AutoAdmin: Self-Tuning Database Systems Technology

Sanjay Agrawal, Nicolas Bruno, Surajit Chaudhuri, Vivek Narasayya
Microsoft Research

1 Introduction

The AutoAdmin research project was launched in the Fall of 1996 in Microsoft Research with the goal of making database systems significantly more self-tuning. Initially, we focused on automating the physical design for relational databases. Our research effort led to successful incorporation of our tuning technology in Microsoft SQL Server and was subsequently also followed by similar functionality in other relational DBMS products. In 1998, we developed the concept of self-tuning histograms, which remains an active research topic. We also attempted to deepen our understanding of monitoring infrastructure in the relational DBMS context as this is one of the core foundations of the "monitor-diagnose-tune" paradigm needed for making relational DBMS self-tuning. This article gives an overview of our progress in the above three areas – physical database design, self-tuning histograms and monitoring infrastructure.

2 Physical Database Design Tuning

One great virtue of the relational data model is its data independence, which allows the application developer to program without worrying about the underlying access paths. However, the performance of a database system crucially depends on the physical database design. Yet, unlike widespread use of commercial tools for logical design, there was little support for physical database design for relational databases when we started the AutoAdmin project. Our first challenge was to understand the physical design problem precisely. There were several key architectural elements in our approach.

Use of Workload: The choice of the physical design depends on the usage profile of the server, so we use a representative workload (defined as a set of SQL DML statements such as SELECT, INSERT, DELETE and UPDATE statements), as a key input to the physical design selection problem. Optionally, a weight is associated with each statement in the workload. A typical way to obtain such a workload is to use the monitoring capabilities of today's DBMSs that allow capture of SQL statements, which execute on the server over a representative time period to a trace file (see Section 4). In some cases, a representative workload can be derived from an appropriate organization or industry specific benchmark.

Optimizer "in the loop": When presented with a query, the database query optimizer is responsible for deciding which available physical design structures (e.g. indexes or materialized views) to use for answering the query. Therefore, it is crucial to ensure that the recommendations of an automated physical design selection

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

tool are in-sync with the decisions made by the optimizer. This requires that we evaluate the goodness of a given physical design for a database using the same metric as optimizer uses to evaluate alternative execution plans, i.e., the optimizer's cost model. The alternative approach of building an external, stand-alone simulator for physical design selection does not work well for both accuracy and software engineering reasons (see [13]). This approach was first adopted in DBDSGN [19], an experimental physical design tool for System R.

Creating "what-if" physical structures: The naïve approach of physically materializing each explored physical design alternative and evaluating its goodness clearly does not scale well, and is disruptive to normal database operations. A "what-if" architecture [14] enables a physical design tool to construct a configuration consisting of existing as well as *hypothetical* physical design structures and request the query optimizer to return the best plan (with an associated cost) if the database were to have that configuration. This architecture is possible because the query optimizer does not require the presence of a fully materialized physical design structure (e.g., an index) in order to be able to generate plans that use such structure. Instead, the optimizer only requires *metadata* entries in the system catalog and the relevant statistics for each hypothetical design structure, often gathered via sampling. The server extensions that support such a "what-if" API are shown in Figure 1.

These three foundational concepts allow us to precisely define physical design selection as a *search* problem of finding the best set of physical structures (or *configuration*) that fits within a provided storage bound and minimizes the optimizer-estimated cost of the given workload.

Challenges in Search for a Physical Design

We now outline the key challenges for solving the physical design selection problem as defined above.

Multiple physical design features: Modern DBMSs support a variety of indexing strategies (e.g., clustered and non-clustered indexes on multiple columns, materialized views). Furthermore, indexes and materialized views can be horizontally partitioned (e.g., range, hash partitioning). The choices of physical design features strongly *interact* with each other [4]. Thus, an integrated approach that considers all physical design features is needed, which makes the search space very large.

