there we faced several key gaps in what [40] offered. First, the necessary ingredients for supporting the needed API functionality in a client-server architecture was not discussed. Specifically, given that the databases for decision support systems were very large and had many columns, creating statistics using traditional full scan techniques was out of question for these databases. Second, the new execution engines offered many more opportunities for sophisticated index usage. Thus the elimination heuristics to reduce the search space of potential indexes (e.g., at most one index per table) was no longer adequate. Third, it was imperative that multi-column indexes were considered extensively as they are very important to provide "covering indexes". The search strategy of [40] did not consider multi-column indexes as they were of relatively low importance for execution engines and application needs of a decade ago. Finally, the scalability of the tool with respect to the workload size was also quite important as traces, either generated or provided by DBAs, could consist of many (often hundreds of thousands of) queries and updates, each of which can be quite complex.

3.2 Key Contributions

The first contribution of our AutoAdmin physical design project was to support creation of a new API that enabled a scalable way to create a hypothetical index. This was the most important server-side enhancement necessary. A detailed description of this interface, referred to as "what-if" (or hypothetical) index, appeared in [27] (see Figure 1). The key aspects 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 what-if index via the use of sampling [25],[20].

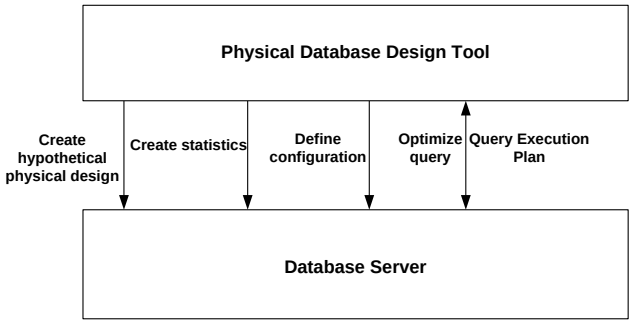

Figure 1. "What-if" analysis architecture for physical database design.

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 was important as the alternative

would have been repeated creation and dropping of what-if indexes, a potentially costly solution that was used by [40]. 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, it 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.

The next contribution was the key decision of defining the search space as consisting of the best configuration(s) for each query in the workload, where the best configuration itself is the one with lowest optimizer estimated cost for the query. Intuitively, this leverages the idea (see also [41]) that an index that is not part of an optimal (or close to optimal) configuration for at least one query, is unlikely to be optimal for the entire workload. Unlike [41], the selection of a *set of candidate indexes* on a per-query basis is done in AutoAdmin in a cost-based manner keeping the optimizer in the loop. This candidate selection step is key to scalable search.

Our VLDB 1997 paper also presented a set of optimizations for obtaining the optimizer estimated cost of a query for a given configuration (denoted by Cost (Q,C)) *without* having to invoke the query optimizer. The essential idea was to show how Cost (Q,C) could be *derived* from the costs of certain important subsets of C. Given the richness of the query processing capabilities of DBMS engines, a key challenge, addressed in [26] was defining what configurations should be considered atomic for a given query. By caching the results of the optimizer calls for atomic configurations, optimizer invocations for several other configurations for that query could be eliminated (often by an order of magnitude).

Finally, a search paradigm that was able to scale with the large space of multi-column indexes was proposed. The idea was to iteratively expand the space of multi-column indexes considered by picking only the winning indexes of one iteration and augmenting the space for the next iteration by extending the winners with an additional column. Intuitively, this exploits the idea that a two-column index (on say (A,B)) is unlikely to be beneficial for a workload unless the single column index on its leading column (i.e., index on (A)) is beneficial.

The above key ideas formed the basis of the Index Tuning Wizard that shipped in Microsoft SQL Server 7.0 [29], the first tool of its kind in a commercial DBMS. In the subsequent years, we were able to refine and improve our initial design.

4. INCORPORATING OTHER PHYSICAL DESIGN STRUCTURES

The VLDB 1997 paper [26] focused on how to recommend indexes for the given workload. Today's RDBMSs however support other physical design structures that are crucial for workload performance. Materialized views are one such structure that is widely supported and can be very effective for decision support workloads. Horizontal and vertical partitioning are attractive since they provide the ability to speed up queries with little or no additional storage and update overhead. The large additional search space introduced by these physical design structures requires new methods to deal with challenges in scalability. In this section we describe the significant extensions to the search architecture of [26] for incorporating materialized views and partitioning (horizontal and vertical). We begin with a brief review of materialized views and partitioning and the new challenges they introduce.

4.1 Physical Design Structures

4.1.1 Materialized Views

A materialized view (MV) is a more complex physical design structure than an index since a materialized view may be defined over multiple tables, and can involve selection, projection, join and group by. This richness of structure of MVs makes the problem of selecting materialized views significantly more complex than that of index selection. First, for a given query (and hence workload) the space of materialized views that must be considered is much larger than the space of indexes. For example, MVs on any subset of tables referenced in the query may be relevant. For each such subset many MVs with different selection conditions and group by columns may need to be considered. Furthermore, a materialized view itself can have clustered and non-clustered indexes defined on it. Finally, if there are storage and update constraints, then it is important to consider materialized views that can serve multiple queries. For example, if there are two candidate multi-table MVs, one with a selection condition Age BETWEEN 25 and 35 and another with the selection condition Age BETWEEN 30 and 40, then a MV with the selection condition Age BETWEEN 25 and 40 can be used to replace the above two materialized views but with potentially reduced storage and update costs. The techniques for searching the space of MVs in a scalable manner are of paramount importance.

4.1.2 Partitioning

Similar to a clustered index on a table, both horizontal and vertical partitioning are *non-redundant*, i.e., they incur little or no storage overhead. Also, in the same way that only one clustering can be chosen for a table, only one partitioning can be chosen for a table. This makes partitioning particularly attractive in storage constrained or update intensive environments.

Commercial systems today support hash and/or range horizontal partitioning, and in some cases hybrid schemes as well. Horizontal partitioning can be useful for speeding up joins, particularly when each of the joining tables are partitioned identically (known as a *co-located join*). Horizontal range partitioning can also be exploited for processing range queries. Finally, not only can a table be horizontally partitioned, but so can indexes on the table. Thus a large new search space of physical design alternatives is introduced. Another important scenario for using horizontal partitioning is *manageability*, in particular to keep a table and its indexes *aligned*, i.e., partitioned identically. Alignment makes it easy to load new partitions of a table (and remove old partitions) without having to rebuild all indexes on the tables. From the perspective of physical design tuning, alignment therefore becomes an additional constraint that must be obeyed while tuning.

