Bulletin of the Technical Committee on

Data Engineering

December 2011 Vol. 34 No. 4

IEEE Computer Society

Letters

Special Issue on Self-Managing Databases: Lessons Learned and Challenges Ahead

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TC on Data Engineering web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

IEEE CS Governance

Beginning in the June Bulletin, and continuing with the September issue, I have been keeping you abreast of a proposed change in how a part of the IEEE Computer Society governs itself. In particular, a governance subcommittee of the Technical Activities Committee (TAC) was formed to propose changes in how its chair is selected. All chairs of Technical Committees (TCs) are members of this committee, and I, as chair of the TC on Data Engineering thus attend its meetings. Under the current rules, the chair is appointed, but under the proposal from the governance committee, the chair would be elected by TAC members.

I am happy to report that the TAC members have now voted to approve this proposal. So now the proposal goes forward to the Computer Society Board of Governors. We are hopeful that they will approve, and if so, the proposal becomes the new way of choosing the TAC chair. I belonged to the subcommittee and am a firm believer in the proposed change, as it gives TCs a way to have their views more effectively expressed, both in our TAC meeting and because the TAC chair represents the TCs at higher levels of the Computer Society. So we are not done yet, but progress has been made. Stay tuned.

The Current Issue

Database systems have been notoriously cranky to deal with, requiring skilled professionals to manage and keep tuned. The research into self-tuning database systems, while not eliminating all such manual involvement, reduced the level of effort considerably. This has permitted the classic database market to service much smaller enterprises than it had earlier. And, of course, it made life simpler for many more enterprises, large and small.

This ”self tuning” research has been going on now for over ten years. The result has been that the ”friendliness” of databases has improved greatly. But it is still the case that databases can be a challenge to deal with. So interest in the field has remained high, leading to the establishment, a few years back, of the Data Engineering Workgroup on Self-Managing Database Systems within the Technical Committee on Data Engineering.

The current issue has been assembled by a pair of editors from the workgroup, Shivnath Babu and Kai-Uwe Sattler. They know well the self-managing databases research community, the researchers, the practitioniers, and their work, and they have successfully brought together a blend of papers that reflect this diversity, with a heavier dose of industrial participation than usual. This is a reflection that this topic has commercial importance to the database industry, and hence is the kind of issue that might only come from the Data Engineering Bulletin. I want to thank Shivnath and Kai-Uwe for their efforts in bringing this fine issue to fruition.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editors

The demand for compute and storage resources in enterprises, science, and society is increasing rapidly. New scientific discoveries are expected to come more and more from the analysis of large datasets such as those generated by particle accelerators and genome sequencers. Cloud platforms with hundreds of thousands of servers are now available for public use. It is impossible to sustain the voracious appetite for compute and storage resources in cost-effective ways without extensive automation of system management. The database community has always been at the forefront of research on auto-tuning and self-managing systems in order to address this challenge. This issue of the Data Engineering Bulletin contains a collection of articles that covers what the database community has learned from a decade of work on self-managing database systems as well as the upcoming challenges in this field.

The first three articles describe some lessons learned over the course of a decade in the process of incorporating automation into database management. Abouzour, Bowman, Bumbulis, DeHaan, Goel, Nica, Paulley, and Smirnios describe their experiences while developing self-managing technologies in Sybase SQL Anywhere. Unlike the Microsoft SQL Server and Oracle database servers considered in the following two articles, Sybase SQL Anywhere is usually embedded along with applications. The article by Bruno, Chaudhuri, König, Narasayya, Ramamurthy, and Syamala gives an overview of the AutoAdmin project; one of the very first projects that implemented automated management techniques in a commercial database system. AutoAdmin started with a focus on physical design tuning, and then expanded in scope to include many other aspects of database tuning. The article by Belknap, Beresniewicz, Dageville, Dias, Shaft, and Yagoub describes the holistic approach taken to improve manageability in the Oracle database system. Three levels with increasing complexity of automation are described: monitoring and reporting, recommendations, and automated actions.

The next two articles present initial solutions to some hard technical challenges that arise while automating database management. Benchmarking and testing self-managing techniques pose hurdles that can limit the widespread use of self-managing systems. Jimenez, LeFevre, Polyzotis, Sanchez, and Schnaitter describe their research efforts in this direction. They present a performance benchmark for online index tuning, and then use the benchmark to study three different online index tuning algorithms from the literature. Adaptive processing techniques for large-scale systems are presented in the paper written by Balmin, Ercegovac, Vernica, and Beyer. They discuss user-defined aggregates such as top-k processing in Jaql – a high-level declarative language for MapReduce – and propose adaptive optimization techniques for computing these aggregates.

The last three articles give an overview of new challenges that self-managing database systems face as they are used in new application domains. Heinis, Branco, Alagiannis, Borovica, Tauheed, and Ailamaki consider the domain of scientific databases in high-energy physics and neuroscience. They describe the specific data management problems in these areas and discuss challenges related to physical database design. Statistical models are usually present at the heart of self-managing approaches for complex systems. However, limited research has been done on problems like how to build these models, maintain them over time, and use the possibly-uncertain estimates generated by these models. Chen, Ganapathi, and Katz discuss the increasing complexity of modeling systems and workloads in large-scale data processing. They describe the process of defining the system boundary, building and evaluating a statistical model for the system, and turning the statistics into knowledge needed for tuning the system. Finally, Mukerjee, Talius, Kalhan, Ellis, and Cunningham present SQL Azure – Microsoft's Cloud-based database service. After a brief overview of the architecture and features, the authors discuss several lessons they have learned while developing and running the service. Based on these lessons, they discuss challenges related to self-management and fault tolerance that Cloud services will face.

We cordially thank David Lomet and all the authors who graciously contributed their time and effort to this special edition. We hope that you will find this edition thought-provoking and enjoyable.

Shivnath Babu and Kai-Uwe Sattler
Duke University (Babu) and Ilmenau University of Technology (Sattler)

Database Self-Management: Taming the Monster

Mohammed Abouzour *Ivan T. Bowman* *Peter Bumbulis* *David E. DeHaan*
Anil K. Goel *Anisoara Nica* *G. N. Paulley* *John Smirnios*
Sybase, An SAP Company

Abstract

We describe our experience in the development of self-managing technologies in Sybase SQL *Anywhere, a full-featured relational database system that is commonly embedded with applications developed by Independent Software Vendors (*ISV*s). We illustrate how* SQL *Anywhere's embeddability features work in concert to provide a robust data management solution in zero-administration environments.*

1 Introduction

Sybase SQL Anywhere is a full-function, ISO SQL standard-compliant relational database server designed to work in a variety of frontline environments, from traditional server-class back-office installations to handheld devices running the Windows CE operating system. SQL Anywhere includes features common to enterprise-class database management systems, such as intra-query parallelism, materialized views, OLAP functionality, stored procedures, triggers, full-text search, and hot failover. SQL Anywhere supports a variety of 32- and 64-bit hardware platforms, including Microsoft Windows; various flavours of UNIX including Linux, Solaris, AIX, HP-UX, and Apple OS/X; and Windows CE. However, the strength of SQL Anywhere is in its self-managing technologies, which allows the server to offer SQL data management, query processing, and synchronization capabilities in zero-administration environments.

In this paper, we describe our experience with some of the self-managing technologies present in SQL Anywhere. These self-managing technologies are fundamental components of the server, not merely administrative add-ons that assist a database administrator in configuring the server's operation—since in embedded environments there is often no trained administrator to do so. These technologies work in concert to offer the level of self-management and adaptiveness that embedded application software requires. It is, in our view, impossible to achieve effective self-management by considering these technologies in isolation.

2 Query Optimization

Certainly one of the most significant areas within the SQL Anywhere server that concerns self-managing behaviour is the SQL Anywhere query optimizer [1; 2; 3]. A significant portion of SQL Anywhere revenue stems from Independent Software Vendors (ISV's) who embed SQL Anywhere with their applications, some of which are deployed on thousands, and in some cases millions, of computers. These types of deployments typically share the following characteristics:

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- There is no DBA available to 'tune' the application, to run workload analysis tools, perform capacity planning analysis, or monitor the system.
- The database workload is mixed, with both OLTP-like transactions and reporting queries mixed together. The nature of embedded applications renders workload partitioning, or sharding, moot.
- There is little correlation between the size of the database instance and the size of the schema, or the complexity of the application's queries. SQL queries with a join degree of 15-to-20 are routine. SQL queries with 50 or more total quantifiers (tables, views, or derived tables) are common enough to be unsurprising.
- In turn, there is often little correlation between a query's complexity and its execution time. In some cases, even complex queries can execute very quickly, particularly over small instances. In such cases, the optimization process must be very efficient. For these queries, the optimizer should not spend ten seconds (for example) optimizing a query that will run in under one second unless this time can be amortized over many executions.
- The database server executes concurrently with other applications on the same computer. Hence the server must co-operate with these applications for limited resources, particularly memory.
- The system hardware, operating system platform, and application workload characteristics, including peak-to-average ratios, may vary from installation to installation, making *a priori* application tuning unhelpful.

Moreover, a significant trend in application development over the past several years is the use of Object-Relational Mapping tools, such as Hibernate and NHibernate, and complex application development environments such as Microsoft Entity Framework, that map a relational schema into object-oriented abstractions within the application [4]. As a result, these frameworks can generate a mix of both simple and complex queries whose construction is largely outside the control of the application developer. Hence the ability to syntactically annotate SQL statements with tuning parameters is largely absent with the use of these frameworks.

As a result of the above constraints, we 'tamed the monster' by engineering the SQL Anywhere optimizer to be relatively inexpensive—in terms of both memory and CPU consumption—so that each SQL request can be optimized while taking into account the server context at that particular moment [5]. Server context includes the amount of available query memory for query execution, the number of available worker threads for intra-query parallelism, and the approximate contents of the server's buffer pool which directly influences I/O cost. One of SQL Anywhere's enumeration methods is a branch-and-bound algorithm [1], which is cheap due to careful engineering to minimize memory allocation and aggressively prune sub-optimal execution plans.

Over time, and due in part to the ever-increasing sophistication of SQL Anywhere applications—particularly those that involve synchronization—along with the ever-expanding SQL language and the explosion in the use of Object-Relational mapping tools, SQL Anywhere's query optimizer has become increasingly more sophisticated—and self-adapting. Included in the optimization process are sophisticated static and cost-based query rewrite optimizations [6; 7], including those for optimizing queries over materialized views [3]; the detailed analysis of a wide range of physical query execution operators; sophisticated selectivity and cardinality estimation; and an exploration of a space of possible plans that include parallelization. Our server software has been spectacularly successful at meeting the needs of our customers, whether they deploy SQL Anywhere server software on 64-bit installations of Windows 2008 Server, or the identical server software on Windows CE handheld devices.

Experience. However, despite the fact that the SQL Anywhere optimizer is relatively cheap, with this greater sophistication comes longer code paths, and consequently greater overhead, with each new SQL Anywhere release. `OPEN` time for a request is the critical measure for many applications, which includes the time to

construct the statement's necessary abstractions within the server in addition to optimization time *per se.* As with other commercial relational database systems, we too have tried to mitigate this additional overhead, in largely two ways. The first technique is execution plan caching, which amortizes the optimization cost over multiple invocations of the same SQL statement. The second is 'bypass' optimization for 'simple' SQL statements where the best execution strategy can be determined heuristically.

These mitigation techniques have been difficult to 'get right' in the sense that errors in selecting which plans to cache (or which cached plans to abandon), or precisely which queries will be optimized heuristically, lead to inconsistencies in query execution times. This inconsistency, coupled with a cost model that takes server context into account, yields another 'monster' that we term the 'dancing optimizer', named after the stereotypical dancing Russian circus bear.

We are actively researching new optimization techniques to tame this monster. The usage patterns of SQL Anywhere described above dictate that the optimization process continue to adapt to available resources, query complexity, and estimated query cost. Join enumeration algorithms can be extremely expensive for complex queries—for example, the exhaustive enumeration of the space of bushy trees—regardless of how small the estimated runtime is. Moreover, the memory consumption of these algorithms may be prohibitive. One important lesson learned from designing the SQL Anywhere optimizer is that there is no ideal join enumeration algorithm which will perform well—that is, generate good-quality access plans in a CPU- and memory-efficient way—for all types of queries, or available system resources. A robust design is to have a set of join enumeration algorithms to choose from when optimizing a particular query, based on the available system resources, query complexity, and its estimated runtime [2]. Query complexity, for example, can be used to predict the memory needed for the memoization process as well as the CPU required for enumeration. Efficient cost-based and heuristic pruning should be employed to adjust the optimization time to the expected query cost.

In addition, we are researching diagnostic tools with which to support the analysis of performance issues in the field. One novel capability of the SQL Anywhere optimizer is to log the set of *rejected* plans for a given SQL statement, so that these may be compared with the plan that was eventually chosen [8]. We are also continuing to research techniques that will permit the diagnosis of performance issues, due to sub-optimal execution plans chosen by the optimizer, when the customer's database is unavailable (for example, due to regulatory restrictions). We have augmented the execution plan information output by the query optimizer to contain information ranging from the cost-model computations to the cost-based and heuristic decisions made by the optimizer during join enumeration. These traces of the exploration of the search space can be analyzed in isolation, without the customer database, and also can be used to generate synthetic data to simulate the original database instance.

3 Self-managing Statistics

SQL Anywhere has used query feedback techniques since 1992 to automatically gather statistics during query processing. Rather than require explicit generation of statistics, by scanning or sampling persistent data, the server automatically collects statistics as part of query execution. More recent work by other researchers [9; 10; 11; 12; 13] has also exploited this notion of collecting statistics as a by-product of query execution.

In its early releases, SQL Anywhere computed frequent-value statistics from the evaluation of equality and `IS NULL` predicates, and stored these statistics persistently in the database for use in the optimization of subsequent queries. Later, support was added for inequality and `LIKE` conditions. An underlying assumption of this model is that the data distribution is skewed; values that do not appear as a frequent-value statistic are assumed to be in the 'tail' of the distribution.

Today, SQL Anywhere uses a variety of techniques to gather statistics and maintain them automatically. These include index statistics, index probing, analysis of referential integrity constraints, three types of single-column self-managing histograms, and join histograms.

Histogram implementation. SQL Anywhere histograms are equi-depth histograms whose number of buckets expand and contract dynamically as column distribution changes are detected. As is typical, we use the *uniform distribution assumption* when interpolating within a bucket. SQL Anywhere histograms combine traditional buckets with frequent-value statistics, known as 'singleton buckets'. A value that constitutes at least 1% or 'top N' of the values is saved as a singleton bucket. The number of singletons retained in any histogram depends on the size of the table and the column's distribution, but lies in the range [0,100]. An additional metric, which we term *histogram density,* is maintained for each histogram. Histogram density is an estimate of the average selectivity of all values in the column that are not covered by singleton buckets. In other words, the histogram density is the average selectivity of individual values in the tail of the distribution. Thus density gives an approximate selectivity estimate for an equality predicate whose value falls within a non-singleton bucket. An advantage of our definition of histogram density is that it does not reflect the high-frequency outliers in the data distribution, so a single average value will serve as an accurate estimate for many data distributions.

For efficiency and simplicity, the same infrastructure is used for all data types except longer string and binary data. An order-preserving hash function, whose range is a double-precision floating point value, is used to derive the bucket boundaries on these data types. For longer string and binary data types, SQL Anywhere uses a different infrastructure that dynamically maintains a list of observed predicates and their selectivities. In addition, not only are buckets created for entire string values, but also for `LIKE` patterns intended to match a 'word' somewhere in the string.