Interactions due to updates and storage: The physical design recommendation for a query may have a significant impact on the performance of other queries and updates on the database system. For example, an index recommended to speed up an expensive query may cause update statements to become much more expensive or may reduce the benefit of another index for a different query in the workload. Thus, good quality recommendations need to balance benefits of physical design structures and their respective storage and update overheads [15].

Scaling Challenges: Representative workloads generated by enterprise applications are often large and consist of complex queries (e.g., in the order of hundreds of thousands of statements). Furthermore, the database schema and the table sizes themselves can be large. Thus, to be usable in an enterprise setting, physical design tuning tools must be designed to scale to problem instances of this magnitude.

Key Ideas for Efficient Search

Table Group and Column Group pruning using frequent itemsets: The space of physical design structures that needs to be considered by a physical design tool grows exponentially with the number of tables (and columns) that are referenced in any query. Therefore it is imperative to prune the search space early on, without compromising the recommendation quality. Often, a large number of tables (and columns) are referenced infrequently in the workload. For many of them, any indexes or materialized views on such tables (and columns) would not have a significant impact on the cost of the workload. Therefore, we use a variation of frequent itemset techniques to identify such tables (and columns) very efficiently and subsequently eliminate these from consideration [2, 4].

Workload compression: Typical database workloads consist of several instances of parameterized queries. Recognizing this, and appropriately compressing the input workload [10] can considerably reduce the work done during candidate set generation and enumeration.

Candidate set generation: A physical structure is considered a candidate if it belongs to an optimal (or close to optimal) configuration for at least one statement in the input workload. The approach described in [13] generates this candidate set efficiently. A more recent work [5] discusses how to instrument the optimizer itself to efficiently generate the candidate set.

Merge and Reduce: The initial candidate set results in an optimal (or close-to-optimal) configuration for queries in the workload, but generally is either too large to fit in the available storage, or causes the updates to slow down significantly. Given an initial set of candidates for the workload, the *Merge* and *Reduce* primitives [15, 2, 5, 6] augment the set with additional indexes and materialized views that have lower storage and update overhead while sacrificing very little of the querying advantages. For example, if 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), while sub-optimal for each Q_1 and Q_2 can be optimal for the workload if there is insufficient storage to build both indexes.

Enumeration: The goal of enumeration is to find the best configuration for a workload from the set of candidates. As described earlier, the choice of various structures interacts strongly with each other and this makes the enumeration problem hard. We have explored two alternative search strategies: top-down [5] and bottom-up [13, 2] enumeration, each of which has relative merits. The top-down approach can be efficient in cases where the storage bound is large or the workload has few updates. In contrast, the bottom-up search can be efficient in storage constrained or update intensive settings.

Customizing Physical Design Selection

Exploratory "what-if" analysis: Experienced DBA's often want the ability to propose different hypothetical physical design configurations and explore the impact of the proposed configuration for a given workload (which statements speeded up or slowed down, and by how much etc.) [14, 3].

Incremental refinement of physical design: Changes in data statistics or usage patterns can introduce redundancies and may make a very well tuned physical design inappropriate over time. However, physical design changes can have a significant overhead (e.g., existing query plans can get invalidated, which is not be desirable on production servers). In such cases, one would like to compact the existing physical design in an incremental manner, without significantly sacrificing performance. Reference [6] describes such a technique that starts from the initial configuration, and progressively refines it using the *merge* and *reduce* primitives until some property is satisfied (e.g., the configuration size or its performance degradation meets a pre-specified threshold).

Constrained physical design selection: DBAs often need the ability to specify a variety of constraints on the physical design (e.g, which tables to tune, or which existing indexes to keep) [3]. An important constraint that impacts manageability is that indexes are partitioned identically as the underlying table (i.e. *aligned*).