Major commercial relational database systems do not natively support vertical partitioning. Thus achieving the benefits of vertical partitioning in such systems raises additional considerations. Specifically, the logical schema (i.e. table definitions) needs to change [8]. In turn, this requires that application queries and updates may need to be modified to run

against the new schema. Alternatively, views can be defined that hide the schema changes from application queries. If the above class of views is updateable, then update statements in the application do not need to be modified either.

4.2 Search Algorithms

The introduction of materialized views and partitioning results in an explosion in the space of physical design alternatives. In this section, we present three techniques that enable physical design tools to explore this large space in a scalable manner. The use of these techniques led to significant extensions and changes in the search architecture presented in [26]. These techniques are general in the sense that the concepts are applicable to all physical design structures discussed in this paper. They enable a uniform search architecture for structuring the code of a physical design tool. The architecture that evolved as a result of these advances is shown in Figure 2. These extensions are in fact part of the product releases of Index Tuning Wizard in SQL Server 2000 [4] and Database Engine Tuning Advisor in SQL Server 2005 [8]. In the rest of this section we describe the key steps in this architecture; highlighting the challenges and solutions. Note that the candidate selection step is unchanged with respect to [26] and hence we do not focus on it here.

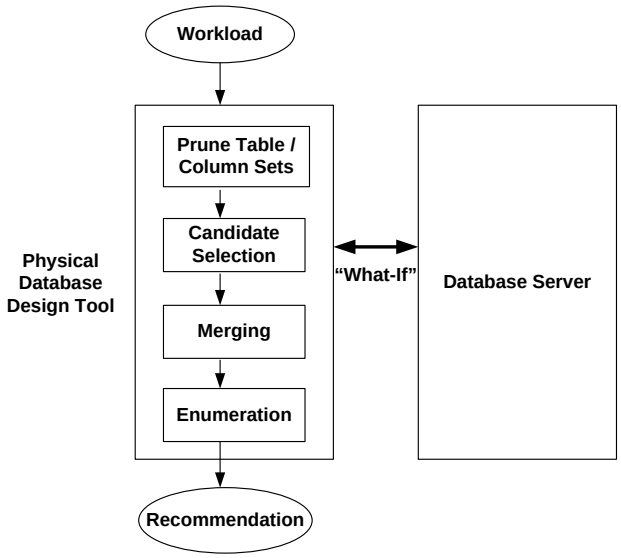

Figure 2. Search Architecture of a Physical Database Design tool.

4.2.1 Pruning Table and Column Sets

Whenever there is a query over multiple tables in the workload, materialized views over tables mentioned in the query (henceforth called *table sets*), or subsets of those tables, can be relevant. Therefore it becomes crucial to prune the search space early on, since otherwise even the candidate selection step does not scale as there could be a very large number of materialized views over table sets in the workload. One key observation presented in [5] is that in many real workloads, a large number of table sets occur infrequently. However, any materialized views on table sets that occur infrequently cannot have a significant impact on overall workload performance. Of course, the impact cannot be measured by frequency alone, but needs to be weighted by the cost of queries. The above observation allows leveraging a variation of frequent itemsets technique [3] to eliminate from consideration a large number of such table sets very efficiently. Only table sets that survive the frequent itemset pruning are considered during the candidate selection step. The same intuition was subsequently extended in [10] to prune out a large number of *column sets*. Column sets determine which multi-column indexes and partitioning keys are considered during the candidate selection step. This technique allowed the elimination of the iterative multi-column index generation step in [26] (see Section 3.2), while still retaining the scalability and quality of recommendations.

4.2.2 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 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 those indexes do not fit within the available storage or incur 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 e.g., if 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 [28]. The key ideas were to: (1) define how a given pair of indexes is merged, and (2) generate merged indexes from a given set, using (1) as the building block.

View merging introduces challenges over and beyond index merging. Merging a pair of views (each of which is a SQL expression with selections, joins, group by) is non-trivial since the space of merged views itself is very large. Furthermore, the expressiveness of SQL allows interesting transformations during merging. For example, given a multi-table materialized view V_1 with a selection condition (State='CA') and V_2 with (State = 'WA'), the space of merged views can also include a view V_{12} in which the selection condition on the State column is eliminated, and the State column is pushed into the projection (or group by) list of the view. Scalable techniques for merging views that explored this space are presented in [5],[16].

An alternative approach for generating additional candidate MVs that can serve multiple queries in the workload by leveraging multi-query optimization techniques was presented in [70]. An open problem is to analyze and compare the above approaches in terms of their impact on the quality of recommendations and scalability of the tool.

4.2.3 Enumeration

Given a workload and a set of candidates, obtained from the candidate selection step and augmented by the merging step, the goal of the enumeration is to find a configuration (i.e., subset of candidates) with the smallest total cost for the workload. Note also that we also allow DBAs to specify a set of constraints that the enumeration step must respect, e.g., to keep all existing

indexes, or to respect a storage bound (See Section 6.1 for details). Since the index selection problem has been shown to be NP-Hard [21],[60], the focus of our work has been on developing heuristic solutions that give good quality recommendations and can scale well.

One important challenge is that solutions that naively *stage* the selection of different physical design structures (e.g., select indexes first followed by materialized views) can result in poor recommendations. This is because: (1) The choices of these structures interact with one another (e.g., optimal choice of index can depend on how the table is partitioned and vice versa). (2) Staged solutions can lead to redundant recommendations. For example, assume that a beneficial index is picked first. In the next stage when MVs are considered, a materialized view that is significantly more beneficial than the index is picked. However, once the materialized view is picked, the index previously selected may contribute no additional benefit whatsoever. (3) It is not easy to determine a priori how to partition the storage bound across different physical design structures [5]. Thus, there is a need for *integrated* recommendations that search the combined space in a scalable manner.

Broadly the search strategies explored thus far can be categorized as bottom-up [26],[55],[69] or top-down [15] 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 but it 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 [15] strategy can be 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.

5. RECENT ADVANCES IN PHYSICAL DATABASE DESIGN

In Section 4 we described impact on the physical design problem that arises from going beyond indexes to incorporate materialized views and partitioning. In this section we discuss more recent advances that revisit some of the basic assumptions made in the problem definition thus far. The importance of a physical design tool to stay in-sync with the optimizer using the "what-if" interfaces [27] was highlighted earlier. First (in Section 5.1), we describe recent work on enhancing this interface to improve both the degree to which the tool is in sync with the optimizer, resulting in both improved quality of recommendation as well as scalability. Next, in Section 5.2, we discuss alternative tuning models that can potentially serve a different class of scenarios.