Statistics collection during query processing. Histograms are created automatically when data is loaded into the database using a `LOAD TABLE` statement, when an index is created, or when an explicit `CREATE STATISTICS` statement is executed. A modified version of Greenwald's algorithm [14] is used to create the cumulative distribution function for each column. Our modifications significantly reduce the overhead of statistics collection with a marginal reduction in quality.

In addition to these bulk operations, histograms are automatically updated during query processing. During normal operation, the evaluation of (almost) any predicate over a base column can lead to an update of the histogram for this column. `INSERT`, `UPDATE`, and `DELETE` statements also update the histograms for the modified columns. Each histogram's density metric is updated using a moving average with exponential smoothing. That is, given histogram density d and an observed value selectivity v, we take the new histogram density to be $d+\lambda(v-d)$ where λ is a small constant. We apply this procedure to observed selectivities due to query feedback from equality predicates, and also for selectivities of singleton buckets that are removed from the histogram.

Experience. Techniques such as query feedback enable SQL Anywhere's ability to perform well in zero-administration environments. However, a monster lurks: these self-managing techniques would occasionally fail to satisfactorily maintain high-quality statistics. Part of the issue is that statistics are modified on a per-connection basis, so concurrent transactions can result in incorrect histogram modifications. Another issue is that, to avoid concurrency bottlenecks, histogram maintenance is not transactional.

These issues have prompted the development of 'self-healing' measures to identify inaccurate statistics, determine appropriate remedies to correct the problem, and schedule those remedies for execution, all without DBA intervention. The SQL Anywhere server automatically tracks estimated predicate selectivities to actual values, and when the amount or frequency of error is considered high enough, determines how the error should be rectified. One technique is to replace outright the current histogram with a reconstructed one based on the query's experience feedback. If the error is more widespread, however, the server may schedule one of several different piggybacking mechanisms (cf. reference [13]) to reconstruct portions of the histogram through (a) piggybacking off a user query, (b) index sampling, or (c) stratified table scan sampling. In addition, the server tracks the frequency with which specific histograms are used, and concentrates its maintenance efforts on those histograms most critical to the user's workload.

These feedback mechanisms, along with on-the-fly, real-time index and table statistics, such as the number of distinct values, number of leaf pages, and clustering statistics, have greatly improved the accuracy of the statistics used for query optimization in SQL Anywhere. Recent customer experience strongly suggests that these

techniques have significantly reduced the need for manual or scheduled intervention to correct statistical errors. Nonetheless, we are continuing to enhance our histogram maintenance techniques, including the development of analysis and reporting tools to enable diagnosis of histogram and other statistical anomalies without requiring access to the raw customer data.

4 Adaptive Query Execution

SQL Anywhere uses feedback control to 'tune' buffer pool memory to meet overall system requirements, including memory demands from other applications. The feedback control mechanism uses the OS working set size, the amount of free physical memory, and the buffer miss rate as its inputs, which are ordinarily polled once per minute but are polled much more frequently at server startup in anticipation of application activity. Since the buffer pool size can grow—or shrink—on demand, query execution must be able to adapt to execution-time changes in the amount of available physical memory.

All page frames in SQL Anywhere's heterogeneous buffer pool are the same size, and can be used for table and index pages, undo or redo log pages, bitmaps, query memory, or connection heaps. Each type of page has different usage patterns: heap pages, for example, are local to a connection, while table pages are shared. Therefore, the buffer pool's page replacement algorithm, a modified generalized 'clock' algorithm [15], must be aware of differences in reference locality. No attempt is made in SQL Anywhere to support buffer pool partitions. In our view, dynamic changes to the system's workload that occur frequently in SQL Anywhere deployments render static workload configuration tools ineffective.

SQL Anywhere's query optimizer can automatically annotate a chosen query execution strategy with alternative plan operators that offer a cheaper execution technique if either the optimizer's choices are found to be suboptimal at run-time, or the operator requires a low-memory strategy at the behest of the memory governor (see below). For example, the optimizer may construct an alternative index-nested loops strategy for a hash join, in case the size of the build input is considerably smaller than that expected at optimization time. Hash join, hash-based duplicate elimination, and sorting are examples of operators that have low-memory execution alternatives. A special operator for execution of `RECURSIVE UNION` is able to switch between several alternative strategies, possibly using a different one for each recursive iteration.

To ensure that SQL Anywhere maintains a consistent memory footprint, in-memory data structures used by query processing operators are allocated within heaps that are subject to page replacement within the buffer pool. Moreover, as the buffer pool can shrink during query execution, memory-intensive query execution operators must be able to adapt to changing memory availability.

The memory governor controls query execution by limiting memory consumption for a statement to not exceed its quota. Approaching this quota may result in the memory governor requesting query execution operators to free memory if possible. For example, hash-based operations in SQL Anywhere choose a number of buckets based on the expected number of distinct hash keys; the goal is to have a small number of distinct key values per bucket. In turn, buckets are divided uniformly into a small, fixed number of partitions. The number of partitions is selected to provide a balance between I/O behaviour and fanout. During the processing of the hash operation, the governor detects when the query plan needs to stop allocating more memory—that is, it has exceeded its quota. When this happens, the partition with the largest number of rows is evicted from memory. The in-memory rows are written out to a temporary table, and incoming rows that hash to the partition are also written to disk.

By selecting the partition with the most rows, the governor frees up the most memory for future processing, in the spirit of other documented approaches in the literature [16]. By paying attention to the memory quota while building the hash table on the smaller input, the server can exploit as much memory as possible, while degrading adaptively if the input does not fit completely in memory. In addition, the server also pays attention to the quota while processing the *probe* input. If the probe input uses memory-intensive operators, their execution

may compete with the hash join for memory. If an input operator needs more memory to execute, the memory governor evicts a partition from the consuming operator's hash table. This approach prevents an input operator from being starved for memory by a consumer operator.

Experience. Adaptive query execution techniques are essential in SQL Anywhere due to the other autonomic systems within it, such as dynamic cache sizing, that can negatively impact query execution strategies. Today, adaptive execution in SQL Anywhere involves trying to salvage an execution strategy when the server context at execution time was not anticipated by the optimizer. This conservative approach 'tames' the monster of catastrophically slow execution plans or memory panic situations at execution time. So far, we have not implemented potentially expensive mitigation techniques such as on-the-fly query re-optimization. However, to better manage the tradeoff between conservative resource allocation and optimal performance, we are considering other approaches, such as more sophisticated execution plan caching techniques.

5 Self-tuning Multiprogramming Level

Several commercial database servers (e.g., Microsoft SQL Server, Oracle, and SQL Anywhere) employ a worker-per-request server architecture. In this architecture, a single queue is used for all database server requests from all connections. A worker dequeues a request from the request queue, processes the request, and then returns to service the next available request in the queue or block on an idle queue. In this configuration, there are no guarantees that a single connection will be serviced by the same worker. This architecture has proved to be more effective in handling large numbers of connections and, if configured properly, has less overhead [17]. The difficult issue with this architecture is how to set the size of the worker pool to achieve good throughput levels [18]. This parameter effectively controls the server's multiprogramming level (MPL).

Setting the server's MPL is subject to tradeoffs. A large MPL permits the server to process a large number of requests concurrently, but with the risk of thrashing due to hardware or software resource contention (locks) or to excessive context switching between threads [19]. With a small MPL, the server is able to apportion a larger amount of server memory for query execution. The obvious drawback, however, is that the server's hardware resources are under-utilized. The issue in zero-administration or deployed environments is that setting the server's MPL *a priori* is largely impossible because of changes to both the server's execution environment and the database workload over time.

SQL Anywhere overcomes this problem by dynamically adjusting the server's multiprogramming level to maximize throughput during server operation [20]. To perform the MPL adjustment, the SQL Anywhere kernel uses a feedback controller that oversees two algorithms—a hill climbing algorithm and a global parabola approximation algorithm—that significantly extends prior work by Heiss and Wagner [21]. The feedback controller periodically switches between the two algorithms. The combined approach is effective at maintaining throughput over varying workloads without large oscillations.

Experience. Hand-tuning of a server's multiprogramming level is a notoriously difficult task even with well-understood workloads, making MPL tuning a monster to be tamed. In SQL Anywhere, hand-tuning is doubly difficult because SQL Anywhere can assign multiple workers to a single request to achieve intra-query parallelism. The degree of parallelization is determined by the query optimizer in a cost-based manner, made with respect to the characteristics of the system context at optimization time. There is no static partitioning of work amongst the workers assigned to a query. Rather, a parallel query plan is organized so that any active thread can grab and process a small unit of work from anywhere in the data flow tree. The unit of work is usually a page's worth of rows. It is not important which thread executes a given unit of work; a single thread can execute the entire query if additional threads are not available. This means that parallel queries adapt gracefully to fluctuations in server load. However, autonomic MPL tuning has not slayed the monster outright; in the field, automatic adjustment does not perform well with periodic, 'bursty' workloads. Additional research is ongoing to develop better feedback control techniques to solve this problem.

6 Indexing Spatial Data

SQL Anywhere offers support for spatial data, with functionality modelled after the SQL/MM spatial standard [22]. Spatial data is indexed by *linear quadtrees* [23], which use a Z-curve to map two-dimensional quadtree tiles into 64-bit integer keys suitable for storage in a B-tree index. Query performance over linear quadtrees is heavily impacted by the granularity with which spatial objects are tessellated [24]. Choosing a tessellation granularity requires navigating a trade-off between index filtering efficacy and index size, and entails knowledge of both the data distribution and the query workload. Other commercial systems specify the tessellation granularity at index creation time, and the same tessellation algorithm is applied to both data objects (upon insertion) and to query objects over the index [24; 25]. Unfortunately, choosing tessellation granularity up front becomes problematic in embedded environments where either the data or the query distributions may not be known at design time. The approach we have taken in SQL Anywhere is to separate the choice of tessellation granularity for data and query objects. In order to ensure robust index performance in all scenarios, data objects are tessellated with a fixed coarse granularity. This conservative approach favours index size over filtering efficacy and guarantees that index scans remain as cheap as possible in case the query workload includes large containment queries; any reduction in filtering quality at the index level is mitigated by various second-stage filters.

Tessellation of spatial query objects is a great example of how several self-tuning technologies in SQL Anywhere work together. Query objects are tessellated dynamically during query execution by a cost-based algorithm that examines both the data distribution and the current server state. Within sparse data regions a query object is tessellated coarsely, thereby reducing the number of key range probes against the index; as data density increases the tessellation becomes more fine-grained, yielding more but tighter range probes. The query execution plan includes alternatives for both the spatial index or a table scan, allowing the tessellation algorithm to make a cost-based decision to revert to a table scan for large query objects whose index scanning cost is prohibitive. Given a spatial join where objects from one table form the queries against the spatial index of a second table, changes in buffer pool contents during execution of the join is taken into account by the cost function when tessellating successive query objects. Furthermore, because the knowledge of the data distribution is obtained from the server's self-tuning histograms, query tessellation adapts dynamically to both changes in the underlying data and improvements in histogram accuracy resulting from query feedback.

7 Conclusions and Future Work

Taming the various self-management monsters is exceedingly difficult. System and regression testing of autonomic systems like SQL Anywhere remain unsolved problems. Yet as challenging as these problems are, diagnosing performance problems experienced by customers can be even more difficult. Increasingly, privacy and legislative constraints prevent access to customer databases. In these situations, the existence of diagnostic tools that the customer can execute is essential. As one example, graphical representations of SQL Anywhere execution plans can contain information related to the search space considered by the query optimizer. This information ranges from the various cost model computations to the cost-based and heuristic decisions made by the optimizer during join enumeration. These traces of optimization search spaces can be analyzed in isolation, and also can be used to generate synthetic data to match the original customer database.

Database self-management does have the potential to exacerbate its own monster: unpredictable query execution times. However, our twenty years of customer experience has shown that SQL Anywhere's capacity to adapt to changing execution contexts largely trumps those infrequent situations when a specific query's execution time has become intolerable.

In the main, the term 'database self-management' typically applies to performance characteristics. However, an often-neglected aspect of self-management is with error handling, increasingly relevant due to the use of commodity hardware that will fail in predictable ways [26].

Finally, we note that with the proliferation of database systems in many different environments, backwards compatibility has become increasingly important. While such constraints are occasionally inconvenient, backwards compatibility is critical when an ISV has millions of deployed seats, and can't possibly upgrade all software instantaneously.

References

[1] Ivan T. Bowman and G. N. Paulley, "Join enumeration in a memory-constrained environment", in *Proceedings, Sixteenth* IEEE *International Conference on Data Engineering*, San Diego, California, Mar. 2000, pp. 645–654.

[2] Anisoara Nica, "A call for order in search space generation process of query optimization", in *Proceedings,* ICDE *Workshops (Self-Managing Database Systems)*, Hanover, Germany, Apr. 2011, IEEE Computer Society Press.

[3] Anisoara Nica, "Immediate materialized views with outerjoins", in *Proceedings,* ACM *Thirteenth International Workshop on Data Warehousing and* OLAP (DOLAP)*, in* ACM *Ninetheenth Conference on Information and Knowledge Management* (CIKM), Toronto, Canada, Oct. 2010, pp. 45–52.

[4] Craig Russell, "Bridging the object-relational divide", ACM *Queue*, vol. 6, no. 3, pp. 16–27, May 2008.

[5] Ivan T. Bowman, Peter Bumbulis, Dan Farrar, Anil K. Goel, Brendan Lucier, Anisoara Nica, G. N. Paulley, John Smirnios, and Matthew Young-Lai, "SQL Anywhere: A holistic approach to database self-management", in *Proceedings,* ICDE *Workshops (Self-Managing Database Systems)*, Istanbul, Turkey, Apr. 2007, pp. 414–423, IEEE Computer Society Press.

[6] G. N. Paulley and Per-Åke Larson, "Exploiting uniqueness in query optimization", in *Proceedings, Tenth* IEEE *International Conference on Data Engineering*, Houston, Texas, Feb. 1994, pp. 68–79, IEEE Computer Society Press.

[7] Kristofer Vorwerk and G. N. Paulley, "On implicate discovery and query optimization", in *Proceedings,* IEEE *International Data Engineering and Applications Symposium*, Edmonton, Alberta, July 2002, pp. 2–11, IEEE Computer Society Press.

[8] Anisoara Nica, Daniel Scott Brotherston, and David William Hillis, "Extreme visualisation of the query optimizer search spaces", in ACM SIGMOD *International Conference on Management of Data*, Providence, Rhode Island, June 2009, pp. 1067–1070.

[9] Ashraf Aboulnaga and Surajit Chaudhuri, "Self-tuning histograms: Building histograms without looking at data", in ACM SIGMOD *International Conference on Management of Data*, Philadelphia, Pennsylvania, May 1999, pp. 181–192.

[10] Nicolas Bruno and Surajit Chaudhuri, "Efficient creation of statistics over query expressions", in *Proceedings, Nineteenth* IEEE *International Conference on Data Engineering*, Bangalore, India, Mar. 2003, pp. 201–212.

[11] Surajit Chaudhuri and Vivek Narasayya, "Automating statistics management for query optimizers", in *Proceedings, Sixteenth* IEEE *International Conference on Data Engineering*, San Diego, California, Mar. 2000, pp. 339–348, IEEE Computer Society Press.

[12] Ihab F. Ilyas, Volker Markl, Peter J. Haas, Paul Brown, and Ashraf Aboulnaga, "CORDS: Automatic discovery of correlations and soft functional dependencies", in ACM SIGMOD *International Conference on Management of Data*, Paris, France, June 2004, pp. 647–658.