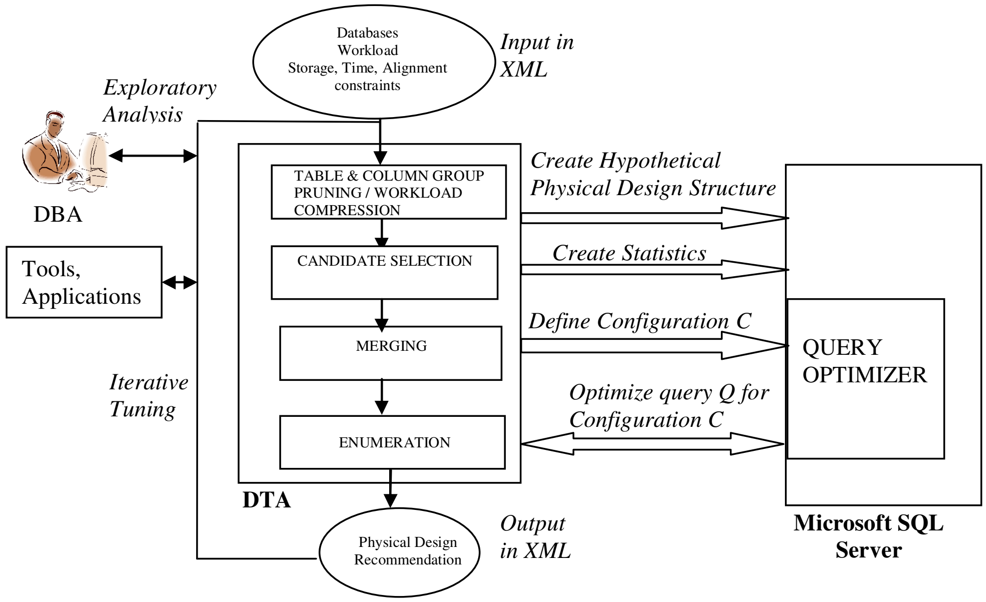

Figure 1: Overview of the Database Engine Tuning Advisor (DTA).

In this case, common operations like per-partition backup/restore and data load/remove become much easier. However as shown in [4], database system performance can be significantly impacted by the choice of a specific partitioning strategy. Therefore, a DBA may still need to find the best physical design where all the indexes are required to be aligned.

When to Invoke Physical Design Tuning?

Data and workload characteristics can change over time. The usage modes described thus far assume that the DBA knows when to invoke a physical design tuning tool. Since physical design tuning has an associated overhead (and impacts the underlying database engine performance), it is useful to identify a priori whether or not physical design tuning on the database can significantly improve performance. We built a lightweight tool, called *Alerter*, that identifies when the current physical design has the opportunity to be improved. It does not make any additional optimizer calls; rather it piggybacks on the optimizer when the latter generates the query plan (see [7] for details).

Product Impact

The AutoAdmin research work on physical database design tuning resulted in a tool called the Index Tuning Wizard that shipped with Microsoft SQL Server 7.0 in 1998. The tool was the first of its kind among commercial relational database systems. It used many of the key building blocks described above, including the "what-if" architecture (which required extending the SQL Server optimizer), candidate selection, merging and bottom-up enumeration. Microsoft added support for materialized (or indexed) views in the SQL Server 2000 release. Our work on table group pruning and view merging in Index Tuning Wizard enabled us to provide efficient, integrated recommendation for indexes and materialized views. In SQL Server 2005, our work resulted in a tool, called the Database Engine Tuning Advisor (DTA) that replaced and significantly expanded the scope and usability of Index Tuning Wizard. DTA can provide integrated recommendations for range partitioning in addition to indexes and materialized views. Furthermore, it incorporates some of the new usage modes described above such as: (a) partitioning "alignment" constraint (b) exploratory "what-if" analysis. An overview of the architecture of DTA is shown in Figure 1 (see [3] for more details).