5.1 Enhancing the "What-if" Interface

The idea of selecting a set of candidate indexes per query in a cost-based manner is crucial for scalability of a physical database design tool (Section 3). Observe that the approach of [26] requires the client tool to search for the best configuration for each query, potentially using heuristics such as those discussed in Section 4. This can result in selection of candidates that are not optimal. A more recent idea presented in [15] is to instrument the optimizer itself to generate the candidate set for each query. Most query optimizers (such as those based on System-R [59] or Cascades [43] frameworks) rely on a crucial component that transforms single-table logical sub-expressions into index-based execution sub-plans. This procedure considers the set of available indexes to generate execution sub-plans including index scans, index intersections, lookups, etc. In the approach of [15], each such transformation request is intercepted. The logical sub-expression is then analyzed to identify the indexes that would result in the optimal execution sub-plan. The metadata for such indexes is then added to the system catalogs and optimization is resumed as normal (see Figure 3). This ensures that the optimizer picks the optimal access methods for the query. This approach leverages the observation that the set of transformation requests issued by the optimizer does not depend on the existing set of indexes.

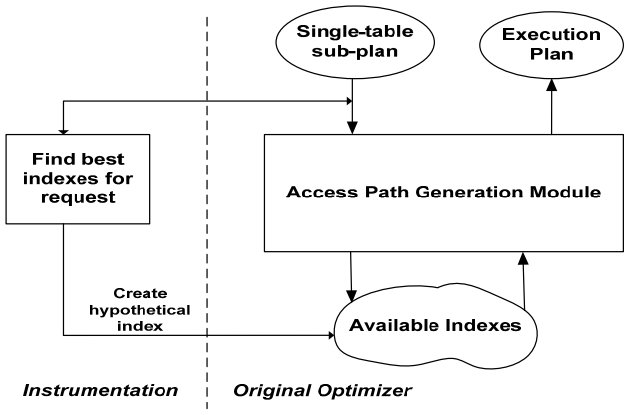

Figure 3. Instrumenting the optimizer to generate candidate indexes.

Thus the candidate selection step is now more in-sync with the optimizer than in the earlier approach of [26]. Since requests are intercepted during optimization, the above technique does not miss candidates as in [26]. Also, unlike [69] it does not propose candidates that are syntactically valid but might not be exploited by the optimizer. The number of such requests and the number of candidates for such requests is shown to be relatively small even for complex workloads. Thus this extension to the "what-if" interface can result in: (a) the solution being more deeply in-sync with the optimizer and (b) improved scalability for obtaining candidates by reducing the number of optimizer calls. Finally, as described in [15] we note that this technique can also be extended to deal with materialized views.

5.2 Alternative Tuning Models

Even for DBAs who would like to depend exclusively on their insights without using a physical design advisor, the "what-if" physical design analysis capabilities [27] is helpful as they are now able to quantitatively explore the impact of their proposed changes using optimizer's cost estimates. Thus, they are now able to iteratively refine and evaluate their alternatives without having to ever create/drop any physical design structures.

The tuning model discussed thus far in this paper requires the DBA to provide a workload, and the tool provides a recommended physical design configuration. As will be discussed in Section 6, this is indeed the model of physical design tool that the commercial relational database systems support. Although the tool frees DBAs from having to choose specific physical design structures one at a time, the DBA needs to: (1) Decide when to

invoke the tool. (2) Decide what "representative" workload to provide as input to the tool. (3) Run the tool and examine the recommended physical design changes, and implement them if appropriate. In this section, we describe some of our recent work in trying to further simplify the above tasks that the DBA faces. While Sections 5.2.1 and 5.2.2 describe techniques that still retains the model of physical design tuning based on static workload, the work on Dynamic Tuning, described in Section 5.2.3, describes initial work on an online approach that continuously monitors the workload and makes changes without the DBA having to intervene.

5.2.1 Alerter (When to Tune)

One way to address the issue of changing workloads and data characteristics requirement on the DBA is deciding when the physical design tool must be invoked. This can be challenging particularly since the workload pattern and data distributions may change. Therefore, a useful functionality is having a *lightweight* "alerter" capability that can notify the DBA when significant tuning opportunities exist. The work in [14] (see also Section 7.2.2) presents a low overhead approach that piggybacks on normal query optimization to enable such functionality. The idea is to have a lightweight adaptation of the optimizer instrumentation techniques presented in Section 5.1 by only recording index requests for the plan chosen by the optimizer. As detailed in [14], this enables the alerter to provide a lower bound on the improvement that would be obtained if the workload were to be tuned by a physical design tool.

5.2.2 Workload as a Sequence

Our tuning model assumes that the workload is a *set* of queries and updates. If we were to instead view the workload as a sequence or a sequence of sets, then better modeling of real world situations are possible. For example, in many data warehouses, there are mostly queries during the day followed by updates at night. Thus, viewing workload a sequence of "set of read queries" followed by a "set of update queries" makes it possible to handle variations in workload over time and to exploit properties of the sequence to give a better performance improvement by creating and dropping structures at appropriate points in the sequence. Of course, the tool has to now take into account the cost of creating/dropping the physical design structure as well (e.g., in the data warehouse situation, the cost to drop the index before the nightly updates, and recreate indexes after the updates are completed). A framework for automated physical design when the workload is treated as a sequence is presented in [9].

5.2.3 Dynamic (Online) Tuning

The goal of Dynamic Tuning is to have a server-side "always-on" solution for physical database design that requires little or no DBA intervention [13],[57],[58]. Thus, dynamic tuning component tracks the workload and makes an online decision to make changes to physical design as needed. In fact, in some situations where the workload may change too unpredictably, dynamic tuning may be the only option. For example, in a hosted application environment, a new application can be deployed, run and removed, all in a relatively short period of time. Naturally, dynamic tuning needs to depend on the enabling technology of *online* index creation and drop, which is supported by today's commercial DBMSs.

There are three key new challenges for a continuous tuning system. First, since it is always-on, the solution has to have very low overhead and not interfere with the normal functioning of the DBMS. Second, the solution must balance the cost of transitioning between physical design configurations and the potential benefits of such design changes. Finally, the solution must be able to avoid unwanted oscillations, in which the same indexes are continuously created and dropped.

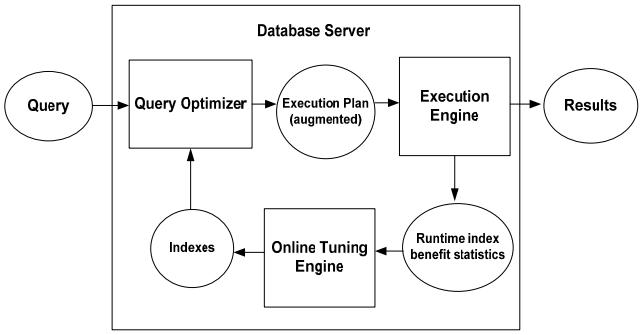

Figure 4. An architecture for online index tuning.