[13] Qiang Zhu, Brian Dunkel, Wing Lau, Suyun Chen, and Berni Schiefer, "Piggyback statistics collection for query optimization: Towards a self-maintaining database management system", *Computer Journal*, vol. 47, no. 2, pp. 221–244, 2004.

[14] Michael Greenwald, "Practical algorithms for self-scaling histograms or better than average data collection", *Performance Evaluation*, vol. 20, no. 2, pp. 19–40, June 1996.

[15] Alan Jay Smith, "Sequentiality and prefetching in database systems", ACM *Transactions on Database Systems*, vol. 3, no. 3, pp. 223–247, Sept. 1978.

[16] Hans-Jörg Zeller and Jim Gray, "An adaptive hash join algorithm for multiuser environments", in *Proceedings of the 16th International Conference on Very Large Data Bases*, Brisbane, Australia, Aug. 1990, pp. 186–197.

[17] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich Nahum, and Adam Wierman, "How to determine a good multi-programming level for external scheduling", in *Proceedings, 22nd* IEEE *International Conference on Data Engineering*, Washington, DC, Apr. 2006, p. 60, IEEE Computer Society.

[18] Stavros Harizopoulos, *Staged Database Systems*, PhD thesis, Carnegie Mellon University, 2005.

[19] Shimin Chen, Phillip B. Gibbons, Tood C. Mowry, and Gary Valentin, "Fractal prefetching B^+-trees: Optimizing both cache and disk performance", in ACM SIGMOD *International Conference on Management of Data*, Madison, Wisconsin, June 2002, pp. 157–168, Association for Computing Machinery.

[20] Mohammed Abouzour, Kenneth Salem, and Peter Bumbulis, "Automatic tuning of the multiprogramming level in Sybase SQL Anywhere", in *Proceedings, 26th* IEEE *International Conference on Data Engineering: Workshops*, Long Beach, California, Mar. 2010, pp. 99–104, IEEE Computer Society Press.

[21] Hans-Ulrich Heiss and Roger Wagner, "Adaptive load control in transaction processing systems", in *Proceedings, 17th International Conference on Very Large Data Bases*, Barcelona, Spain, Sept. 1991, pp. 47–54, Morgan-Kaufmann.

[22] International Standards Organization, ISO/IEC *13249-3:2011, Information technology—Database languages—SQL Multimedia and application packages: Part 3, Spatial*, 2011.

[23] Irene Gargantini, "An effective way to represent quadtrees", *Communications of the* ACM, vol. 25, no. 12, pp. 905–910, Dec. 1982.

[24] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov, "Quadtree and R-tree indexes in Oracle spatial: a comparison using GIS data", in ACM SIGMOD *International Conference on Management of Data*, Madison, Wisconsin, 2002, pp. 546–557, Association for Computing Machinery.

[25] Yi Fang, Marc Friedman, Giri Nair, Michael Rys, and Ana-Elisa Schmid, "Spatial indexing in Microsoft SQL Server 2008", in ACM SIGMOD *International Conference on Management of Data*, Vancouver, Canada, 2008, pp. 1207–1216, Association for Computing Machinery.

[26] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, "An analysis of data corruption in the storage stack", in *FAST*, Mary Baker and Erik Riedel, Eds. 2008, pp. 223–238, USENIX.

AutoAdmin Project at Microsoft Research: Lessons Learned

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

Microsoft Corporation

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

1 Introduction

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

2 Physical Database Design

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

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2.1 Core Learnings

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

2.1.1 "What-if" API in the Server

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

2.1.2 Reducing the Number of Optimizer Calls

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

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

2.1.3 Importance of Merging

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

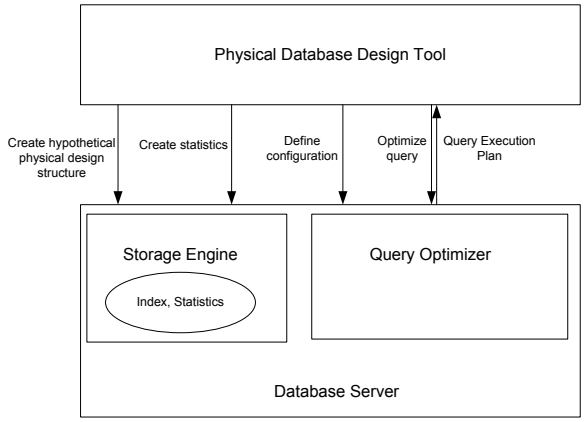


Figure 1: What-if analysis architecture for physical database design

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

2.1.4 Integrated Selection of Different Physical Design Structures

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

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

2.1.5 Techniques for Handling Large Workloads

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

2.2 Open Challenges

2.2.1 Top-down vs. Bottom-up search

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

2.2.2 Handling Richer Constraints

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

2.2.3 Exploiting Test Servers for Tuning

The process of tuning a large workload can impose significant overhead on the server being tuned since the physical design tool needs to potentially make many "what-if" API calls to the query optimizer component. In enterprises there are often test servers in addition to the production server(s). A test server can be used for a

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

2.2.4 Different Modes of Tuning

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

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

3 Exploiting Execution Feedback for Query Optimization

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

3.1 Self-Tuning Histograms

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

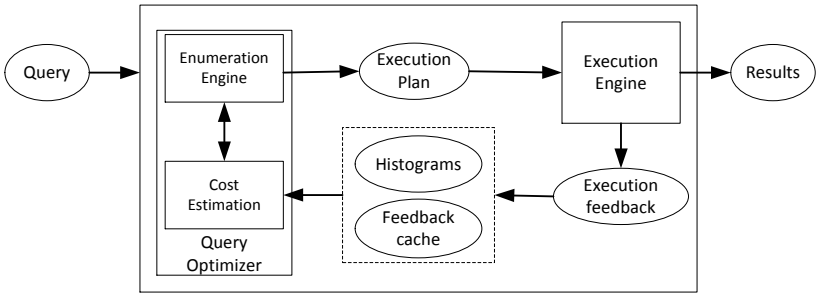


Figure 2: Architecture for incorporating execution feedback into query optimization using (a) histograms or (b) via a feedback cache.

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

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

3.2 Feedback-Cache Approach

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

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

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

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

4 Query Progress Estimation

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

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A Decade of Oracle Database Manageability

Pete Belknap John Beresniewicz Benoit Dageville Karl Dias Uri Shaft Khaled Yagoub

Oracle Corporation

Abstract

Oracle took on the challenge of ever-increasing management complexity for the database server beginning some eleven years ago with Oracle 10g and continuing through the 11g release. The Oracle RDBMS offers numerous services and features that share infrastructure and resources to enable a rich portfolio of industry-leading data management capabilities. Oracle Database Server Manageability refers to a cross-organization effort across various RDBMS core teams to deliver a coherent feature suite that enables customers to manage an Oracle database with consistency, confidence and relative ease through the Enterprise Manager UI. Our efforts to improve database manageability can be classified into three levels of increasing sophistication:

- *Statistics and reporting level– where key workload performance statistics and run-time data are continuously and efficiently collected by default, persisted into a self-managing repository, and exposed through reports and interactive UI*
- *Advisory level – where instrumentation level performance data are periodically analyzed using a time-based methodology that makes findings and recommendations about sub-optimal performance, its root cause, and opportunities for improvement*
- *Automated implementation level – where advisory level recommendations that can be implemented with low and well-understood risk are automatically put in place (given prior consent of administrators)*

Each layer builds upon the ones below it, and each plays a part in satisfying the key customer use cases – it is only through exposing features at all three levels that the solution becomes complete. In this paper we will discuss our key offerings at each level, review motivating use cases and lessons learned, and conclude by looking forward to some problems that excite us in the near future.

1 Introduction

Manageability has been a major focus area in every Oracle RDBMS release since version 10 (2003)[2], reaching beyond self-tuning to consider other approaches as well. Improving the ease-of-use of the database is an effort that encompasses all development teams. We have focused on the following high-level management problems:

- Ease of **out-of-box configuration**: we have shrunk the set of mandatory parameters by establishing sensible default values, removed others when the system can properly self-tune them, and in general improved core RDBMS components to tune themselves.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

- Reducing the **conceptual burden** on the DBA: for example, we classified the hundreds of Oracle wait events into just twelve wait classes to make it easier for a DBA to understand the nature of bottlenecks inside the server.

- Simplifying the basic **monitoring** of an Oracle database: we have introduced innovative mechanisms to track time spent inside the server at finer levels of granularity, in real-time, and with visibility into the past. This has allowed us to create more scientific, measurement-oriented performance tuning workflows.

- Assisting with the **analysis of performance data**: Our intelligent advisors productize the performance tuning methodologies that Oracle experts use to find the root cause of performance problems and recommend corrective actions. In some cases these advisors can be configured to take action automatically themselves.

We could not achieve such wide goals by building each solution uniquely or from scratch–we needed a unified architecture. The one we engineered ensures conceptual consistency across the board and provides common data interfaces. It consists of the following three layers added to the RDBMS server:

1. Statistics and reporting level – where key workload performance statistics and run-time data are continuously and efficiently collected by default, persisted into a self-managing repository, and exposed through reports and an interactive graphical user interface.

2. Advisory level – where instrumentation level performance data are periodically analyzed using a time-based methodology that makes findings and recommendations about sub-optimal performance, its root cause, and opportunities for improvement.

3. Automated implementation level – where advisory level recommendations that can be implemented with low and well-understood risk are automatically put in place (given prior consent of administrators).

The Oracle manageability stack follows a bottom-up learning paradigm, where facts and knowledge are derived from basic workload and system statistics, advisors analyze those facts and recommend actions to the user, and automatic logic tests and implements a small subset of those findings.

Each layer serves a critical purpose in its own right. While some of us thought at first that automated features and intelligent advisors would eliminate the need for raw statistics, we found each piece to be necessary and the set to be complementary. Sometimes the reasons were situational (e.g., DBAs of mission-critical systems often want to exercise more fine-grained control than those running hundreds of smaller databases) and at other times more practical (e.g., some problems caused by the application, such as a poorly written SQL or user-level locking issues, cannot be automatically solved within the RDBMS). Also, many problems proved significantly easier to solve when we can engage directly with the user in an advisory capacity. The parallel SQL execution finding in the SQL Tuning Advisor [3] illustrates this nicely: while it was relatively simple for us to generate a set of facts and estimates for the user about the benefits of enabling parallel execution for a particular SQL statement (e.g., estimated speedup, increase in resource consumption, etc), implementing the action automatically was not feasible. Without complete workload insight we could not quantify the risk of doing so, and we needed the user's involvement to evaluate the trade-off between resource consumption and response time. Had we limited our scope to what we could automatically implement, we would have built a far less useful product.

Oracle has also developed a parallel technology stack to make the database self-healing and self-diagnosing with respect to software failures and defects. This infrastructure, which we call the diagnosability framework [7], is beyond the scope of this paper. From the beginning we recognized its distinction from manageability was unclear: for example, a performance problem could be caused by a software bug. To avoid confusion we have focused the manageability initiative on performance monitoring and tuning and diagnosability on product defects. We will stay with the former for the remainder of this paper, which will be structured as follows: Sections

2, 3, and 4 go into depth on the statistics, advisory, and automation layers, respectively; Section 5 explains how Enterprise Manager connects them together into packaged workflows; and in Section 6 we conclude and look forward to the future by mentioning some of the bigger challenges that our customers face today.

2 Statistics and Reporting Layer

The foundation of the manageability stack is the statistics and reporting layer. It gathers key workload information in real-time to expose the most important aspects of database performance to internal components and end users as they require. For many releases Oracle has provided basic monitoring capabilities through a layer of views that expose in-memory performance data. The Oracle 10g manageability framework brought three key innovations for the tracking and capturing of performance data:

- Statistics are computed **continuously**, updated in **real-time**, and track the **trend** of values over time.
- **Historical data** is persisted in a built-in and self-managing workload repository (AWR) [5].
- The core mechanisms are **enabled by default**.

Each one of these represents significant advances, but especially the third one. Collecting comprehensive performance data is a necessary precondition to building automated higher-level features, and having the instrumentation mechanisms on by default makes them always available. More importantly, it forces a requirement onto those mechanisms to be robust across platforms and versions as well as to be extremely lightweight. Automated statistics infrastructure must never itself tax system throughput.

2.1 Statistics Framework

2.1.1 Active Session History

The **Active Session History (ASH)** facility exemplifies these characteristics both in the volume and detail of the data it collects by default, and in how this enables sophisticated root cause analysis of a wide array of performance issues. ASH is a sampled history of session activity collected by a single kernel process into a circular buffer and flushed to disk regularly. ASH samples contain both an instantaneous snapshot of activity (current user, SQL, etc) as well as a set of resource consumption statistics, such as actual I/O or CPU consumption. ASH uses a negligible amount of CPU and memory, is implemented without latching or locking, and has proven to be highly robust. ASH enables the breakdown of database activity along some eighty dimensions, the identification of top SQL statements, database users, wait times and reasons, and much more. This helps one understand where time is spent inside the database, even after performing arbitrary filtering to limit the scope of analysis.

2.1.2 Other Components

- **Real-Time SQL Monitoring** [8] captures detailed information about individual long-running SQL executions, including key items such as plan operation-level memory consumption, actual run-time cardinality values, and time distribution across parallel server processes.
- The **Automatic Workload Repository (AWR)** captures periodic snapshots of performance data, allowing users to get a complete understanding of what the system was doing at any period in the past. It keeps data for one week by default. Users can choose to keep all data for as long as they wish, or they can assign a retention policy to a specific window of AWR data, enabling comparisons to system baselines.
- The **Time Model** measures and computes our core performance metric – Database Time [5] – at the session and system levels, as well as broken down at finer granularities (e.g., SQL execution, SQL parsing). Time spent in the database is the basis for all Oracle performance analysis.

- **Global Wait Chains** expose a cluster-wide waits-for graph that allows one to find top blockers on the system and understand where they are currently active.

2.2 Active Reports

While we love the flexibility and expressiveness of exposing statistics in database views, we have found the SQL interface too cumbersome for everyday customer use cases. Each of our statistic infrastructure components exposes a reporting API, allowing users to generate reports with the most useful data from each source. We have made many reports highly graphical and interactive through technology we call Active Reports [12]. Active Reports incorporate the same rich, engaging UI technology as Oracle Enterprise Manager while still being available on the Database command-line. They are fully contained in one file and, like standard reports, can be viewed without any mid-tier or database connectivity.

These features are the key players in our statistics layer. While they serve as the basis of features in the advisory and automated implementation layers, they are invaluable in their own right when users need to understand the rationale behind the output of the upper layers or when they prefer to drive an investigation themselves.

3 Advisory Layer

Oracle provides a suite of intelligent advisors within the RDBMS and Enterprise Manager that transform the data provided by the Statistics and Reporting Level using a tested tuning methodology into concrete recommendations for the user to review.

The following advisors specialize in throughput analysis and are the most mature of the group:

- **ADDM** [5] uses ASH and AWR data to perform a holistic analysis of system throughput, computing Database Time – the cumulative amount of time spent within the server working directly on behalf of the user – and analyzing it along multiple dimensions to identify the key performance bottlenecks.

- **SQL Tuning Advisor (STA)** [3] focuses on a single SQL statement, examining it for many common root causes of poor performance, such as missing or stale statistics and cardinality mis-estimation .

- **SQL Access Advisor (SAA)** attempts to find new access structures (indexes, materialized views, partitioning) to benefit the workload as a whole.

- **Segment Advisor** works from a different goal – optimizing the storage layout of an application schema. There is still a strong relationship to performance because of caching benefits and concurrency impact in shrinking or compressing objects.