3 Self-Tuning Histograms: Exploiting Execution Feedback

Query optimization in DBMSs has traditionally relied on single column histograms to estimate selectivity values. Despite the fact that several proposals for multi-dimensional histograms have been put forward [21, 22, 20] to address obvious inaccuracies in estimating multiple selection conditions on different columns using single-column histograms, none is presently supported among leading relational database products. This is partly explained by the fact that like single-column histograms, multi-dimensional histograms implicitly assume that all multi-dimensional queries are equally likely. This is rarely true in practice and this incorrect assumption has much more adverse impact on multi-dimensional histograms than single column histograms.

The above observation led the AutoAdmin team to propose the notion of a *self-tuning histogram*. The key intuition in self-tuning histogram is to use the query workload as a key driver in defining the structure of the multi-dimensional histogram. Self-tuning histograms are incrementally built up by exploiting workload information and query execution feedback. Intuitively, we exploit query workloads to zoom in and spend more resources in heavily accessed areas while allowing some inaccuracy in the rest. We exploit query execution feedback in a truly multidimensional way to: (i) identify promising areas to enclose in histogram buckets, (ii) detect buckets that do not have uniform density and need to be "split" into smaller and more accurate buckets, and (iii) collapse adjacent buckets that are too similar thus recuperating space for more critical regions. Self-tuning histograms can gracefully adapt to changes in the data distribution they approximate, without the need to periodically rebuild them from scratch. Additionally, some data sources might only expose their values through queries (e.g., web-services), and thus traditional techniques, which require the complete data set to proceed, are of little or no value.

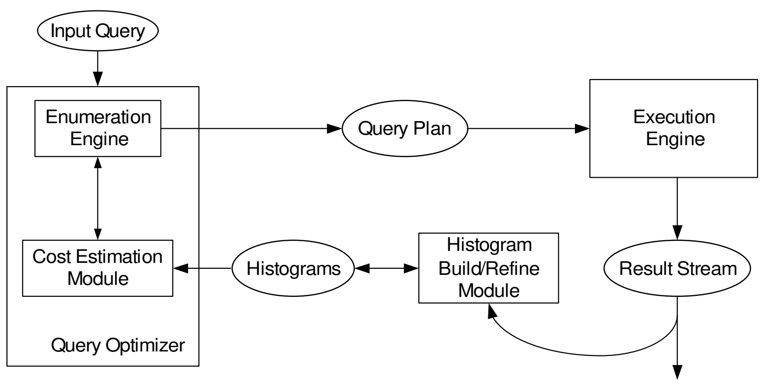

Figure 2: Self-tuning histograms.

Figure 2 shows schematically how to maintain self-tuning histograms. For each incoming query, the optimizer exploits existing histograms and produces an execution plan. The resulting execution plan is then passed to the execution engine, where it is processed. A build/refine histogram module monitors the query execution feedback and diagnoses whether the relevant buckets are accurate enough. If that is not the case, the corresponding histogram buckets are tuned so that the refined histogram becomes more accurate for future similar queries.

To define a self-tuning histogram, we need to address three key issues: the multidimensional structure that holds histogram buckets, the monitoring component that gathers query execution feedback, and the tuning procedures that restructure histogram buckets. Our first attempt at self-tuning histograms was STGrid histograms [1] where we (i) greedily partition the data domain into disjoint buckets that form a grid, (ii) monitor query results by aggregating coarse information that is used to refine bucket frequencies, and (iii) periodically restructure buckets by merging and splitting rows of buckets at a time (to preserve the grid structure). Later, in reference [8] we introduced STHoles histograms, which use a novel partitioning strategy that is better suited to represent multi-dimensional structures. Specifically, some buckets can be completely included inside others. This way, we implicitly relax the requirement of rectangular regions while keeping rectangular bucket structures. STHoles histograms gather query execution feedback on a per-bucket level, and can refine individual buckets for better accuracy.