The work in [13] presents an online algorithm that can modify the physical design as needed. It is prototyped inside the Microsoft SQL Server engine. The broad architecture of the solution is shown in Figure 4. At query optimization time, the set of candidate indexes desirable for the query are recorded by augmenting the execution plan. During execution time the Online Tuning Engine component tracks the potential benefits that are lost by not creating these candidate indexes, as well as the utility of existing indexes. When sufficient evidence has been gathered that a physical design change is beneficial, then the index creation (or deletion) is triggered online. Since an online algorithm cannot see the future, its choices are bound to be suboptimal compared to an optimal offline solution (which knows the future), but the design of the algorithm attempts to bound the degree of such sub-optimality. The work in [57],[58] share similar goals as [13] but differ in the design points of: (a) the degree to which they are coupled with the query optimizer (b) permissible overheads for online index tuning.

A new approach for online physical design tuning is *database cracking* [45],[46]. In this work, each query is interpreted not only as a request for a particular result set, but also as a suggestion to crack the physical database store into smaller pieces. Each piece is described by a query expression, and a "cracker index" tracks the current pieces so that they can be efficiently assembled as needed for answering queries. The cracker index is built dynamically while queries are processed and thus can adapt to changing query workloads. In the future, a careful comparison of database cracking to other online tuning approaches such as the ones described above, needs to be done.

6. IMPACT ON COMMERCIAL DBMS

All major commercial database vendors today ship automated physical design tools. We discuss these commercial tools in Section 6.1. Building an industrial strength physical design tool poses additional challenges not discussed thus far. We highlight three such challenges and approaches for handling them in Section 6.2-6.4.

6.1 Physical Design Tuning Tools in Commercial DBMS

In 1998, Microsoft SQL Server 7.0 was the first commercial DBMS to ship a physical database design tool, called the Index Tuning Wizard (ITW) based on the techniques presented in [26],[27]. In the next release, Microsoft SQL Server 2000, ITW was enhanced to provide integrated recommendations for indexes, materialized views (known as *indexed views* in Microsoft SQL Server) and indexes on indexed views. In the most recent release of Microsoft SQL Server 2005, the functionality of ITW was replaced by a full-fledged application called the Database Engine Tuning Advisor (DTA) [7]. DTA can provide *integrated recommendations* for indexes, indexed views, indexes on indexed views and horizontal range partitioning. DTA allows DBAs to express constraints including aligned partitioning (see Section 4.1.2), storage constraints, existing physical design structures that must be retained, etc. In addition, DTA exposes a rich set of tuning options, e.g., which tables to tune, etc. An important usage scenario for DTA is tuning a single problem query. Therefore DTA functionality can also be invoked directly from the SQL Server Management Studio, the tool environment from which DBAs often troubleshoot queries. DTA's recommendations are accompanied by a set of detailed analysis reports that quantify the impact of accepting DTA's recommendations. The tool also exposes "what-if" analysis functionality to facilitate manual tuning by advanced DBAs. More details of DTA, its usage and best practices are available in a white paper [8].

IBM's DB2 Universal Database (UDB) version 6 shipped the DB2 Advisor [66] in 1999 that could recommend indexes for a given workload. Subsequently, the DB2 Design Advisor tool in DB2 version 8.2 [69] provides integrated recommendations for indexes, materialized views, shared-nothing partitioning and multi-dimensional clustering. One difference between this tool and DTA (or ITW) is *how* an integrated recommendation for different physical design structures is produced. Unlike DTA where the search over all structures is done together, DB2 Design Advisor is architected to have independent advisors for each physical design structure. The search step that produces the final integrated recommendation iteratively invokes each of the advisors in a staged manner.

Oracle 10g shipped the SQL Access Advisor [37], which takes as input a workload and a set of candidates for that workload (generated by the Oracle Automatic Tuning Optimizer on a per-query basis), and provides a recommendation for the overall workload. The tool recommends indexes and materialized views.

Finally, we note that recent textbooks on database design e.g., [49] devote significant coverage to advances in this area over the past ten years and suggest use of automated physical design tools in commercial DBMS systems.

6.2 Tuning Large Workloads

One of the key factors that affect the scalability of physical design tools is the size of the workload. DBAs often gather a workload by using server tracing tools such as DB2 Query Patroller or 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 [7]. Therefore, techniques for compressing large workloads become essential. A constraint of 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 presented in [22]. The idea is to exploit the inherent templatization in workloads by partitioning the workload based on the "signature" of each query, i.e., two queries have 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 [22], the 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.

6.3 Tuning Production Servers

Ideally, DBAs would like to perform physical design tuning directly against the production server. The tuning architecture described above can however impose non-trivial load since a physical design tuning tool may need to make repeated calls to the query optimizer. A key idea is to transparently leverage test servers that are typically available in enterprises. We can leverage the fact that the "what-if" analysis architecture [27] does not require the physical design structures to be actually materialized since queries are only optimized and never executed. Thus only a "shell" database is imported into the test server [7] before tuning. A shell database contains all metadata objects (including statistics) but not the data itself. Observe that since physical design tools may need to create statistics while tuning, any required statistics are created on the production server and imported into the test server. Finally, the "what-if" interfaces of the query optimizer need to be extended to take as input the H/W characteristics such as CPU and memory. This allows the tool to simulate H/W characteristics of the production server on a test server whose actual H/W characteristics may be different, and thereby ensure that the recommendations obtained are identical as if the production server was tuned directly.

6.4 Time Bound Tuning

In many enterprise environments, there is a periodic batch window in which database maintenance and tuning tasks are performed. DBAs therefore would like to run physical database design tools so that they complete tuning within the batch window. Intuitively, we need to find a good recommendation very quickly and refine it as time permits. To address this requirement at each step during tuning, the physical database design tool must make judicious tradeoffs such as: (1) Given a large workload should we consume more queries from the workload or tune the ones consumed thus far? (2) For a given query, should we tune both indexes and materialized views now or defer certain physical design structures for later if time permits? (e.g., an index may be useful for many queries whereas a materialized view may be beneficial only for the current query). Thus the techniques described in Sections 3 and 4 require adaptations to be effective in the presence of such a time constraint. We note that current commercial physical design tuning tools such as [7],[37],[69] support time bound tuning.

7. ADVANCES IN OTHER SELF-TUNING DATABASE TECHNOLOGY

Self-tuning databases is a wide area of research and it is hard to even draw boundaries around it. Our coverage of recent advances in this area is by no means exhaustive. There are several good resources that give additional details of recent work in self-tuning databases, e.g., the VLDB ten-year award paper from 2002 [67], a tutorial on self-tuning technology in commercial databases [18], a tutorial on foundations of automated tuning that attempts to break down the area into paradigms [33]. We have discussed several recent advances in the areas of physical database design in previous sections.