3.1 ADDM

The following diagram shows how ADDM integrates with the other advisors to provide a single starting point for throughput tuning. ADDM builds a useful model from the underlying statistics (ASH and AWR) for consumption by a higher-level rule set which outputs useful findings. ADDM builds a comprehensive, hierarchical model of database time that is capable of answering a variety of important questions. For example, it can determine how time is spread across CPU, IO, and Network waits, and then drill down into each of these categories (e.g., to determine whether high Network waits were due to high distributed lock usage or large amounts of parallel query communication). ADDM's rules engine can then pose these questions and find a possible remedy. Generic, well-defined concepts like the ADDM time model and rule set are what enable an intelligent advisor like ADDM to scale to so many different types of problems.

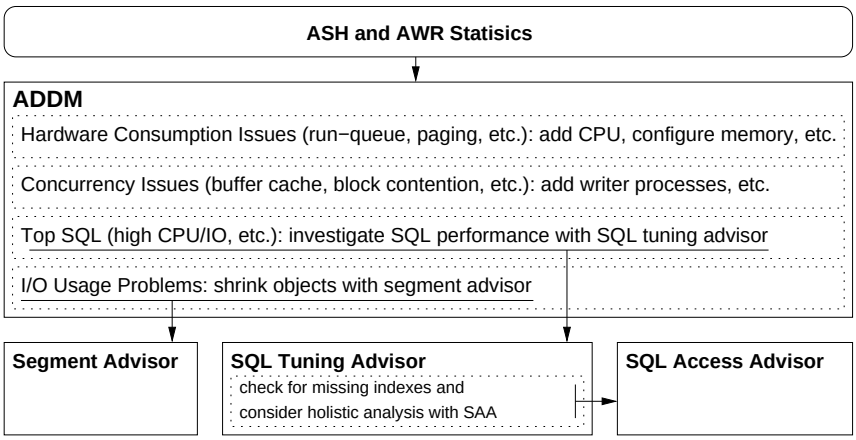


Figure 1: ADDM integration with other database components

3.2 Other Advisors

- Comparative Analysis, namely, comparing workload statistics across two periods of time:
  - **Compare Period ADDM** takes a system-wide perspective, searching for relevant configuration changes and associating them with specific performance impacts. It might identify that a regression in performance, with a symptom of increased I/O time, was caused by a manual reconfiguration of the memory available to the instance.
  - **SQL Performance Analyzer (SPA)** works at the SQL statement level, identifying key improved and regressed statements, and providing a remedy to fix the regressions [10]. When a regression is caused by a plan change, users can create SQL plan baselines to instruct the optimizer to choose the better execution plan [11].
- Real-Time Analysis: **Real-Time ADDM** views the most recent five minutes of system activity and identifies the biggest performance problems happening on the system at the moment. It can, for example, recommend killing a process to clear a hang or reconfiguring shared memory to lower contention.

We have learned through experience the importance of providing actionable advice and come to understand the true nature of root-cause analysis as well. We recognize that advice is most useful when it indicates a clear action to the DBA, and that this only happens when it is accompanied by the right bits of additional information. We have found this to be an easy goal to miss when we provide information which we think is useful but ends up being not truly actionable, or we make a good finding without the right data to back it up and convince a user that it will solve his problem. For example, we had to fine-tune ADDM's memory configuration findings over time as we saw they were not truly useful until we quantified the benefit that they bring and explained the payoff or penalty that comes with each incremental increase or decrease in available memory.

It was always our goal to provide root-cause analysis, but it took time to develop a definition that could be put to practical use – over time we have come to view it as a spectrum, realizing that it may be impossible to reach the ultimate root cause of a problem but that we should seek to get as close as possible. Each step closer is valuable to users when it helps them understand the cause of a problem to a finer level of detail. To take an example from the SQL tuning advisor: while we may never identify the precise mis-estimate that was the cause of a bad execution plan (and why that is), understanding that the problem is related to cardinality estimation in general is useful to a DBA because it helps him narrow down the issue to a particular part of query optimization.

These advisors offer broad, intelligent advice and are generally useful in everyday system performance scenarios. In addition to running an advisor to solve a problem and accepting its advice, a DBA might do a

manual run for deeper analysis than the automatic run could perform or use it as a convenient starting point when embarking on a longer manual investigation. In this way, the manual advisors work well alongside the statistics and automated implementation layers while still adding value in their own right.

4 Automated Implementation Layer

Atop our statistics and advisory layers, the automated implementation layer attempts to find actions that are good candidates for automatic action. This is obviously attractive because of its potential to simplify a complex system. We have learned, however, to be conservative in this space – this is also where we take the most risk of harming the system. In this section we describe the Oracle self-tuning solution first at the component level and then at the system level.

4.1 Self-tuning Components

4.1.1 Automatic Memory Management

Automatic memory management [4] has likely seen more adoption than any other component-level self-tuning feature. It allows DBAs to set a single memory target for the entire instance, or two memory targets, one each for shared and private (SQL) memory. Before it was introduced, DBAs had to tune the allocation of each memory pool, such as the buffer pool and sort buffer size. With one or both memory targets in place, the feature divides the memory set into smaller granules which are exchanged between components as they are needed. This eases the configuration burden and at the same time reduces the likelihood of out of memory errors or unnecessary spills to disk. To help users understand the overall memory need of a system, Oracle includes a simulator that determines the optimal size of each memory component. ADDM also exposes this advice as a set of cache tuning findings. In this way ADDM helps the user set the overall sizing, after which the system can effectively allocate memory internally.

4.1.2 Other Components

- The **Query Optimizer** examines execution-time statistics and feeds significant cardinality mistakes back into the engine to generate a better plan for future executions [9].
- The **Parallel Execution engine** automatically determines the degree of parallelism (DOP) for a SQL statement based on the statement's cost [6]. Previously users had to set a DOP manually at the table level.
- The **Database Resource Manager** caps CPU consumption of automatic system maintenance to ensure that it does not negatively impact application performance.
- **Real Application Clusters** performs load balancing across instances of a clustered database and also prevents a single instance from negatively impacting the cluster by terminating instances and misbehaving processes if they lose I/O responsiveness or cause a hang.

4.2 Advisory Automation

The SQL Tuning Advisor runs in an automated mode during nightly maintenance windows to tune SQL statements without the DBA's involvement [1]. While the statistics layer made it easy to find the SQL statements that consumed the most time over the past week, and the advisor layer already had the tuning algorithms and methodologies we needed for this feature, there were many new challenges unique to automated implementation. Chief amongst these were determining how to allocate our tuning resources, deciding what recommendations are good candidates for auto-implementation, and providing additional safeguards as necessary to ensure that the tuning itself will not harm the database system. These are more general than SQL tuning: each one would arise in automating any intelligent advisor and would make sense in another context with similar characteristics.

Obviously the most important decision to get right was which recommendations to implement, and why. To follow a conservative approach, we decided that for a recommendation to make sense for automated implementation, we must be able to test it effectively. This limited us to the SQL plan tuning recommendations, since testing was fairly straightforward – execute the SQL with both versions of the plan and choose the better one. Even so, we found plenty of complexity: we had to decide what exactly we mean by "better", what to do about parallel queries, how to execute DML without impacting the system, and consider many other issues.

4.3 Challenges and Experience

Automated implementation has turned out to be very challenging when it comes to providing the essential guarantees, and we have realized that it makes more sense in some areas than others. We have found it useful to evaluate the promise of each potential action according to three variables: scope – the ability to target only the performance problem, testability – how effectively we can test an action before taking it, and flexibility – how quickly the system can change a decision after making it. Automatic memory management and SQL plan tuning perform well under this test, as the former is highly flexible and the second testable and limited in scope.

Comparing to the feature set of the lower two layers, this level is relatively sparse. The automated implementation functionality provides an invaluable service to DBAs in terms of fixing small nagging problems before they become big enough to demand someone's attention, but we know we will never reach a time when performance problems cease to exist. A robust statistics and advisory layer will always be critically important.

5 Building a Complete Solution

Were our manageability offering limited to one RDBMS feature, then we would have missed a critical need for a holistic management product to tie all of the various components together into one coherent story. Oracle Enterprise Manager (EM) fills the gap and is for us the primary database management interface. Through EM we accomplish two important goals: first, by using end-to-end workflows in EM, DBAs need not memorize a logical sequence of feature transitions in order to complete a single goal. Instead they can interact with the system in a more natural way, one task at a time. And second, by building interactive user interfaces, we have created ways for DBAs to engage directly with the system to access the richness of our on-by-default collections simply and in real time.

Our database throughput tuning workflow is a prime example. The entry point for performance monitoring in EM is the database performance page. From here, users can view key throughput metrics on their system like CPU load, database activity, and I/O metrics, even comparing to a baseline that represents normal performance. Should the activity look significant, the user can drill down to ADDM findings, where she might see a list of top SQL statements running on the system that are candidates for tuning. She can then run the SQL Tuning Advisor directly and implement a recommendation of her choosing, such as gathering statistics or choosing a new plan. On another attempt, she might see that the SQL Tuning advisor automatically identified and fixed the problem itself. Workflows like this one join all three levels of our stack, and transparently: users need not understand where one feature ends and the next begins, but can rather drive a logical progression through a simple methodology. This greatly lessens the learning curve and creates a seamless user experience.

6 Remaining Challenges

The last eleven years have been a great adventure in database manageability for us. We have heard our customers request a wide spectrum of features. In various contexts, they have looked for active management through manual statistics analysis, passive management through intelligent advisors, and independent self-management

through full automation. Our solution contains key offerings in each category, and it is only after covering them all and connecting them with a compelling graphical user interface that it feels complete.

Looking forward to our future can be as simple as finding the things our customers are doing today which require a number of labor-intensive manual steps. On the other hand, it can be as hard as trying to guess at what they will be doing next. Fortunately, by doing the former we find many fascinating problems to occupy us:

- The trend towards **Cloud** or centralized computing clusters increases the need for cross-database management functionality.
- **Consolidating Databases**, whether by merging multiple databases into one, or putting several databases on one host, must be driven from an analysis of the right performance metrics.
- Many basic **monitoring, advisory**, and **automation** challenges are yet to be addressed. For example, tuning for access structures is still a manual process.

We look forward to seeing where our customers will take us next.

Acknowledgements

The authors would like to thank our colleagues at Oracle for their valuable contributions over the past eleven years. In addition, we would like to specially thank Manivasakan Sabesan for helping us with LATEX.

References

[1] P. Belknap, B. Dageville, K. Dias, and K. Yagoub, Self-Tuning for SQL Performance in Oracle Database 11g, in *SMDB ICDE '09*, 2009.

[2] B. Dageville and K. Dias, Oracle's Self-Tuning Architecture and Solutions, in *IEEE Data Engineering Bulletin*, vol. 29, 2006.

[3] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin, Automatic SQL Tuning in Oracle10g, in *VLDB '04*, 2004.

[4] B. Dageville and M. Zait, SQL Memory Management in Oracle9i, in *VLDB '02*, 2002.

[5] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood, Automatic Performance Diagnosis and Tuning in Oracle, in *CIDR '05*,2005.

[6] J. Dijcks, H. Baer, and M. Colgan, Oracle Database Parallel Execution Fundamentals, *http://www.oracle.com/technetwork/database/focus-areas/bi-datawarehousing/twp-parallel-executionfundamentals-133639.pdf*

[7] M. Fallen and Y. Sarig, Automatic Fault Diagnostics, *http://www.oracle.com/technetwork/database/focus-areas/manageability/diagnosability-white-paper-ow07-131084.pdf*

[8] S. Koltakov, B. Dageville, and P. Belknap, Real-Time SQL Monitoring, *http://www.oracle.com/technetwork/database/focus-areas/manageability/owp-sql-monitoring-128746.pdf*

[9] A. Lee, M. Zait, Closing the Query Processing Loop in Oracle 11g, in *VLDB '08*, 2008.

[10] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu, Oracles SQL Performance Analyzer, in *IEEE Data Engineering Bulletin*, vol. 31,2008.

[11] M. Ziauddin, D. Das, H. Su, Y. Zhu, K. Yagoub, Optimizer Plan Change Management: Improved Stability and Performance in Oracle 11gg, in *VLDB '08*, 2008.

[12] Real-Time SQL Monitoring Active ReportsAn optimal on-line algorithm for metrical task system, *http://www.oracle.com/technetwork/database/focus-areas/manageability/sqlmonitor-084401.html*.

Benchmarking Online Index-Tuning Algorithms

Ivo Jimenez* Jeff LeFevre* Neoklis Polyzotis* Huascar Sanchez* Karl Schnaitter†

*University of California Santa Cruz †Aster Data

Abstract

The topic of index tuning has received considerable attention in the research literature. However, very few studies provide a comparative evaluation of the proposed index tuning techniques in the same environment and with the same experimental methodology. In this paper, we outline our efforts in this direction with the development of a performance benchmark for the specific problem of online index tuning. We describe the salient features of the benchmark, present some representative results on the evaluation of different index tuning techniques, and conclude with lessons we learned about implementing and running a benchmark for self tuning systems.

1 Introduction

Several recent studies [1; 2; 3] have advocated an online approach to the problem of index tuning. In a nutshell, an online tuning algorithm continuously monitors and analyzes the current workload, and automatically changes the set of materialized indexes in order to maximize the efficiency of query processing. Hence, the main assumption in online tuning is that the workload is volatile and requires the system to be retuned periodically. This is in direct contrast to offline tuning, where the system is tuned based on a fixed, representative workload.

Despite the recent flurry of research on this topic, little work has been done to comparatively evaluate different tuning algorithms using the same experimental methodology. In this paper, we describe a benchmark that we developed specifically for this purpose. The benchmark is grounded on the empirical studies of previous works [1; 2; 3], but also includes significant extensions that can stress-test the performance of an online tuning algorithm.

The remainder of the paper describes the salient features of the benchmark along with experimental results from a specific implementation over PostgreSQL. The complete details can be found in the respective publications [4; 5].

2 Online Index Tuning: Problem Statement

Before describing the details of our benchmark, we formally define the problem of online index tuning. The formal statement reflects several of the metrics and scenarios that appear in the benchmark specification.

We define a configuration as a set of indexes that can be used in the physical schema. We use $\mathcal{S}$ to denote the space of possible configurations, and note that in practice $\mathcal{S}$ contains all sets of indexes that can be defined over the existing tables, whose required storage is below some fixed limit. We use $cost(q, C)$ for the cost of

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

evaluating query q under configuration C. There is also a cost $\delta(C, C')$ involved with changing between two configurations $C, C' \in \mathcal{S}$.

We formulate the problem of online index tuning in terms of online optimization. In this setting, the problem input provides a sequence of queries $q_1, q_2, \ldots, q_n$ to be evaluated in order. The job of online index selection is to choose for each q_i a configuration $C_i \in \mathcal{S}$ to process the query, while minimizing the combined cost of queries and configuration changes. In other words, the objective of an online tuning algorithm is to choose configurations that minimize

$$TotWork = \sum_{i=1}^{n} \delta(C_{i-1}, C_i) + cost(q_i, C_i)$$

where C_0 denotes the initial configuration. Furthermore, the online algorithm must select each configuration C_i using knowledge only from the preceding queries $q_1, \ldots, q_{i-1}$, i.e., without knowing the queries that will be executed in the future. This is precisely what gives the problem its online nature.