While STGrid focuses on efficiency by monitoring aggregated coarse information, STHoles focuses on accuracy at the expense of a more heavyweight monitoring. In many cases, STHoles histograms are more accurate for the expected workload than alternative techniques that require multiple scans over the whole data set. Recently, reference [23] introduces ISOMER, a variation of STHoles histograms that balances accuracy and monitoring overhead. By using a lightweight monitoring mechanism and applying the maximum entropy principle to refine buckets, ISOMER provides a good middle-ground between the faster but less accurate STGrid histograms and the relatively slower but more accurate STHoles histograms.

4 Monitoring the Database Server

Our goal for making the database systems self-tuning requires the ability to observe and analyze the "state" of the server often over a period of time. The previous two sections of this article point to the importance of capturing the query workload as well as execution feedback. Despite the clear benefit of monitoring the state of the server, database systems have traditionally been limited in their support for monitoring. Therefore, the AutoAdmin project considers this an important area of further exploration. In this section, we summarize some of our progress in this area, after reviewing the current state of the art for DBMS monitoring infrastructure.

Today's relational database systems support two basic monitoring mechanisms in addition to those provided by the operating system. The first exposes a *snapshot of counters* that captures current database state. These counters can be obtained at any point in time by polling the server via system defined views/functions. For example, in Microsoft SQL Server 2005, these snapshots can be obtained by Dynamic Management Views or *DMVs* (`www.msdn.microsoft.com`). The second mechanism, which we refer to as *event recording*, allows system counters to be logged to a table/file whenever a pre-specified event occurs. For example, in Microsoft SQL Server 2005, the Profiler provides such event recording functionality. Both these mechanisms form the basis of diagnostic and tuning tasks (e.g., DTA uses Profiler, DMVs can be used to diagnose performance bottlenecks such as excessive blocking). We now highlight two pieces of work on the monitoring infrastructure.

Query Progress Estimation

Consider the problem of *estimating progress* of a currently executing query. An estimate of the percentage completed for a query is useful to DBAs for many reasons (e.g., to decide whether to kill the query to free up resources and for admission control decisions). However, this problem is significantly more challenging than the common problem of measuring progress of a file download. In particular, unlike the file download example, it is not always possible to ensure that *estimated* progress monotonically increases over time without compromising accuracy.

It is important to recognize that since query progress estimation will be used for monitoring, such estimation must be computationally lightweight and yet be able to capture progress at fine granularity (being accurate at only at 0% and 100% is trivial and uninteresting). In solving this monitoring problem, our first challenge is to define a meaningful metric for work done by a query. For example, the count of answer tuples is not a good indicator since there could be one or more blocking operators in the execution plan. Next, we must provide estimators that rely on the model of work done for doing robust, lightweight estimation.

Metric for Work done: The requirements for modeling the work done for progress estimation are different from those of the query optimizer, which uses its cost model to compare alternative execution plans. Estimation of progress requires the ability to incorporate execution feedback and progressively refine the a priori estimation, obtained initially using the optimizer's model. These considerations lead us to a different metric for work, as explained in [16]. We observe that the operators in a query execution plan are typically implemented using a demand-driven iterator model, where each physical operator in the execution plan supports Open(), Close() and GetNext(). We model the work done by the query as the total number of GetNext() calls executed in the plan, which can satisfy the requirements of a progress estimator mentioned above.

Estimators: The above metric for work leads to the natural definition of an idealized measure for progress as $\sum K_i / \sum N_i$, where K_i is the number of GetNext() calls executed by operator Op_i thus far, and N_i is the total number of GetNext() calls that will be executed by operator Op_i when the query execution is complete. While K_i is easily measured as the query is executing, this is not so for N_i. Thus, the key challenge is to estimate N_i as accurately as possible *while the query is executing*. The work in [11] analyzes the characteristics of the progress estimation problem from the perspective of providing robust, worst-case guarantees. Despite the fact that we have to contend with a negative result in the general case, for many common scenarios it is possible to design effective progress estimators with bounded error. For example, if we assume input tuples arrive in random order, then measuring progress at the leaf nodes that "drive" the execution of the pipeline by supplying tuples to the other nodes (e.g, table or index scans), can provide robust estimation of progress for the entire query [16, 11].