In Sections 7.1 and 7.2, we focus on *statistics management* and DBMS *monitoring infrastructure*, two research ideas that the AutoAdmin project explored. Given the large breadth of the area, we are able to highlight only a few of the many notable advances in other self-tuning database topics in Section 7.3.

7.1 Statistics Management

Absence of the right statistical information can lead to poor quality plans. Indeed, when we discussed automated selection of physical design, along with our recommendation for indexes, we needed to recommend a set of database statistics to ensure that the optimizer has the necessary statistics to generate plans that leverage the recommended indexes. However, the problem of selection of database statistics arises even if we are not contemplating any changes in physical design. Thus, in Sec 7.1.1, we discuss the problem of selecting statistics to create and maintain in a database system. In Section 7.1.2, we focus on self-tuning histograms, an active area of research. The key idea behind self-tuning histograms is to see how an individual statistics object (specifically histograms) can leverage execution feedback to improve its accuracy. Thus, these two problems are complementary to each other.

7.1.1 Selection of Statistics

Determining which statistics to create is a difficult task, since the decisions impact quality of plans and also the overhead due to creation/update of statistics. Microsoft SQL Server 7.0 pioneered in 1998 use of auto-create-statistics, which causes the server to automatically generate all single-column histograms (via sampling) required for the accurate optimization of *an ad-hoc* query. This technology is now available among all commercial relational database management systems. A recent paper [39] suggests expanding the class of such statistics (beyond single column statistics) that are auto-created in response to an incoming ad-hoc query. While attractive from the perspective of improving quality of plans, such proposals need careful attention so that the incremental benefit of auto-creating such structures does not make the cost of optimization disproportionately high.

A fundamental problem underlying the selection of statistics to auto-create is evaluating the usefulness of a statistic without creating it. Today's technology for auto-create-statistics uses syntactic criteria. However, for a wider class of statistics (such as multi-column), syntactic criteria alone are not sufficient and more powerful pruning of candidate set is desirable. Magic number sensitivity analysis (MNSA) [30] was proposed as a technique to address this problem. The key idea is to impose a necessary condition before a syntactically relevant statistics is materialized. Specifically, if the statistics for a join or a multi-column selection predicate p in a query Q is potentially relevant, then the choice of query plan for Q will vary if we optimize the query Q by injecting artificially extreme selectivities for p (e.g., 1-ε, 0.5, ε). If the plan for Q does not change, then we consider the candidate statistics irrelevant and do not build it. MNSA was initially proposed to solve the problem of finding an ideal set of statistics for a given static workload (referred to as the *essential set* of statistics in [30]) and further improvements are desirable to adapt the technique for completely ad-hoc queries. Note also that the decision to determine which statistics to create can be driven not only by ad-hoc queries or by a static workload, but also by leveraging execution feedback to determine where statistical information may be lacking [2]. Finally, the challenge of maintenance is also non-trivial and needs to rely on coarse counters to track modification of tables (and potentially columns) as well as execution feedback.

All the above challenges of selection of statistics are significantly magnified as the class of statistics supported in DBMS expands. Recent proposals [17],[42] suggest using statistics on the result of a view expression (including joins and selections). Such statistics can lead to improved estimates as effects of correlation among columns and tables can be directly captured. In addition to the increased creation and maintenance cost, including such statistics also greatly expands the space of database statistics. The challenging problems of automated selection of such statistics and leveraging query execution feedback to refine them remain mostly unexplored thus far.

7.1.2 Self-Tuning Histograms

Histograms represent compact structures that represent data distributions. *Self-tuning histograms*, first proposed in [1], use execution feedback to bias the structure of the histogram so that frequently queried portions of data are represented in more detail compared to data that is infrequently queried. Use of self-tuning histograms can result in better estimates if incoming queries require cardinality estimation for point or range queries in the interval that have been queried in the past. Thus, instead of keeping observations from execution feedback as a separate structure as in [62], self-tuning histograms factor in execution feedback by modifying the histogram itself. Naturally, self-tuning histograms are especially attractive for multi-dimensional histograms where biasing the structure of the histogram based on usage pattern can be especially beneficial as the space represented by the histograms grow exponentially with number of dimensions. It should be noted that while online execution feedback is needed to record the observed cardinalities, the actual changes to the histogram structure can be done offline as well.

The challenge in building self-tuning histogram is to ensure that online execution feedback can be used in a robust manner without imposing a significant runtime overhead. The original technique proposed in [1] monitored only the overall actual cardinality of selection queries and was a low overhead technique. The actual cardinality of the query was compared with the estimated cardinality and the error was used to adjust the bucket boundaries as well as the frequency of each bucket. However, the histogram modification technique was based on relatively simple heuristics that could lead to inaccuracies. A subsequent work [18] made two significant improvements. First, it proposed using a multi-dimensional histogram structure that is especially suited to incorporate execution feedback. Next, it also recognized that a

finer granularity of execution feedback can significantly improve accuracy of tuning the histogram structure. Thus, it suggested techniques to track differences between execution and estimated feedback at individual bucket level of the histogram. However, despite improvement with respect to accuracy, the added monitoring raised the overhead of execution. A recent paper [61] addressed this concern by using the same multi-dimensional structure as proposed in [18] but restricting monitoring to the coarse level as in [1]. Instead of additional monitoring, ISOMER uses the well-known maximum entropy principle to reconcile the observed cardinalities and to approximate the data distribution.

7.2 Monitoring Infrastructure

In the past, database management systems provided rather limited visibility into the internal state of the server. Support for database monitoring (in addition to monitoring tools provided by operating systems) included ability to generate event traces, e.g., IBM Query Patroller or SQL Server Profiler. Awareness that transparency of relevant server state can greatly enhance manageability has led to significant extensions to monitoring infrastructure in commercial database systems. Examples of such extensions include Dynamic Management Views and functions (DMV) in Microsoft SQL Server that return server state information which can be used to monitor the health of a server instance, diagnose problems, and tune performance. These views/functions can represent information that is scoped for the entire server or can be specific to database objects. Another example of advanced monitoring infrastructure is Oracle's Automatic Workload Repository (AWR) that represents performance data-warehouse of information collected during execution of the server. Despite these significant developments, we consider this area to be very promising for further work. We illustrate this promise by pointing out the difficulty of answering a simple monitoring task such as query progress estimation (Section 7.2.1) and then the challenge of providing a platform to enable ad-hoc DBA defined monitoring tasks (Sec 7.2.2).