$TotWork$ has been used in studies of metrical task systems [6; 7] and also in more closely related work in online tuning [1]. It is a natural choice to describe the problem, as it accounts for the main costs of query processing and index creation that are affected by online index selection. It is also straightforward to compute using standard cost models, making it a useful yardstick for measuring the performance of a self-tuning system. On the other hand, it does not model all practical issues involved with online tuning. For instance, observe that our statement of the problem assumes that the configuration C_i is always installed before query q_i is evaluated. This may not be feasible in practice because there may not be enough time between q_{i-1} and q_i to modify the physical schema, unless the evaluation of q_i is delayed by the system. It is more realistic to change the index configuration concurrently while queries are answered, or wait for a period of reduced system load.

The $TotWork$ metric also does not reflect the overhead work required to analyze the observed queries $q_1, \ldots, q_{i-1}$ in order to choose each C_i. One significant source of this overhead can be the use of *what-if* optimization [8; 9] to estimate $cost(q, C)$ for a candidate configuration C. The what-if optimizer simulates the presence of C during query optimization to obtain the hypothetical cost, which is used in turn to gauge the benefit of materializing C in the physical schema. Although this is an accurate measure of query cost, each use of the what-if optimizer can take significant time, akin to normal query optimization. It is thus important for a tuning algorithm to limit the use of what-if optimization, or evaluate $cost(q, C)$ more efficiently.

3 Benchmark Specification

We first discuss the building blocks of the benchmark in terms of the data, the workload, the performance metrics and the operating environment. Subsequently, we define the specific testing scenarios that form the core of the benchmark.

3.1 Building Blocks

Data. At a high level, the proposed benchmark models a hosting scenario where the same system hosts several separate databases, and the workload may shift its focus to different subsets of the databases over time. Specifically, the current incarnation of the benchmark contains four databases: TPC-H, TPC-C, and TPC-E from the TPC benchmarking suite [1], and the real-life NREF data set [2]. (Note that TPC-H and NREF have been used in a previous benchmarking study on physical design tuning.)

The selection of these specific data sets is not crucial for the benchmark. Indeed, it is possible to apply the overall methodology to a different choice of databases. Any selection, however, should contain sufficiently different data sets, so that we can create a diverse workload by shifting from one database to another. On the

[1] `http://www.tpc.org`

[2] `http://pir.georgetown.edu/pirwww/dbinfo/nref.shtml`

other hand, the data sets should be roughly equal in size, so that no database becomes too cheap (conversely, too expensive) to process. To model an interesting setting, we require that the total size of the databases exceeds the main memory capacity of the hardware on which the benchmark executes.

Workloads. We generate synthetic workloads over the databases based on shifting distributions. Specifically, each workload may contain several phases, where each phase corresponds to a distribution that favors queries or updates on a different subset of the databases. For instance, the workload may start with a focus on queries over the TPC-H data and updates over the TPC-C data, then switch to a phase that contains queries and updates over the TPC-C data, then to a phase that contains queries over the TPC-C data and updates over the TPC-E data, and so on. To model a realistic setting, the transition between phases happens in a gradual fashion. Clearly, our goal is to generate a workload that requires a different set of indexes per phase, with phases that have overlapping distributions and gradual shifts. This is exactly the scenario targeted by online tuning algorithms.

The characteristics of the statements that appear in a workload are controlled by the following parameters:

Query-only vs. Mixed A query-only workload contains solely SELECT statements, whereas a mixed workload also contains UPDATE statements. The latter are modeled to affect a numerical attribute in the database, using small randomized increments and decrements that average out to zero on expectation. This scheme ensures that UPDATE statements do not change the value distribution of the updated attribute. In this fashion, update statements cause maintenance overhead for the materialized indexes without affecting the accuracy of the optimizer's data statistics.

Statement Complexity We define three complexity classes for SELECT statements: single-table queries with one (highly selective) selection predicate, single-table queries with a conjunction of selection predicates (of mixed selectivity factors), and general multi-table queries with several join and selection predicates. The first class represents workloads that are easy to tune, since each query has a single "ideal" index for its evaluation. The third class represents significantly more complex workloads, where indexes are useful for a subset of the join and selection predicates of each query, and index interactions play a more prominent role in index tuning [10]. Similarly, we define two classes (easy and hard) for UPDATE statements, by changing the structure of the WHERE clause.

Phase Length This parameter controls the number of queries in each phase, and hence controls the volatility of the whole workload.

Distribution of DB Popularity We employ two types of distributions for the query and update statements within each phase. The first distribution, termed "80-20", places 80% of the workload on two databases (referred to as the *focus* of the phase), whereas the remaining 20% is uniformly distributed among the remaining databases. Consecutive phases have distinct foci that overlap, which introduces a realistic correlation between phases. The second distribution, termed "Uniform", simply divides the workload equally among all the databases.

Maximum Size of Predicated Attribute-Set This parameter controls the size and diversity of candidate indexes for the workload, by constraining the maximum number of attributes, per table, that may appear in selection predicates. More concretely, assuming that the value of this parameter is x, queries can place selection predicates on the x attributes of each table with the highest active-domain size. The rationale is that these attributes are the most "interesting" in terms of predicate generation. A small parameter value implies a small number of candidate indexes, and vice versa.

An alternative approach to random queries would be to use standardized query sets that come with the chosen databases. As an example, for the TPC-H database, we could draw at random from the predefined TPC-H queries. However, we still need to define a query generator for databases that do not come with a standardized query set, e.g., NREF. It is thus meaningful to use the same generator for all databases to ensure uniformity.

Moreover, it is difficult to create interesting scenarios for online tuning using standardized workloads, as the latter are not designed for this purpose. For instance, it is challenging to create a diverse workload using just the 22 queries of the TPC-H workload.

Metrics. We employ two metrics to measure the performance of an online tuning algorithm for a specific workload. The first is the $TotWork$ metric defined previously, which is based on the query optimizer's cost model and captures the total work performed by the system when it is tuned by the specific algorithm. This metric includes the estimated cost to process each workload statement using the current set of materialized indexes, plus the estimated cost to materialize indexes. (The assumption is that all index materializations take place in between successive statements.) The total-work metric is commonly used in the experimental evaluation of tuning algorithms, and it is a good performance indicator that is not affected by system details, such as the errors in the optimizer's cost model or the overhead of creating indexes in parallel to normal query processing.

The second metric simply measures the total wall-clock time to process the workload. This metric reflects clearly the quality of the tuning, but is also affected by factors such as the accuracy of the optimizer's cost model (which in turn drives the selections of the online tuning algorithm) and the overhead of creating indexes online.

For both metrics, we report the improvement brought by online tuning compared to a baseline system configuration that does not contain any indexes (except perhaps for any primary or foreign key indexes that are created automatically). We choose this stripped-down baseline in order to gauge the maximum benefit of a particular online tuning algorithm. More concretely, assuming that X_{ot} and X_b measure the cost of the online tuning system and the baseline system respectively using one of the aforementioned metrics, we measure the performance of online tuning with the ratio $\rho = (X_b - X_{ot})/X_b$. The sign of ρ indicates whether online tuning improves ($\rho > 0$) or degrades ($\rho < 0$) system performance compared to the baseline. The absolute value $|\rho|$ denotes the percentage of improvement or degradation.

Operating Environment. We assume that the data is stored and manipulated in a relational database management system. We chose to not constrain the features of the DBMS in any way, except to require that the system implements the services required by the online index-tuning algorithms that are being tested. This requirement is not ideal, since it causes the benchmark specification to depend on the algorithms being tested, but it avoids the greater danger of choosing system features that are biased toward one index selection algorithm.

Another important aspect of the environment is the maximum storage allowed for indexes created by the online tuning algorithm, which we refer to as the *storage budget*. A larger storage budget intuitively makes the problem easier since it allows a large number of indexes to be materialized without needing to choose indexes to evict. On the other hand, the budget should be large enough for the tuning algorithm to materialize an index configuration with significant benefit. Based on these observations, we compute the physical storage required by the tables in each database, and set the storage budget to the mean of the database sizes. This results in a moderate storage budget that is interesting for index selection.

3.2 Workload Suites

The core of the benchmark consists of four workload suites that exercise specific features of an online tuning algorithm. Each workload suite uses a variety of parameter settings for our workload generation methodology. For ease of reference, Table 1 summarizes the parameter settings for each suite.

C1: Reflex The first suite examines the performance of online tuning for shifting workloads of varying phase length. This parameter essentially determines the speed at which the query distribution changes, and thus aims to test the reaction time of an online tuning algorithm.

A workload contains queries in the moderate class, which involve multi-column covering indexes that may interact. There are $m = 4$ phases, and each phase uses an 80-20 distribution. The focus of the workload shifts to two different databases with each phase, ensuring that consecutive phases overlap on exactly one database. This implies that consecutive phases have overlapping sets of useful indexes, which creates an interesting case for online tuning.

Suite	Number of Phases	Phase Length	Distribution of DB popularity	Query Template	Update Template	Maximum size of predicated attribute-set
C1	4	50 100 200 400	80-20	Moderate	-	∞
C2	4	300	80-20	Simple Moderate Complex	-	∞
C3	4	300	80-20	Moderate	Simple Complex	∞
C4	1	500	Uniform	Moderate	-	∞
C5	4	300	80-20	Moderate	-	1 2 3 4

Table 1: Workload generation parameters for the test suites.

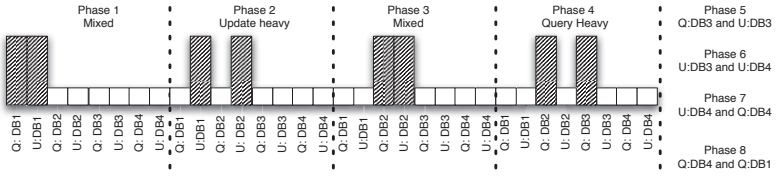


Figure 1: Distribution for the phases in an update workload. Shaded bars denote the focus of the phase. The notation "Q: DB1" denotes queries on database DB1, whereas "U: DB1" denotes updates.

The suite comprises four workloads with phase length 50, 100, 200, and 400. An aggressive tuning algorithm is likely to observe good performance across the board, even when transitions are short-lived. A conservative algorithm is likely to view short-lived transitions as noise, and show gains only for larger phase sizes.

C2: Queries The second test scenario examines the performance of online tuning as we vary the query complexity of the workload. This parameter affects the level of difficulty for performing automatic tuning.

A workload consists of m phases, with the same distribution as in the previous suite. Thus, we have a gradually shifting workload with overlapping phases. The phase length is set to 300 queries, under the assumption that this is long enough for a reasonable algorithm to adapt to each phase.

We generate three workloads consisting of simple, moderate, and complex queries respectively. The simple workload is expected to yield the highest gain for any tuning algorithm. The moderate workload exercises the ability of the algorithm to handle multi-column indexes and index interactions. Finally, the complex workload introduces join indexes to the mix.

C3: Updates The third suite examines the performance of a tuning algorithm for workloads of queries and updates. A good tuning algorithm must take into account both the benefit and the maintenance overhead of indexes.

A workload in this suite consists of $2m$ phases, following the ordering shown in Figure 1. Each phase applies again an 80-20 rule for the distribution. The ordering of phases creates the following types of phases: mixed, where the same database is both queried and updated; query-heavy, where most of the statements are queries over two different databases; and, update-heavy, where most of the statements are updates over two different databases. The idea is to alternate between phases with a different effect on online tuning. For instance, a query-heavy phase is likely to benefit from the materialization of indexes, whereas an update-heavy phase is likely to incur a high maintenance cost if any indexes are materialized.

We generate two workloads with simple and complex updates respectively. In both cases, the queries are of moderate complexity. Clearly, simple updates involve less complicated index interactions and hence the bookkeeping is expected to be simpler. Conversely, we expect online tuning to be more difficult for updates of moderate complexity.

C4: Convergence This suite examines online tuning with a stable workload that does not contain any shifts, i.e., the whole workload consists of a single phase. The expectation is that any online tuning algorithm will converge to a configuration that will remain unchanged by the end of the workload. Thus, it is interesting to identify the point in the workload where the configuration "freezes".

The suite consists of a single workload with a single phase of 500 moderate queries and a uniform distribution. This creates a setting where the choice of optimal indexes is less obvious, compared to a phase that focuses on a specific set of databases.

C5: Variability The last suite examines the performance of online tuning as we expand the set of attributes that carry selection predicates.

The workloads in this suite are identical to suite C2, except that we vary the maximum size of the predicated attribute-sets as 1,2,3, and 4. We expect to observe a direct correlation between the variability of the workload and the effectiveness of an online tuning algorithm, but it is also interesting to observe the relative difference in performance for the different values of the varied parameter.

4 Experimental Study

We performed an experimental study of several online tuning algorithms using the described benchmark. We first describe the setup of the experimental study and the discuss briefly the key results.

4.1 Experimental Setup

Index-Tuning Algorithms. We tested three index tuning algorithms: COLT[3], BC[1] and WFIT[11]. We chose these algorithms as they have very different features: COLT employs stochastic optimization to make its tuning decisions; BC employs retrospection over the past workload to choose index-sets, and requires specific machinery from the underlying system; WFIT also employs retrospection, but it relies on standard system interfaces and also carries theoretical performance guarantees on its choices. The three algorithms also have different overheads, with COLT being the most light-weight and WFIT being the most expensive.

We implemented these algorithms with PostgreSQL as the underlying DBMS. We chose PostgreSQL since it is a mature, robust system and the source code is freely available. The implementation required us to extend PostgreSQL with several query- and index-analysis primitives, including a what-if optimizer to estimate the benefit of hypothetical indexes. We also added support for true covering indexes, which can bring significant benefits to query evaluation.

DBMS Setup. PostgreSQL reported a total size of 4GB after loading all data and building indexes for primary keys. Of this, 2.9GB was attributed to the base table storage. As specified by the benchmark, we set the index storage budget to the mean of the four database sizes, which in our case was about 750MB. To create an interesting execution environment, we artificially limited the available RAM to 1GB and set the DBMS buffer pool to 64MB.[3] We note that we kept a small data scale to ensure timely completion of the experiments.

4.2 Summary of Results

Due to space constraints, we provide only a summary of our findings based on the experimental results. The complete details can be found in the relevant full publications[5; 4].

[3]We followed the practice advocated by PostgreSQL administrators to delegate cache management to the file system.

C1: Reflex. All algorithms had improved performance as the phase length increased. Overall, the two retrospective algorithms BC and WFIT performed better than COLT, which relies on stochastic optimization. One possible explanation is that COLT had limited data (especially for short phases) on which to make reliable predictions, whereas BC and WFIT employ a retrospective analysis of the complete past workload and avoid predictions altogether.

C2: Queries. All algorithms followed the same trend of decreasing performance as the query complexity increased. The retrospective algorithms again performed better than COLT, although the difference was not drastic for the simplest class of queries. WFIT exhibited a consistent advantage over BC, but the difference was small in all experiments.

C3: Updates. The inclusion of updates in the workload resulted in an interesting reversal of the ranking of the three algorithms. WFIT continued to be the top performer, but now COLT performed better than BC, which took a significant performance hit. One possible explanation is that BC relies on heuristics in order to update benefit statistics with the effects of update statements and index interactions, and these heuristics do not work well for the given workload and the specific DBMS that we employ (the original algorithm was developed for MS SQL Server).

C4: Convergence. All algorithms improve the workload, with WFIT offering about twice as much benefit as the other algorithms. As noted in the previous section, the purpose of this experiment is to verify that the algorithms converge quickly, since we expect the online algorithm to be stable when the workload is stable. WFIT excels in this regard: in the second half of the workload, the index materializations of WFIT had a total cost that was two orders of magnitude lower compared to the corresponding costs of COLT and BC respectively.