SQLCM: A Continuous Monitoring Infrastructure

Beyond ensuring that we have the right plumbing to monitor status of the server (e.g., query progress monitoring), another key challenge is that of tracking and aggregating changes in one or more selected counters over time, aggregating from multiple counters that are being monitored, or a combination of both. For example, consider the task of detecting instances of a stored procedure that are 3 or more times slower than the historical average execution time of the stored procedure. If we use event recording, then a very large volume of monitored data needs to be written out by the server (all stored procedure completion events). On the other hand, if we use the mechanism of repeatedly polling the server using DMVs, we could compromise the accuracy of answers obtained if we do not poll frequently enough (i.e., miss outliers). If instead, we poll too frequently, then we may impose significant load on the server. Thus, neither of the prevalent mechanisms provides adequate support for handling the above task – what we need is a lightweight server-side mechanism to aggregate events generated by the monitored counters (also referred to as *probes* in this section).

These requirements led us to build the SQLCM prototype (Figure 3) [12], with the following characteristics. First, it is implemented inside the database server. Second, monitoring tasks can be specified to SQLCM in a declarative manner using a simple class of Event-Condition-Action (ECA) rules. A rule implicitly defines what conditions need to be monitored (e.g., an instance of a stored procedure executes 3 times slower than the average instance, a statement blocks others for more than 10 seconds) and what actions need to be taken (e.g., report the instance of the stored procedure to a table, cancel execution of the statement). Third, the monitored information can be automatically grouped and aggregated based on the ECA rule specifications. This grouping and aggregation can be done very efficiently using an in-memory data structure called the lightweight aggregation table (LAT). Consequently, the volume of information that needs to be written out by the server is small, thus dramatically reducing the overheads incurred on the server by the monitoring tasks. SQLCM only incurs monitoring overhead that is necessary to implement currently specified rules (see details in [12]).

5 Conclusion

As part of the AutoAdmin research project, we have had the opportunity to address several significant challenges that relate to endowing the relational database system with increased self-tuning capabilities. These self-tuning capabilities rely on a monitoring infrastructure and leverage that to build specialized diagnostic and

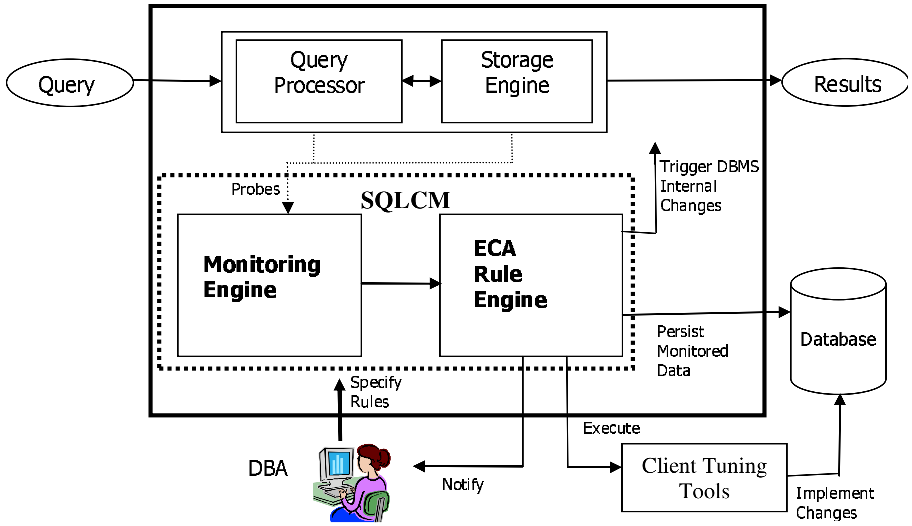

Figure 3: Architecture of SQLCM.