7.2.1 Query Progress Estimation

For a given query, one can query its historically aggregated usage information or its current state of execution. A very natural monitoring task is to be able to estimate "query progress", i.e. to estimate the percentage of a query's execution that has completed. In fact, one can view query progress estimation as an instance of a property of current execution. This information can be useful to help the DBA of an overloaded system select queries to be killed or to enforce admission control. The problem of estimating progress of a sequential scan is easy. Of course, query progress estimation is a harder problem than estimating the cost of a sequential scan since SQL queries can have selection, join, aggregation and other operators. Indeed, it has been shown that even for the simple class of SPJ queries, this problem is surprisingly hard. In the worst case, with the limited statistics available in today's database systems, no progress estimator can guarantee constant factor bounds [23]. Despite this negative result, the properties of the execution plan, data layout, and knowledge of execution feedback can be effectively used to have robust progress estimators that are able to overcome exclusive dependence on query optimizer's cardinality estimates [31],[50],[51],[52],[54]. This is analogous to query optimization – despite the problem being difficult, many queries are well served by our repertoire of query optimization techniques.

7.2.2 Ad-hoc Monitoring and Diagnostics

Despite availability of more capable monitoring infrastructure as mentioned at the beginning of Sec 7.1, support for ad-hoc monitoring is limited to selection of attributes to monitor and their thresholding. For example, it is hard for DBAs to pose a question such as: "Identify instances of a stored procedure that execute more than twice as slow as the average instance over a window of last 10 executions". Of course, the challenge in supporting such ad-hoc monitoring queries is to ensure that the overhead is not high. In [24], we presented a preliminary proposal for the SQL Continuous Monitoring (SQLCM) infrastructure that is built on the server-side with the goal of supporting such ad-hoc monitoring queries. This infrastructure supports aggregation of system state and allows the user to also specify ad-hoc monitoring tasks by using lightweight Event-Condition-Action (ECA) rules. Finally, the area of database system diagnostics has received much less attention so far than it deserves. The Automatic Diagnostic Monitor (ADDM) in Oracle database system represents an example of a diagnostic system that is able to analyze information in its performance data-warehouse and can invoke appropriate performance tuning tool based on pre-defined rules [37]. In our opinion, ad-hoc monitoring and diagnostics deserves much more attention than it has received so far.

7.3 Examples of Key Self-Tuning Initiatives

The COMFORT project [68] was one of the early self-tuning efforts that focused on important problems such as load control for locking, dynamic data placement in parallel disk systems, and workflow server configuration. Although feedback control loops are used in setting the appropriate tuning knobs, problem specific techniques were needed to achieve robust auto-tuning [67].

Improving accuracy of cardinality estimates using execution feedback has been an active area of work. The first paper leveraging execution feedback was [34] and was followed by papers on self-tuning histograms, discussed earlier. The Learning Optimizer (LEO) project [62], part of IBM's *autonomic computing* initiative, identifies incorrect cardinality estimates and saves the execution feedback for future optimization. Their goal is to serve a larger class of query expressions through such feedback beyond selection queries. Although a recent effort proposed using their execution feedback to create a self-tuning histogram [61], it remains an open problem on how effectively and efficiently execution feedback can be leveraged for more general class of query expressions, even if an incoming query does not exactly match the query expressions observed in the past.

Note that exploiting query execution feedback is useful not only for cardinality estimates for the future queries or for progress estimation, but such feedback has been leveraged for dynamic query re-optimization [47][53]. A novel query processing architecture that fundamentally relies on adaptive approaches rather than on careful static optimization was proposed in [12]. An upcoming survey [38] summarizes this direction of work.

All commercial relational database systems today consider manageability and self-tuning features as key requirements. In the earlier sections, we have described product features related to physical design tuning, statistics management, monitoring and diagnostics. Two other areas where there has been significant progress in the past decade include automated memory as well as automated storage/data layout management. For example, automated memory management in database servers makes it

possible to leverage adaptive operators and adjust memory assigned to each operator's working memory dynamically depending on its own needs as well as on global memory demands [19],[36], [63].

Finally, we would like to end our discussion of past work by mentioning two other directions of work that strike us as thought-provoking. The GMAP framework [65] suggested that physical designs can be specified as expressions over logical schema coupled with references to key storage organization primitives. The vision underlying the approach is to represent different physical organizations uniformly. Another topic that could be potentially interesting from the perspective of self-tuning technology is virtualization. While hardware and operating systems virtualization is increasingly popular, the ability to support high performance database applications on shared virtual machines raise many challenges since database systems traditionally use machine resources in a deliberate and careful manner [64].

8. FUTURE DIRECTIONS

As mentioned in the previous section, there are many active directions of work in the context of self-tuning database technology. In this section, we highlight a few of the interesting open issues:

- Today's commercial database systems include physical database design tools as part of the products. However, the ability to compare the quality of automated physical design solutions in these products remains an elusive task. To be fair, this is no different than the state of the art in comparing the quality of the query optimizers. But, from a research perspective, this situation is quite unsatisfactory and requires further thought.
- For large databases, any changes in the physical design are "heavyweight" operations. There have been proposals on more lightweight approaches towards defining physical design structures, e.g., partial indexes/materialized views [48], database "cracking" [45][46]. Such changes can redefine our approaches to physical database design.
- Emerging shopping, CRM, and social network services on the internet use database systems on the backend and they bring unique self-tuning challenges. Specifically, they employ *multi-tenancy*, i.e., data from different tenants (customers of their services) co-reside in the same database objects. Multi-tenancy makes self-tuning harder as workload characteristics and performance tuning are less predictable. Furthermore, efficient distributed monitoring and serviceability to handle failure and performance problems is an essential requirement for such internet services. This requirement provides a rare opportunity to rethink system architectures with self-tuning and self-manageability in mind. In fact, there are already several initiatives towards new generation of distributed architectures for storage, computing and data analysis that are being built with such monitoring and serviceability requirements, e.g., Amazon's S3 and EC2, Google Map Reduce, Microsoft Dryad.
- Machine learning techniques, control theory, and online algorithms have the potential to be leveraged even more for self-tuning tasks that we face for our data platforms. The main challenges here are in modeling the self-tuning tasks for which any of these paradigms could be applied in a robust way. For example, in order to apply machine learning, we need a clear understanding of what features (observed as well as computed) should be used for learning.