C5: Variability. As expected, all three algorithms performed best when the maximum size was set to one, which in turn resulted in very few candidate indexes that had a clear benefit. WFIT was the top performer in all cases, followed by BC and COLT, in that order. BC exhibited an interesting drop in its performance when the set size increased from two to three, which we can attribute to the increase in the intensity of index interactions and the heuristics that BC uses to deal with these interactions.

Overhead. There is a noticeable difference between the three algorithms in terms of their overhead, which we separate in two disjoint components: the overhead to analyze each statement in the incoming workload and update internal statistics, and the number of what-if optimizations performed by the algorithm. The reason behind this separation is that the speed of what-if optimization is tightly coupled with the specific DBMS, whereas the first component is specific to the algorithm and hence likely to be unaffected by changes to the DBMS.

COLT has the least analysis overhead of all algorithms, as it is designed to offer a light-weight solution for online index tuning. In fact, COLT averages less than one what-if call per query, which means that some queries are analyzed without even contacting the what-if optimizer. BC has a higher analysis overhead due to its more complex internal logic, but it does not require any what-if optimization calls. Instead, all analysis is done by examining the optimal plan for executing the query under analysis. WFIT has the highest analysis overhead, with an order of magnitude difference from the other two algorithms. This is due to two factors: WFIT employs a sophisticated analysis of the workload, which enables it to choose indexes with strong performance guarantees; and, the algorithm is implemented in unoptimized Java code that runs on top of the DBMS, whereas both COLT and BC are implemented in optimized C code inside PostgreSQL. Accordingly, it also performs far more what-if calls than the other two algorithms. Hence, WFIT is a viable choice only if the what-if optimizer is fast, or if it is possible to employ a fast what-if optimization technique such as INUM or CPQO.

5 Lessons Learned

The implementation of our benchmark allowed us to gain useful insights on the relative performance, strengths and weaknesses of three online index-tuning algorithms. We strongly believe that the proposed benchmark provides a principled methodology to stress-test online tuning algorithms under several scenarios.

At the same time, our experience from implementing the benchmark (and using it subsequently for several experimental studies) has revealed several ways to improve the overall methodology. More concretely, it is important to couple the complex workload suites with simpler micro-benchmark workloads which contain a handful of queries and can thus offer greater visibility in the behavior of an online tuning algorithm. Furthermore, we have observed that the generated workloads have a varying degree of difficulty across different systems. It would be desirable to have a benchmark with a more homogeneous behavior across different DBMSs.

In a different direction, we attempted to implement the benchmark in a database-as-a-service environment, in order to gauge the plausibility of online index tuning "in the cloud". Unfortunately, we were not able to get reliable results due to the extremely high variance of our measurements. Our observations corroborate the results of a more general study by Schad et al.[12] We theorize that the source of this variance was the execution of the DBMS inside a virtual machine at the service provider, and the performance characteristics of distributed block storage vs. local storage. Our take-away message, for now, is that it may be difficult to execute an online tuning algorithm in this environment, given that the what-if optimizer will not be able to accurately predict the actual benefit of different indexes.

References

[1] N. Bruno and S. Chaudhuri, "An online approach to physical design tuning," in *ICDE '07: Proceedings of the 23rd International Conference on Data Engineering*, 2007.

[2] K.-U. Sattler, M. Lühring, I. Geist, and E. Schallehn, "Autonomous management of soft indexes," in *SMDB '07: Proceedings of the 2nd International Workshop on Self-Managing Database Systems*, 2007.

[3] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis, "On-line index selection for shifting workloads," in *SMDB '07: Proceedings of the 2nd International Workshop on Self-Managing Database Systems*, 2007.

[4] K. Schnaitter and N. Polyzotis, "A benchmark for online index selection," in *SMDB '09: Proceedings of the 4th International Workshop on Self-Managing Database Systems*. IEEE, 2009, pp. 1701–1708.

[5] K. Schnaitter, "On-line index selection for physical database tuning," Ph.D. dissertation, University of California Santa Cruz, 2010.

[6] A. Borodin, N. Linial, and M. E. Saks, "An optimal on-line algorithm for metrical task system," *Journal of the ACM*, vol. 39, no. 4, pp. 745–763, 1992.

[7] W. R. Burley and S. Irani, "On algorithm design for metrical task systems," *Algorithmica*, vol. 18, no. 4, pp. 461–485, 1997.

[8] S. Finkelstein, M. Schkolnick, and P. Tiberio, "Physical database design for relational databases," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 91–128, 1988.

[9] S. Chaudhuri and V. Narasayya, "Autoadmin "what-if" index analysis utility," *SIGMOD Rec.*, vol. 27, no. 2, pp. 367–378, 1998.

[10] K. Schnaitter, N. Polyzotis, and L. Getoor, "Index interactions in physical design tuning: modeling, analysis, and applications," *Proc. VLDB Endow.*, vol. 2, 2009.

[11] K. Schnaitter and N. Polyzotis, "Semi-Automatic Index Tuning: Keeping DBAs in the Loop," *CoRR*, vol. abs/1004.1249, 2010.

[12] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *PVLDB*, vol. 3, no. 1, pp. 460–471, 2010.

Adaptive Processing of User-Defined Aggregates in Jaql

Andrey Balmin
abalmin@us.ibm.com

Vuk Ercegovac
vercego@us.ibm.com

Rares Vernica*
rares.vernica@hp.com

Kevin Beyer
dr.beyer@gmail.com

Abstract

Adaptive techniques can dramatically improve performance and simplify tuning for MapReduce jobs. However, their implementation often requires global coordination between map tasks, which breaks a key assumption of MapReduce that mappers run in isolation. We show that it is possible to preserve fault-tolerance, scalability, and ease of use of MapReduce by allowing map tasks to utilize a limited set of high-level coordination primitives. We have implemented these primitives on top of an open source distributed coordination service. We expose adaptive features in a high-level declarative query language, Jaql, by utilizing unique features of the language, such as higher-order functions and physical transparency. For instance, we observe that maintaining a small amount of global state could help improve performance for a class of aggregate functions that are able to limit the output based on a global threshold. Such algorithms arise, for example, in Top-K processing, skyline queries, and exception handling. We provide a simple API that facilitates safe and efficient development of such functions.

1 Introduction

The MapReduce parallel data-processing framework, pioneered by Google, is quickly gaining popularity in industry [1; 2; 3; 4] as well as in academia [5; 6; 7]. Hadoop [1] is the dominant open-source MapReduce implementation backed by Yahoo!, Facebook, and others. IBM's BigInsights [8] platform is one of the enterprise "Big Data" products based on Hadoop that have recently appeared on the market.

A MapReduce job consist of two phases: map and reduce. In the map phase, input data is partitioned into a number of *splits* and a *map task* processes a single split. A map task scans the split's data and calls a *map function* for every record of its input to produce intermediate key,value pairs. The map task uses the intermediate keys to partition its outputs amongst a fixed number of *reduce tasks* and for sorting. Each reduce task starts with a *shuffle* where it copies over and merges all of its input partitions. After shuffle, the reducer calls a *reduce function* on each set of map output records that share the same intermediate key.

In order to provide a simple programming environment for users, MapReduce offers a limited choice of execution strategies, which can adversely affect performance and usability of the platform. Much of this rigidity is due to one key assumption of MapReduce that map tasks run in isolation and do not depend on each other. This *independent mappers* assumption allows MapReduce to flexibly partition the inputs, arbitrarily order their processing, and safely reprocess them in case of failures. This assumption also implies that only embarrassingly

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Work done at IBM Almaden Research Center.

parallel algorithms can run during the map phase. All other computation must be shifted to the reduce phase, which requires, potentially very expensive, redistribution of the data during the shuffle. Consequently, MapReduce users are often forced to choose less efficient embarrassingly parallel versions of the algorithms, even if much more efficient options are available that require modest communication between tasks.

In this paper we describe our Adaptive MapReduce approach that breaks the independent mappers assumption by introducing an asynchronous communication channel between mappers using a transactional, distributed meta-data store (DMDS). This store enables the mappers to post some metadata about their state and see state of all other mappers. Such "situation-aware mappers" (SAMs) can get an aggregate view of the job execution state to make globally coordinated optimization decisions. However, while implementing SAM-based features, one has to be careful to satisfy key MapReduce assumptions about scalability and fault tolerance, and not introduce noticeable performance overhead.

To externalize SAMs, we propose an API for developing user defined aggregates (UDAs) for the Jaql [9] query language. This API is powerful enough to allow efficient implementation of an interesting and diverse set of functions, ranging from Top-K to Skyline queries, sampling, event triggering, and many others. At the same time, it provides simple primitives to UDA developers, and transparently takes care of DMDS communication, ensuring fault-tolerance and scalability of the approach. Jaql's unique modularization and data heterogeneity features make the resulting UDAs applicable for a wide variety of scenarios.

In our prior work [10] we employed SAM-based adaptive techniques to enhance Hadoop with variable checkpoint intervals, best effort hash-based local aggregation, and flexible, sample based, partitioning of map outputs. Note that Hadoop's flexible programming environment enabled us to implement all of these adaptive techniques without any changes to Hadoop. We chose an adaptive runtime to avoid performance tuning, which is currently challenging [11]. Adaptive algorithms demonstrate superior performance stability, as they are robust to tuning errors and changing runtime conditions, such as other jobs running on the same cluster. Furthermore, adaptive techniques do not rely on cardinality and cost estimation, which in turn requires accurate analytical modeling of job execution. Such modeling is extremely difficult for large scale MapReduce environments, mainly for two reasons. First, is the scale and interference from other software running on the same cluster. Parallel databases, for instance, rarely scale to thousands of nodes and always assume full control of the cluster. Second, MapReduce is a programming environment where much of the processing is done by black-box user code. Even in higher-level query processing systems, such as Jaql, Pig [3], and Hive [4], queries typically include user-defined functions that are written in Java. That is why all these systems offer various query "hint" mechanisms instead of traditional cost-based optimizers. In this environment, the use of adaptive run-time algorithms is an attractive strategy.

One of the adaptive techniques in [10], called Adaptive Combiners, helps speed up generic UDAs by keeping their state in a fixed-size hash-based buffer in local memory. However, to preserve generality, we did not allow communication between UDA implementations. This prevented efficient implementation for a wide class of UDAs. A simple example of such a UDA is a Top-K query. Consider a query over a webserver click-log dataset that asks for the last million records. Unless the dataset is partitioned by time, the simplest way to execute this request in MapReduce is to produce the last million keys from every map task, and then merge these map output lists in a single reducer until one million results are produced. Running this job with a thousand map tasks will result in up to a billion records output by map tasks and copied by the reducer. Needless to say, this may be very expensive. Imagine if the map tasks were able to maintain the current one-millionth most recent time-stamp. It could be used to filter the map outputs, to minimize the number of false positives sent to the reducer. Smaller map outputs means quicker local sorts and less data for the reducer to shuffle and merge.

Of course, maintaining this one-millionth key *threshold* exactly will require a lot of communication. Recent studies [12; 13] show that it is possible to efficiently minimize the amount of communication between worker nodes for the Top-K queries and other similar algorithms. The high level approach is to use relatively few communications to maintain a small global state, e.g., in case of Top-K, a histogram of the keys processed so far. This state is used to compute an approximate threshold. In this case, most false positives could be filtered

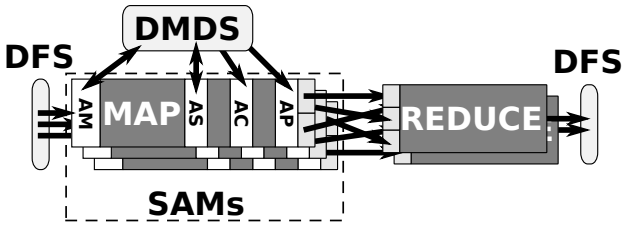


Figure 1: Adaptive techniques in SAMs and their communication with DMDS.

out from map outputs. Thus, Top-K algorithm could be much more efficient if map tasks were able to coordinate their efforts.

The same approach works for all other *thresholding* aggregation algorithms [12], which are able to filter out the data that requires aggregation, based on some global threshold. Examples of such algorithms include skyline queries, data sampling, and event triggering, among others. Thresholding algorithms do not fit a MapReduce paradigm as their efficient implementation require communication between map tasks. Allowing arbitrary communication amongst map tasks introduces potential dependencies which may lead to deadlocks and would prevent re-execution of tasks in case of failures.

In [10] we focused on system-level adaptive runtime options that could not be customized by the user. Thus, we could address fault-tolerance and scalability issues for each technique individually in an ad-hoc fashion. For each adaptive technique we implemented a unique recovery mechanism that allowed SAMs to be re-executed at any time. We also avoided synchronization barriers and paid special attention to recovery from task failures in the critical path.

Our experience with these techniques led us to believe that while SAMs are a very useful mechanism for system programming, the DMDS API is too low level for developers. Thus we have designed more straightforward APIs that could abstract out the details needed for fault-tolerance and performance. In this paper we describe one such API that enables developers to efficiently implement arbitrary thresholding functions in a very straightforward manner.

2 Background

2.1 Adaptive MapReduce

In our prior work we utilized SAMs in a number of adaptive techniques, some of which are already available in IBM's BigInsights product. Adaptive Mappers (AMs) dynamically "stitch" together multiple splits to be processed by a single mapper, thus changing the checkpoint interval, Adaptive Combiners (ACs) use best-effort hash-based aggregation of map outputs, Adaptive Sampling (AS) uses some early map outputs to produce a global sample of their keys, and Adaptive Partitioning (AP) dynamically partitions map outputs based on the sample. Figure 1 shows the adaptive techniques in the SAMs and their communication with DMDS.

The API proposed in this paper builds upon **Adaptive Combiners (ACs)** to perform local aggregation in a fixed-size hash table kept in a mapper, as does Hive and as was suggested in [14]. However, unlike these systems, once the hash table is filled up, it is not flushed. An AC keeps the hash table and starts using it as a cache. Before outputting a result, the mapper probes the table and calls the local aggregation function (i.e., *combiner*) in case of a cache hit. The behavior in case of a cache miss is determined by a pluggable replacement policy. We study two replacement policies: No Replacement (NR) and Least Recently Used (LRU). Our experimental evaluation shows that NR is very efficient - its performance overhead is barely noticeable even when map outputs are nearly unique. LRU overhead is substantially higher, however it can outperform NR, if map output keys are very skewed or clustered, due to a better cache hit ratio.

While ACs use an adaptive algorithm, their decisions are local and thus do not utilize SAMs. However, we employ SAM-based Adaptive Sampling technique, to predict if AC will benefit query execution and decide which replacement policy and cache size to use. Also, AMs further improve performance benefit of ACs as they increase the amount of data that gets combined in the same hash table.

All adaptive techniques mentioned above are packaged as a library that can be used by Hadoop developers through a simple API. Notice that the original programming API of MapReduce remains completely unchanged. In order to make the adaptive techniques completely transparent to the user, we also implemented them inside the Jaql [9] query processor.

2.2 ZooKeeper

One of the main components of our SAM-based techniques is a distributed meta-data store (DMDS). The store has to perform efficient distributed read and writes of small amounts of data in a transactional manner. We use Apache ZooKeeper [15; 16], an open-source distributed coordination service. The service is highly available, if configured with three or more servers, and fault tolerant. Data is organized in a hierarchical structure similar to a file system, except that each node can contain both data and sub-nodes. A node's content is a sequence of bytes and has a version number attached to it. A ZooKeeper server keeps the entire structure and the associated data cached in memory. Reads are extremely fast, but writes are slightly slower because the data needs to be serialized to disk and agreed upon by the majority of the servers. Transactions are supported by versioning the data. The service provides a basic set of primitives, like `create`, `delete`, `exists`, `get` and `set`, which can be easily used to build more complex services such as synchronization and leader election. Clients can connect to any of the servers and, in case the server fails, they can reconnect to any other server while sequential consistency is preserved. Moreover, clients can set watches on certain ZooKeeper nodes and they get a notification if there are any changes to those nodes.