tuning capabilities that are appropriate to the task at hand. Thus, they conceptually share a common *"monitor-diagnose-tune"* pattern. As new queries are executed, the DBMS internally monitors and keeps information about the workload. After a triggering condition happens (e.g., a fixed amount of time, an excessive number of recompilations, significant database updates), the diagnostics component is launched automatically and evaluates the situation quickly. After the lightweight diagnostics, if it is determined that the database needs to change, a tuning component proceeds to recommend/incorporate changes for better performance. The diagnostics and tuning components are typically specific to each task. However, the monitoring component can be shared by multiple "vertical" diagnose-tuning components. Due to lack of space, we have only highlighted a few selected aspects of the AutoAdmin project. Information about other work done in the AutoAdmin project can be found at `research.microsoft.com/dmx/AutoAdmin`.

Since launching of the AutoAdmin effort, there has been increased awareness of the need to reduce the total cost of ownership of database systems and several initiatives in other research groups and database vendors have helped contribute to the development of self-tuning technology [9, 18]. Finally, our experience over the last decade has also convinced us that today's relational database architecture may in fact stand in the way of robust self-tuning capability. Specifically, recognizing the trade-off between adding features and providing self-tuning capability requires careful thinking [17].

Acknowledgments

Over the years, the AutoAdmin project has received significant contributions from current and past project members. Manoj Syamala was a key member of the development team of DTA for Microsoft SQL Server 2005. Christian König was a significant contributor to the SQLCM project. Raghav Kaushik and Ravi Ramamurthy led our work on the query progress estimation problem. AutoAdmin has also greatly benefited from the interns and visitors who worked with us diligently. We thank Ashraf Aboulnaga, Ashish K. Gupta and Gerhard Weikum for their contributions to AutoAdmin work referenced in this article.

References

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *Proceedings of the ACM International Conference on Management of Data*, 1999.

[2] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2000.

[3] S. Agrawal et al. Database Tuning Advisor for Microsoft SQL Server 2005. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, 2004.

[4] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[5] N. Bruno and S. Chaudhuri. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[6] N. Bruno and S. Chaudhuri. Physical design refinement: The "Merge-Reduce" approach. In *International Conference on Extending Database Technology (EDBT)*, 2006.

[7] N. Bruno and S. Chaudhuri. To tune or not to tune? A Lightweight Physical Design Alerter. In *Proceedings of the 32th International Conference on Very Large Databases (VLDB)*, 2006.

[8] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A multidimensional workload-aware histogram. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2001.

[9] S. Chaudhuri, B. Dageville, and G. M. Lohman. Self-managing technology in database management systems (tutorial). In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, 2004.

[10] S. Chaudhuri, A. K. Gupta, and V. R. Narasayya. Compressing sql workloads. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2002.

[11] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. Can We Trust Progress Estimators For SQL Queries? In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[12] S. Chaudhuri, A. C. König, and V. R. Narasayya. SQLCM: A Continuous Monitoring Framework for Relational Database Engines. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2004.

[13] S. Chaudhuri and V. Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*, 1997.

[14] S. Chaudhuri and V. Narasayya. Autoadmin 'What-if' index analysis utility. In *Proceedings of the 1998 ACM International Conference on Management of Data (SIGMOD)*, 1998.

[15] S. Chaudhuri and V. Narasayya. Index merging. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1999.

[16] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Estimating Progress of Execution for SQL Queries. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2004.

[17] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2000.

[18] S. Chaudhuri and G. Weikum. Foundations of automated database tuning (tutorial). In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2005.

[19] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Trans. Database Syst.*, 1988.

[20] D. Gunopulos et al. Approximating multi-dimensional aggregate range queries over real attributes. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2000.

[21] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multidimensional queries. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1988.

[22] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB)*, 1997.

[23] U. Srivastava et al. ISOMER: Consistent histogram construction using query feedback. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2006.