9. CONCLUSION

The widespread use of relational database systems for mission critical OLTP and decision support applications has made the task of reducing the cost of managing relational database systems an important goal. It has been a decade since we started the AutoAdmin research project. During this time, other research projects and industrial efforts also began to address this important problem. In some areas such as automated physical design and monitoring, our progress has had led to incorporation of new tools and infrastructure in relational database systems. Other areas remain active areas of research. Nonetheless, the challenge in making database systems truly self-tuning is a tall task. For example, the nature of tuning a buffer pool or tuning allocation of working memory for queries is very different from that of selecting the right set of indexes or statistics. Each such tuning problem has different abstractions for workloads and different constraints on the desired solution. Therefore, it will probably be impossible to make database systems self-tuning by a single architectural or algorithmic breakthrough. As a consequence, it will be a long journey before this goal is accomplished just as it took the automobile industry a sustained effort to reduce the cost of ownership. However, one worrisome factor that will slow our progress towards making relational database systems self-tuning is the complexity of internal components that have been fine tuned for performance for a powerful language such as SQL. As argued in [32],[67], it is worthwhile to explore alternative architectures of database servers for performance (and functionality) vs. manageability trade-off. While the business need for backward compatibility makes it difficult to revisit such trade-offs for traditional enterprise relational servers, the emergence of extremely scalable storage and application services over the internet that absolutely demand self-manageability could lead to development of newer structured store that is built grounds-up with self-manageability as a critical requirement.

10. ACKNOWLEDGMENTS

We thank the VLDB 10-year Best Paper Award Committee for selecting our paper for the award. Working with members of the AutoAdmin research team has been a great experience for both of us. Sanjay Agrawal and Manoj Syamala made significant contributions to physical database design tuning including incorporation of this technology to the Microsoft SQL Server product. Nicolas Bruno has driven much of the recent AutoAdmin advances on physical design tuning. Raghav Kaushik, Christian Konig, Ravi Ramamurthy, and Manoj Syamala contributed to the monitoring aspects of the AutoAdmin project. The shipping of the Index Tuning Wizard and subsequently the Database Engine Tuning Advisor in Microsoft SQL Server was made possible due to the commitment and support of many people in the SQL Server team over the past 10+ years. We sincerely thank them for their continued support. We are indebted to David Lomet for his support and encouragement. Over the years several visitors to our group at Microsoft Research made important contributions to the AutoAdmin project. Specifically, in this paper, we referred to the work done by Ashraf Aboulnaga, Eric Chu, Mayur Datar, Ashish Gupta, Gerhard Weikum, and Beverly Yang. Last but not least,

we thank Arvind Arasu, Nicolas Bruno, Raghav Kaushik, and Ravi Ramamurthy for their thoughtful comments on drafts this paper.

11. REFERENCES

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

[2] Aboulnaga, A., Haas, P., Lightstone,S., Lohman,G., Markl, V., Popivanov, I., Raman, V.: Automated Statistics Collection in DB2 UDB. *In Proceedings of VLDB 2004.*

[3] Agrawal, R., Ramakrishnan, S. Fast Algorithms for Mining Association Rules in Large Databases. *In Proceedings of VLDB 1994*.

[4] Agrawal S., Chaudhuri S., Kollar L., and Narasayya V. Index Tuning Wizard for Microsoft SQL Server 2000. http://msdn2.microsoft.com/en-us/library/Aa902645(SQL.80).aspx

[5] 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..

[6] Agrawal, S., Chaudhuri, S., Das, A, and Narasayya, V.: Automating Layout of Relational Databases. *In Proceedings of ICDE 2003*.

[7] Agrawal, S. et al. Database Tuning Advisor for Microsoft SQL Server 2005. *In Proceedings of the 30th VLDB, Toronto, Canada, 2004.*

[8] Agrawal, S. et al. Database Tuning Advisor in SQL Server 2005. White paper. http://www.microsoft.com/technet/prodtechnol/sql/2005/sql2005dta.mspx

[9] Agrawal,S., Chu, E., and Narasayya, V.: Automatic physical design tuning: workload as a sequence. *In Proceedings of ACM SIGMOD Conference 2006.*

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

[11] Anderson, E. et al. Hippodrome: running circles around storage administration. *Conference on File and Storage Technology (FAST'02)* pp. 175-188 (28-30 January 2002, Monterey, CA).

[12] Avnur, R., and Hellerstein, J. Eddies: Continuously Adaptive Query Processing. *In Proceedings of ACM SIGMOD Conference 2000.*

[13] Bruno, N., and Chaudhuri, S. An Online Approach to Physical Design Tuning. *Proceedings of the 2007 ICDE Conference.*

[14] Bruno N. and Chaudhuri S. To Tune or not to Tune? A Lightweight Physical Design Alerter. *In Proceedings of the VLDB Conference, 2006.*

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

[16] Bruno N. and Chaudhuri S. Physical Design Refinement. The Merge-Reduce Approach. *In Proceedings of the EDBT Conference, 2006.*

[17] Bruno, N., and Chaudhuri, S.: Exploiting statistics on query expressions for optimization. *In Proceedings of ACM SIGMOD Conference 2002*.

[18] Bruno,N, Chaudhuri, S., Gravano, L.: STHoles: A Multidimensional Workload-Aware Histogram. *In Proceedings of ACM SIGMOD Conference 2001.*

[19] Chaudhuri, S. Christensen, E., Graefe, G., Narasayya, V., and Zwilling, M.: Self-Tuning Technology in Microsoft SQL Server. IEEE Data Eng. Bull. 22(2): 20-26 (1999)Chaudhuri, S, Dageville, B., and Lohman, G.: Self-Managing Technology in Database Management Systems. VLDB 2004.

[20] Chaudhuri, S., Das, G., and Srivastava, U. Effective Use of Block-Level Sampling in Statistics Estimation. *In Proceedings of ACM SIGMOD Conference 2004*.

[21] Chaudhuri, S., Datar, M., and Narasayya V. Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution. *IEEE Transactions on Knowledge and Data Engineering, VOL. 16, NO. 11, November 2004.*

[22] Chaudhuri, S., Gupta, A., and Narasayya, V. Compressing SQL Workloads. *Proceedings of the ACM SIGMOD, Madison, USA, 2001.*

[23] Chaudhuri S., Kaushik, R, and Ramamurthy R. When Can We Trust Progress Estimators For SQL Queries? In *Proceedings of the ACM SIGMOD, Baltimore, USA, 2005.*

[24] Chaudhuri S., König, A., and Narasayya V. , SQLCM: A Continuous Monitoring Framework for Relational Database Engines. In *Proceedings of ICDE, Boston, USA, 2004*

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

[26] Chaudhuri, S. and Narasayya, V. An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server. *In Proceedings of the VLDB*, Athens, Greece, 1997.

[27] Chaudhuri, S. and Narasayya, V. AutoAdmin "What-If" Index Analysis Utility. *In Proceedings of ACM SIGMOD*, Seattle, WA, 1998.

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

[29] Chaudhuri S., and Narasayya V. Index Tuning Wizard for Microsoft SQL Server 7.0. Microsoft SQL Server 7 Technical Articles. http://msdn2.microsoft.com/en-us/library/aa226167(SQL.70).aspx

[30] Chaudhuri, S. and Narasayya, V.: Automating Statistics Management for Query Optimizers. In Proceedings of ICDE 2000.