2.3 Jaql

IBM's BigInsights [8] platform includes the Jaql system [9] to flexibly and scalably process large datasets. The Jaql system consists of a scripting language that features: 1) core operators that are based on relational algebra, 2) functions and higher-order functions for extensibility, modularity, and encapsulation, and 3) a data model that is based on JSON [17] to flexibly model the heterogeneity and nesting that often arises in big data. The Jaql compiler parses and transforms scripts into a DAG of MapReduce jobs for scalable evaluation.

Jaql's scripting language enables developers to blend high-level, declarative queries with low-level expressions that directly control the evaluation plan. This feature is referred to as *physical transparency* since it allows developers to judiciously design part of their script to directly access physical operators – either to tune performance or to exploit new operators. We exploit Jaql's *physical transparency* to expose Adaptive MapReduce functionality so that developers can efficiently implement a large class of aggregate functions. Since Jaql is a functional language, we also exploit (higher-order)functions to encapsulate the low-level details of Adaptive MapReduce so that such *adaptive* aggregates are accessible to a broader user base.

3 Adaptive Thresholding Functions

All adaptive thresholding functions in Jaql implement the following Java interface, which completely hide DMDS interactions, allowing the developer to concentrate purely on the semantics of the thresholding functions.

- `boolean filter(inVal)`: Compares the `inVal` against the current `threshold` structure, possibly using a comparator passed as a parameter to the Jaql thresholding function, and filters out the inputs that don't pass the threshold.

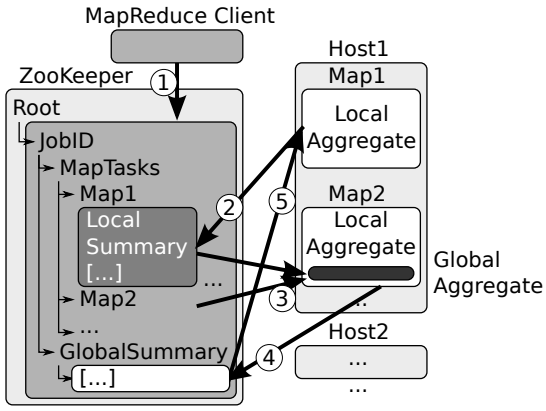


Figure 2: Communication between map tasks and ZooKeeper for a job computing an aggregate.

- `void aggregate(inVal)`: Combines inputs that pass the `filter()`, with an internal `accumulator` structure. This function is already implemented by Jaql's *algebraic* aggregate functions.
- `val summarize()`: Produces a `local summary` based on the current state of the aggregate function runtime, including the `accumulator`. If it returns a new summary, it is automatically written to ZooKeeper and eventually used to update the `threshold`.
- `val combineSummaries([val])`: Produces a global summary given a set of local ones. This function is called automatically every time some local summary changes,
- `val computeThreshold(val)`: Computes `threshold` structure given a global summary. This function is called automatically every time the global summary changes.

Jaql translates aggregate functions into a low-level expression called `mrAggregate()` that acts as a MapReduce client, directly submitting a MapReduce job which is parameterized by the appropriate map, combine (e.g., partial aggregate), and reduce functions. `mrAggregate()` invokes the functions listed above, hiding from the developer the low-level interactions with ZooKeeper that are shown in Figure 2. We describe this process in detail below.

Before the job is launched, a ZooKeeper structure is created for the job by the MapReduce client. The structure starts with a `JobID` node which contains two sub-nodes: a *MapTasks* node where mappers post local information, and a *GlobalSummary* node where the leader-mapper posts global information (Step 1 in Figure 2.)

Once the map task is running, `mrAggregate()` initializes the aggregate function runtime, and calls `filter()` for every map output record. If the result is *true*, it calls `aggregate()` and then `summarize()` for this record. If `summarize()` returns a non-empty local summary, the operator connects to ZooKeeper (or uses an already existing connection) and updates the *LocalSummary* corresponding to map task ID of the current task (Step 2 in Figure 2.)

One of the mappers is elected a *leader* using standard ZooKeeper techniques. We also use ZooKeeper to efficiently maintain a list of *potential leaders*, to facilitate quick recovery in case of a leader failure.

The leader uses the *watch* mechanism of ZooKeeper to keep track of updates to *LocalSummary* structures (Step 3 in Figure 2.) Once some local summary has been updated, subject to throttling rules that prevent the leader from becoming a bottleneck, the leader calls `combineSummaries()` with a set of all local summaries. If the resulting global summary is different from the one currently in ZooKeeper, the leader also calls `computeThreshold()` on the new global summary. The leader writes both global summary and the threshold data structure to ZooKeeper, in their respective nodes under *GlobalSummary* (Step 4 in Figure 2.) The *Global-*

Summary is stored to expedite leader recovery. Otherwise, the new leader would have to recompute it during the recovery.

Every map task, during the initialization stage, sets a watch on the ZooKeeper node that stores the threshold. Thus, as soon as the leader updates the threshold in ZooKeeper, every map task gets notified and asynchronously copies the new threshold into their local aggregate data structure (Step 5 in Figure 2.)

In the remainder of this section, we illustrate how an adaptive version of the *Top-K* aggregate is implemented in Jaql.

Example 1: Consider a collection of web pages that has been parsed, hyper-links extracted, and the top 1000000 pages by their number of out links need to be selected for further processing. This can be easily specified using Jaql's `topN` aggregate function:

```
read( parsedPages ) -> topN( 1000000, cmp(x) [ x.numOutLinks desc ] );
```

The `read` function takes an I/O descriptor, `parsedPages`, as input and returns an array of objects, also referred to as *records*. The `parsedPages` descriptor specifies where the data is located (e.g., a path to a file) and how the data is formatted (e.g., binary JSON). In the example above, each record returned by `read` represents a parsed web page. One of the fields of each parsed page is `numOutLinks` and we want to obtain the top 1000000 pages according to the highest number of out links. The `topN` aggregate function takes an array as input, a threshold `k`, and a comparator, and returns an array whose length is $\leq k$ according to the ordering specified by the comparator. The input array is implicitly passed to `topN` using Jaql's `->` symbol, which is similar to the Unix pipe operator (`|`). The comparator is specified using a lambda function, `cmp`, whose input x is bound to each parsed page. Given two parsed pages, p_1 and p_2, $p_1 > p_2$ iff p_1.`numOutLinks`$> p_2$.`numOutLinks`.

Jaql transforms the above query into a single MapReduce job where the map phase reads the input, computes a partial Top-K aggregate, and the reduce phase aggregates the partial Top-K results into a final result. In the case of `topN`, a heap of size k is maintained that is then sent to the reducers for further aggregation. Since the example computes a global aggregate, all Top-K partial aggregates are sent to a single reduce task.

Note that each map task computes and sends its entire Top-K list to the reduce phase, which is conservative. With Adaptive MapReduce, a little global state allows us to be more aggressive so that substantially less data is sent to the reducers. Our Top-K aggregate function exploits Adaptive MapReduce by leveraging Jaql's physical transparency to control the MapReduce job and is packaged up as a Jaql function, *amrTopN*, so that its usage is identical to the `topN` aggregate:

```
read( parsedPages ) -> amrTopN( 1000000, cmp(x) [ x.outLinks desc ] );
```

The `amrTopN` can be implemented by slightly modifying `topN` as follows. The function runtime keeps a single key, such as `outLinks` in the above example, as the threshold. `filter(x)` simply calls `cmp(x,threshold)`, to purge all values (above)below the threshold. `aggregate(x)` remains the same as in `topN` - it still maintains its Top-K heap accumulator. `summarize()` returns the current (upper)lower-bound, in case a new one has been fund by the last `aggregate()` call. If a new local (upper)lower-bound is produced, ZooKeeper will notify the leader, who will invoke `combineSummaries()` to compute min(max) of all the local (upper)lower-bounds. That min(max) is both the global summary and a threshold in this case, thus `computeThreshold()` is the identity function. Once this global (upper)lower-bound is updated, all map tasks are notified of the change, so they update their local threshold and start using the new one for their `filter()` calls.

When the mappers complete, they sort all outputs and run the combiner function that does final local aggregation. In this case, the combiners will do some final purging using the latest global threshold. After that, map outputs, i.e., local Top-K aggregates will be sent to the reduce phase. Usually, mappers will discard much of their input due to threshold filtering and will send over less data to the reducer.

We intentionally used a very basic implementation of `amrTopN` above for demonstration purposes. A more efficient implementation uses a small (e.g., $log_2(k)$ points) histogram of the top k keys as local state. `combineSummaries()` creates a single histogram by combining all the local ones, and `computeThreshold()` produces the max threshold that it can guarantee to be outside top k keys produced so far *globally*. For example, for $k = 1000$, we'll keep 10-point local histograms, each consisting of v_{100}, v_{200}, ..., v_{1000}, where v_n is the current n-th top value. Having 10 local histograms, each with $v_{100} \geq N$, guarantees that the global threshold is at least N.

This approach is a generalization of the basic algorithm, which essentially always uses a histogram of size one. It is obviously going to produce thresholds that are at least as tight as those of the basic algorithm, and in practice, provide much better filtering for a very minor increase in communication.

Group-by queries. Up to now we considered top-level aggregate queries, which computed a single instance of a thresholding aggregate function. However, aggregate functions are often used in `group by` queries, where records are partitioned by some criteria and an aggregate is computed per partition. For example, suppose our collection of web pages also includes the top level domain of each page's URL, and that the top 1000000 pages by their number of out links need to be found, per domain. This can be done by the following Jaql query:

```
read( parsedPages ) -> group by g = $.domain into
  { domain: g, top: amrTopN( 1000000, cmp(x) [ x.outLinks desc ] )};
```

To implement this case we utilize our prior work in Adaptive Combiners (AC), in particular their Jaql implementation that enhances the `mrAggregate()` implementation with a cache of the most popular group keys along with their accumulator structures. To support thresholding functions, we use a similar fixed-size cache structure in ZooKeeper by creating a child node per group key under the *JobID* node. Each of these group nodes, contains child *MapTasks* and the *GlobalSummary*. However, we only allow up to a certain number of group nodes. If a group node is evicted (deleted) from ZooKeeper, its corresponding group may remain in local cache(s), where its local copy of the threshold will be maintained, but not synchronized with other mappers. This caching approach provides best-effort performance improvement. Mappers will filter out as many false positives as possible, subject to the cache size constraints.

Fault Tolerance. We implemented a number of techniques to facilitate correct and efficient recovery from leader failure. As soon as the leader's ZooKeeper connection is interrupted, the next task on the list of "potential leaders" is notified and it becomes a leader. Since both local and global summaries are stored in ZooKeeper and are versioned, the new leader can quickly pick up where the last one left off.

Non-leader task recovery is handled transparently by MapReduce framework and does not impact other tasks, as long as thresholds are monotonic, as they are in all of our examples. We currently do not support non-monotonic functions, however recent work on the subject [12] takes a promising approach of implementing general thresholding functions using monotonic ones as a building block.

Skyline Computation. Support for Skyline queries is another interesting example of thresholding functions. They have been initially proposed in [18] and gained popularity due to their applicability in multi-criteria decision making. Given a d-dimensional dataset D, the skyline is composed of points that are not dominated by any other point, with respect with some specific preference on (e.g., min, max) on each dimension. Various algorithms have been proposed in the literature for computing the skyline of a dataset distributed over a set of nodes [19; 20]. Some of the previous work [19] assumes a specific partitioning (e.g., grid-based, angular) of the points over the nodes. In this work, we make no assumptions regarding the data distribution over the nodes. We implement the same thresholding function API as follows. Each mapper computes a local skyline and maintains the local summary of the most "promising" data points. More precisely, the local summary is composed of the *k* points with the highest domination probability, as defined in [20]. The global summary contains the skyline computed from the union of the points contained in the local summaries. The intuition behind this approach is to identify points that are likely to make the greatest impact, i.e., dominate and filter out the most inputs. These points are shared with other map tasks to greatly reduce the number of false positives they produce.

Adaptive Sampling Another example of thresholding function is data sampling, which we already explored in [10]. In *Adaptive Sampling (AS)*, local aggregates collect samples of map output keys and the leader aggregates them into a global histogram. AS uses a pluggable threshold function to stop sampling if a sufficient sample has been accumulated. The default for this "stopping condition" function is to generate k samples. AS enabled us to produce balanced range partitioning of map outputs, which are required by the global sort operator.

In [10] we evaluated performance of AS and found it to be very robust and vastly superior to currently popular "client-side" sampling, which essentially executes the job on the sample of the dataset using a single node. In our experiments, client-side sampling had overhead of up to 250 seconds. In contrast, AS is able to leverage the full cluster. In our experiments, sampling was done in 5-20 seconds, with only a minor fraction of that time attributed to AS overhead, since we overlap sampling and query execution.

4 Conclusions and Future Work

We have presented adaptive techniques for the MapReduce framework that dramatically improve performance for user-defined aggregates. We expect SAMs to become an important MapReduce extension. Advanced users can implement essentially system-level enhancements using this mechanism directly. We demonstrated one approach for exposing SAMs to less advanced developers in a safe and controlled fashion. We plan to continue extending the programming API of Adaptive MapReduce and provide further integration points with Jaql.

References

[1] Apache Hadoop, http://hadoop.apache.org.

[2] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, 2010.

[3] A. Gates et al., "Building a highlevel dataflow system on top of MapReduce: the Pig experience," *PVLDB*, vol. 2, no. 2, pp. 1414–1425, 2009.

[4] A. Thusoo et al., "Hive - a petabyte scale data warehouse using hadoop," in *ICDE*, 2010, pp. 996–1005.

[5] Y. Bu and et al., "Haloop: Efficient iterative data processing on large clusters," *PVLDB*, vol. 3, no. 1, 2010.

[6] D. Jiang et al., "The performance of MapReduce: an in-depth study," *PVLDB*, vol. 3, no. 1, pp. 472–483, 2010.

[7] T. Nykiel et al., "MRShare: sharing across multiple queries in MapReduce," *PVLDB*, vol. 3, no. 1, pp. 494–505, 2010.

[8] "IBM InfoSphere BigInsights," http://www-01.ibm.com/software/data/infosphere/biginsights.

[9] K. Beyer et. al., "Jaql: A scripting language for large scale semistructured data analysis," in *VLDB*, 2011.

[10] R. Vernica, A. Balmin, K. Beyer, and V. Ercegovac, "Adaptive mapreduce using situation-aware mappers," IBM, Tech. Rep. RJ 10494, 2011.

[11] S. Babu, "Towards automatic optimization of MapReduce programs," in *SoCC*, 2010, pp. 137–142.

[12] G. Sagy et. al., "Distributed threshold querying of general functions by a difference of monotonic representation," in *VLDB*, 2011.

[13] H. Yu et. al., "Efficient processing of distributed top-k queries," in *DEXA*, 2005.

[14] A. Shatdal et al., "Adaptive parallel aggregation algorithms," in *SIGMOD Conf.*, 1995, pp. 104–114.

[15] Apache ZooKeeper http://hadoop.apache.org/zookeeper.

[16] P. Hunt et al., "ZooKeeper: wait-free coordination for internet-scale systems," in *USENIX Conf.*, 2010.

[17] "JSON," http://www.json.org/.

[18] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001.