[31] Chaudhuri S., Narasayya V., and Ramamurthy R., Estimating Progress of Execution for SQL Queries. *In Proceedings of the ACM SIGMOD, Paris, France, 2004.*

[32] Chaudhuri S. and Weikum G., Rethinking Database System Architecture: Towards a Self-tuning, RISC-style Database System . In Proceedings VLDB, Cairo, Egypt, 2000.

[33] Chaudhuri, S., and Weikum, G.: Foundations of Automated Database Tuning. In Proceedings of VLDB 2006.

[34] Chen, C., and Roussopoulos, N.: Adaptive Selectivity Estimation Using Query Feedback. SIGMOD Conference 1994.

[35] Consens, M., Barbosa, D., Teisanu, A., Mignet, L.: Goals and Benchmarks for Autonomic Configuration Recommenders. SIGMOD Conference 2005.

[36] Dageville, B., Zaït, M.: SQL Memory Management in Oracle9i. *In Proceedings of VLDB 2002*.

[37] Dageville, B., Das, D., Dias, K., Yagoub, K., Zaït, M., Ziauddin, M.: Automatic SQL Tuning in Oracle 10g. *In Proceedings of VLDB 2004.*

[38] Deshpande, A., Ives, Z., and Raman, V. Adaptive Query Processing. Foundations and Trends in Databases, 2007.
[39] El-Helw, A., Ilyas, I., Lau, W., Markl, V., and Zuzarte, C. Collecting and Maintaining Just-in-time statistics. *In Proceedings of ICDE 2007*.
[40] Finkelstein, S., Schkolnick, M., and Tiberio, P. Physical Database Design for Relational Databases. *ACM Trans. on Database Systems*, Vol 13, No 1, March 1988.
[41] Frank,M,, Omiecinski, E., Navathe, S.: Adaptive and Automated Index Selection in RDBMS. EDBT 1992.
[42] Galindo-Legaria, C., Joshi, M., and Waas, F., Wu, M.: Statistics on Views. *In Proceedings of VLDB 2003*.
[43] Graefe, G. The Cascades framework for query optimization. *Data Engineering Bulletin*, 18(3), 1995.
[44] Hammer, M. and Chan, A., Index Selection in a Self-Adaptive Data Base Management System", In Proceedings of ACM SIGMOD 1976.
[45] Idreos, S., Kersten, M., and Manegold, S.: Database Cracking. In Proceedings of CIDR 2007.
[46] Idreos, S., Kersten, M., Manegold, S.: Updating a cracked database. *In Proceedings of SIGMOD Conference 2007*.
[47] Kabra, N. and DeWitt, D. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. *In Proceedings of SIGMOD Conference 1998*.
[48] Larson, P., Goldstein, J., Guo, H., and Zhou, J: MTCache: Mid-Tier Database Caching for SQL Server. *IEEE Data Eng. Bull. 27(2): 35-40 (2004)*
[49] Lightstone, S., Teorey, T., and Nadeau, T. Physical Database Design: the database professional's guide to exploiting indexes, views, storage, and more, *Morgan Kaufmann Press*, 2007. ISBN: 0123693896.
[50] Luo, G., Naughton, J., Ellmann, C. and Watzke, M.: Toward a Progress Indicator for Database Queries. *In Proceedings of ACM SIGMOD 2004*.
[51] Luo, G., Naughton, J., Ellmann, C.,and Watzke, M.: Increasing the Accuracy and Coverage of SQL Progress Indicators. *In Proceedings of ICDE 2005.*
[52] Luo, G., Naughton, J, and Yu, P.: Multi-query SQL Progress Indicators. In Proceedings of EDBT 2006.
[53] Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H.: Robust Query Processing through Progressive Optimization. *In Proceedings of ACM SIGMOD Conference 2004.*
[54] Mishra, C., Koudas, N.: A Lightweight Online Framework For Query Progress Indicators. *In Proceedings of ICDE 2007*.
[55] Papadomanolakis,S., and Ailamaki, A.: AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. *In Proceedings of SSDBM 2004*.
[56] Rozen,S. and Shasha, D.: A Framework for Automating Physical Database Design. In Proceedings of *VLDB 1991*.
[57] Sattler, K, Geist, I. Schallehn, E. Quiet: Continuous query-driven index tuning. In Proceedings of VLDB, 2003.
[58] Schnaitter, K., Abiteboul, S., Milo, T., Polyzotis, N.: COLT: continuous on-line tuning. *In Proceedings of ACM SIGMOD Conference 2006*.
[59] Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T. Access path selection in a relational database management system. In *Proceedings of SIGMOD* 1979.
[60] Shapiro, G.P: The Optimal Selection of Secondary Indices is NP-Complete. *SIGMOD Record* 13(2): 72-75 (1983).
[61] Srivastava, U. et al. ISOMER: Consistent Histogram Construction Using Query Feedback. *In Proceedings of ICDE*, 2006.
[62] Stillger, M., Lohman, G., Markl,V., and Kandil, M.: LEO - DB2's LEarning Optimizer. *In Proceedings of VLDB 2001*.
[63] Stonebraker, M. The Choice of Partial Inversions and Combined Indices. *International Journal of Computer and Information Sciences, 3(2), June 1974.*
[64] Soror, A., Aboulnaga, A., Salem, K. Database Virtualization: A New Frontier for Database Tuning and Physical Design. *In 2nd International Workshop on Self-Managing Database Systems (SMDB 2007).*Storm, A., Garcia-Arellano, C., Lightstone, S, Diao, Y. and Surendra, M.: Adaptive Self-tuning Memory in DB2. *In Proceedings of VLDB 2006*.
[65] Tsatalos, O, Solomon, M, and Ioannidis, Y.: The GMAP: A Versatile Tool for Physical Data Independence. *In Proceedings of VLDB 1994*.
[66] Valentin, G., Zuliani, M., Zilio, D., Lohman, G. and Skelley, A. DB2 Advisor: An Optimizer Smart Enough to Recommend its Own Indexes. *In Proceedings of ICDE*, San Diego, USA, 2000.
[67] Weikum,G., Mönkeberg, A., Hasse, C., and Zabback, P.: Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. *In Proceedings of VLDB 2002*.
[68] Weikum, G., Hasse,C., Moenkeberg, A., Zabback, P.: The COMFORT Automatic Tuning Project, Invited Project Review. Inf. Syst. 19(5): 381-432 (1994)
[69] Zilio et al. DB2 Design Advisor: Integrated Automatic Physical Database Design. *In Proceedings of the VLDB*, Toronto, Canada, 2004.
[70] Zilio et al. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. *In Proceedings of ICAC 2004.*