[19] G. Valkanas and A. N. Papadopoulos, "Efficient and adaptive distributed skyline computation," in *SSDBM*, 2010.

[20] B. Cui et. al., "Parallel distributed processing of constrained skyline queries by filtering," in *ICDE*, 2008.

Challenges and Opportunities in Self-Managing Scientific Databases

Thomas Heinis, Miguel Branco, Ioannis Alagiannis, Renata Borovica
Farhan Tauheed, Anastasia Ailamaki

Data-Intensive Applications and Systems Laboratory, École Polytechnique Fédérale de Lausanne, Switzerland
{firstname.lastname}@epfl.ch

Abstract

Advances in observation instruments and abundance of computational power for simulations encourage scientists to gather and produce unprecedented amounts of increasingly complex data. Organizing data automatically to enable efficient and unobstructed access is pivotal for the scientists. Organizing these vast amounts of complex data, however, is particularly difficult for scientists who have little experience in data management; hence they spend considerable amounts of time dealing with data analysis and computing problems rather than answering scientific questions or developing new hypotheses. Therefore scientific experiments are in many ways ideal targets for research in self-managing database systems.

In this paper, we describe challenges and opportunities for research in automating scientific data management. We first discuss the problems faced in particular scientific domains using concrete examples of large-scale applications from neuroscience and high-energy physics. As we will show, the scientific questions are evolving ever more rapidly while datasets size and complexity increases. Scientists struggle to organize and reorganize the data whenever their hypothesis change and therefore their queries and their data changes as well.

We identify research challenges in large-scale scientific data management related to self-management. By addressing these research challenges we can relieve the burden of organizing the data off the scientists, thereby ensuring that they can access it in the most efficient way and ultimately enabling the scientists to focus on their science.

1 Introduction

We are entering the "data deluge" era where scientists are flooded with experimental data [3] because they use ever-faster hardware to run simulations and ever more precise instruments to observe phenomenon in space or nature. While this data is much easier to collect today due to advances in storage systems, our ability to organize and process this data has not kept pace with the growth; and scientists struggle to organize it for fast access. Drawing examples from observational science, the telescopes of the Sloan Digital Sky Survey (SDSS) record 73 TB raw observation data annually, while the instruments of CERN's large hadron collider (LHC) store 15 PB raw event data per year. In the simulation sciences, scientists are also struggling with data deluge. For example,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

already today the detailed models of the neuroscientists in the Blue Brain project (BBP) [2] are several hundreds of Gigabytes big and the simulation output has a size in the order of Petabytes (depending on the simulation length). Models as well as simulation data are predicted to grow by orders or magnitude in the coming years. In addition, while each of the aforementioned projects (SDSS, LHC and BBP) individually forecasts a tremendous growth in data, Gray et al. [3] predict that the amount of scientific data across all projects and disciplines will double every year.

Quick access to these data sets is crucial for the scientists as it helps them to speed up the cycle of experiment preparation, execution, analysis and refinement. Only thorough analysis of the data enables them to improve and refine their simulations or to calibrate their instruments, ultimately helping them to develop and corroborate new hypotheses. Deploying, maintaining and tuning a data management system for their perpetually changing query workloads and data, however, is a tremendously hard problem. The complexity as well as lack of support for complex data models discourage scientists from using DBMSs and often lead them to develop home-grown solutions that neither scale nor are efficient. Scientists are in dire need of scalable tools and methods for automated organization of the data, relieving them from having to do it themselves and ultimately allowing them to concentrate on their science again.

In this paper, we discuss the data management problems scientists in different disciplines face. We analyze how state-of-the-art methods can help scientists to tackle their data issues, but more importantly, we point out the gaps between what data management tools and approaches offer today and what researchers need to move ahead in their disciplines. As we show, these gaps translate into exciting research challenges related to automating data management and we sketch a roadmap for self-managing database management research for scientific data. In particular, we identify the following problems and their data management research challenges:

a) Amount of Data: the rapid growth of data volumes leads to two problems for scientists. First, having to wait until all data is loaded into the database system before they can analyze it keeps them from using database systems at all. Research into querying raw data files is hence needed so they can start querying their data instantly. Second, scaling into the cloud is a straightforward way to deal with the data volumes. Organizing the data in the cloud efficiently (partitioning, distributing, etc.), however, complicates the automated physical design problem, requiring more research in this area.

b) Fast-Changing Environment: many scientists constantly revise their hypothesis and hence they collect different data over time and they also continuously change the queries asked. Consequently, finding the proper physical design of the data is no longer a one-off operation, but instead has to be done continuously, challenging today's automated physical design methods.

c) Fine-grained Models and Complex Queries: the models used in simulation sciences grow equally complex. Today's indexes do not scale to the level of detail of their spatial models. Similarly, the questions scientists today ask about their data have become increasingly complex. No indexes exist to answer them efficiently and so scientists resort to using unscalable workarounds. New indexes are needed to answer the questions efficiently and thus add to the complexity of the automated physical design problem.

The rest of this paper is structured as follows: in the first part in Section 2 we identify data management problems in high energy physics and in neuroscience. In response to these problems, we identify the challenges involved in developing self-managing database systems for scientists in the second part in Section 3. We conclude in Section 4.

2 Data Management Problems in the Sciences

Data management challenges vary across different scientific disciplines. In the following we discuss the problems in two example disciplines, high energy physics from the observational sciences and neuroscience from the

simulation sciences. While in the former the problems are mostly related to the sheer size of the data resulting from experiments and the challenges are related to large-scale data organization, the challenges in the latter arise primarily from the growing complexity of the data and the questions asked about it.

2.1 High-Energy Physics

High-Energy Physics studies the fundamental constituents of matter and the laws governing their behavior. It relies on high energy probes or particle accelerators to break particles into their building blocks. The results are observed by large detectors and are analyzed to validate and formulate new theories. The ATLAS [4] experiment, one of the detectors of the Large Hadron Collider (LHC), collects tens of Petabytes of raw data per year. This data is being continuously analyzed by a large number of physicists in universities and research institutions worldwide. The data analysis for an experiment like ATLAS is a complex process (see [5] for a simplified view). The events at the end of this process are stored on a permanent basis and are then passed through a series of reconstruction and analysis steps to infer and calculate their properties. Events are also regularly re-analyzed given newer and more precise algorithms.

The main challenges for self-manageability in the LHC context are that (a) there is a need to store, share and analyze Petabytes of data and (b) it is a fast-changing environment, with regular modifications made to the software and hardware, but more importantly, depending on the hypothesis, researchers want to corroborate changes to queries and data. In the remainder of this section we focus on the two aforementioned data management challenges. We discuss the complexity of the currently-used storage solution in ATLAS and how the fast-changing environment makes the data management problem even more demanding.

2.1.1 Amount of Data

The LHC generates a total 15 PB of new data per year [6], most of which is raw data acquired by the detectors and is collected on a nearly continuous basis. The remainder is derived data products being produced during processing, simulated data and metadata (accelerator logs, conditions, geometry, calibration or alignment data etc). The derived data is significantly smaller than the raw data. In the case of ATLAS, the derived data products are either the detailed output of an event reconstruction with sufficient information to identify particles or summaries of the reconstructed event with sufficient information for common analysis. Currently the LHC uses a hybrid storage solution for its data, with some data stored in relational databases and the remaining in flat files using in-house data formats (e.g., POOL [7]). Flat files are used to store most of the event data using an object-oriented model, while relational databases contain high-level catalogs.

Just at CERN, one of the largest relational databases is the LHC accelerator logging database with over 50 TB. All relational databases are hosted in 100 Oracle RAC database clusters of 2 to 6 nodes. In total there are over 3000 disk spindles providing just over 3 PB of raw disk space for relational data. The remaining (event) data is stored in storage systems like the CASTOR [8] service, which uses a tape backend for long-term data preservation as well as a distributed file system for data access from batch processing nodes. CASTOR currently hosts approximately 50 PB of data in over 300 million files at CERN. A growth rate of Petabytes per year, data from various experiments, numerous files stored both in relational databases and using custom formats and the need to store and organize produced data in order to provide efficient access to participants from different institutions constitute only some of the parameters in the tremendously hard data management problem scientists face in the ATLAS project. Each of the problems enumerated before would pose significant manageability issues by itself; their combination demands automated systems.

2.1.2 Fast-Changing Environment

The high-energy physics environment and science exploration in general is subject to frequent changes. For instance, analysis algorithms are improved regularly and new features or applications are implemented in response

to scientific needs. The scientific needs and with it the queries and the data change quickly as the phenomenon the scientists study evolves. Each query initiates an exploration step in the data and its result might change the way scientists understand the data and the way they will access the data in future queries. Finding the optimal physical design therefore is not a static problem but a dynamic and ever-evolving process.

The infrastructure also changes at a rapid pace. There are multiple institutions participating, each with its own hardware acquisition and renewal cycles or software upgrades (including security patches). The LHC experiments rely on strict operational procedures to minimize the impact of changes. Integration and pre-production testbeds are used to ensure that changes are tested well in advance. Database administrators (DBA) are responsible for these tasks and any possibility for automation is important to alleviate the heavy workload.

2.2 Neuroscience

For at least two and a half thousand years, humans have tried to understand what it means to perceive, to feel, to remember, to reason and to know. In the last two centuries, the quest to understand the brain has become a major scientific enterprise. The increasing prevalence of brain disease has lent new urgency to this quest. To better understand and answer fundamental questions about the brain, neuroscientists have therefore started to build and simulate models of the brain on an unprecedented level of detail on an equally unparalleled scale.

For this reason, the neuroscientists in the Blue Brain project (BBP) [2] analyze real rat brain tissue in order to simulate an in-silico brain model on a supercomputer (BlueGene/P, 16K cores). Instead of simulating neural networks, the neuroscientists in the BBP already simulate a model at unprecedented levels of detail (with each neuron represented by several thousand cylinders) and now start to build even finer grained models. For instance, to better understand the propagation of impulses between neurons, they model the conductivity of the neurotransmitter at the subcellular level.

The small models used at the outset of the project occupy in the order of GBs and can easily be handled by state-of-the-art access methods. Today, however, neuroscientists build and simulate models of 100,000 or 500,000 neurons, amounting to 350 Gigabytes or 1.7 Terabytes respectively on disk. These sizes can no longer be handled efficiently by state-of-the-art indexes used to build and analyze the models. At the same time, today's models are still orders of magnitude smaller than a model of the entire human brain with 100 Billion neurons, the ultimate goal of the neuroscientists in the BBP. In the following we discuss how the increasingly detailed models as well as the increasingly complex questions challenge state-of-the-art indexing methods.

2.2.1 Fine-grained Models

In addition to scaling the size, neuroscientists also want to build and simulate the brain at ever more fine-grained detail, increasing the level of detail many times. When at the outset of the project each neuron was only modeled with several thousands of cylinders, today several thousand synapses are added to each neuron. The plan is to model the neurotransmitter as well as the organelles of the neurons at the molecular level, leading to several orders of magnitude more detailed/dense model.

Building ever more fine-grained models is also a trend in other sciences (models of the human arterial tree, lung airways, protein synthesis etc.). During many tasks in building and analyzing these models, scientists rely on the efficient execution of spatial queries on the models or on the simulation results. Spatial indexing and data partitioning tools at the scientists disposal, however, do not scale when it comes to analyzing the more detailed models due to the density of spatial datasets. Most state-of-the-art indexes, e.g., the R-Tree [10], build on a hierarchical organization of the spatial data. Like the R-Tree, however, they suffer from the problem of overlap, which slows down query execution considerably. Because overlap grows with increasingly fine-grained models, current index technology needs to develop new classes of spatial indexes which are oblivious to overlap.

2.2.2 Complex Queries

Not only the models, but also the questions the neuroscientists ask have become ever more complex. To analyze the models efficiently, simple spatial queries such as range or point queries are no longer enough to answer sophisticated scientific questions. For instance, the question "What postsynaptic synapses are close to a particular dendrite?" translates into a nearest-neighbor query where the neighbors must satisfy additional constraints.

The neuroscientist's workaround for complex queries is to implement ad-hoc solutions based on basic spatial queries. For the example query, they use a nearest-neighbor approach that retrieves near neighbors until enough neighbors satisfying the additional constraints are found. Clearly this means that many objects are retrieved unnecessarily and that by using the constraints earlier in the execution, the query could be executed substantially faster. As datasets grow exponentially, it is clear that workaround solutions for complex queries do not scale and that new indexes answering complex questions efficiently need to be developed.

3 Research Challenges in Automated Physical Design

Each of the challenges mentioned above (amount of data, fast-changing environments, need for fine-grained models and complex queries) significantly complicates the design of self-managing database systems; addressing all challenges simultaneously appears to be an unattainable goal in the short or even medium term. Nonetheless, in this section we describe promising avenues of research, each tackling part of the problem of developing self-managed databases for the sciences. In particular, we describe the research challenges in dealing with huge amounts of data, with fast evolution of queries and data, with increasingly fine-grained models in the simulation sciences and with complex queries. Addressing the challenges will relieve scientists from dealing with current data management technology, namely from defining a suitable schema, designing a proper physical organization and loading the data inside a DBMS. Because scientists typically work with data in an explorative way, i.e., their next query depends on the last, they need answers instantly and can only rarely rely on offline processing and analysis approaches for big data [21].

3.1 Coping with Large Amounts of Data

In order to support ever-growing data volumes and achieve high performance at the same time, DBMS today need to rely on parallelism [15]. A definite step toward this end is cloud computing, where farms of commodity machines are exploited to provide storage facilities and offer resources as a service. In the following, we discuss the challenge of scaling the data into the Cloud and addressing the associated problem of finding the proper physical design for databases distributed in such an environment. Then, we discuss an approach that bridges the chasm between scientists and database systems, by providing features for querying raw data files in-situ, i.e., without the need for data loading a priori, defining a schema or any other hurdle that repels scientists from using DBMS in the first place.

3.1.1 Scaling into the Cloud

Partitioning the data and distributing it in the cloud is the straightforward way of handling its sheer size. What makes the cloud computing additionally appealing to scientists is that it relieves them from worrying about purchasing the software, provisioning the hardware for their computation, maintaining hardware and software or providing reliability (through data redundancy or disaster recovery). Scientists can outsource their computation to public cloud providers and pay for the service usage on demand, exploiting a *pay-as-you-go* charging principle [14]. By sharing storage and computational capacities, and accordingly energy cost as well, customers are likely to pay much less than in case they buy all the facilities for themselves. Additionally, provided there is mutual trust among scientists, use of the Cloud also facilitates sharing their data and research results among

each other, while leaving the details such as data ownership, data confidentiality and data dependencies to the cloud providers.

Scaling into the cloud however also involves major research challenges, particularly in the area of physical database design. Configuring a DBMS residing on a single machine and choosing a proper physical design that will boost the performance of an a priori given training workload is a difficult task that requires profound knowledge of different physical design structures and their interactions. Configuring it correctly is of paramount importance, since the right physical design can improve performance of the running queries by several orders of magnitude. To replace the need of having a highly skilled and expensive expert, commercial DBMS vendors offer automated physical design tools that perform database tuning automatically [1; 16]. Despite being able to reduce database ownership cost significantly, existing automated physical design tools employ different heuristics to cope with the complexity even when considering a DBMS residing on a single machine [1; 16]. In a Cloud setting, the problem is further exacerbated by having additional parameters that need to be considered, like for example data ownership.

Because data movement operations in the cloud are much more expensive than relational operations on each machine, partitioning the data appropriately becomes a critical design problem. Additionally, in case of scientific applications, data related to the same experiment is often stored distributed on equipment owned by different institutions. Hence, during partitioning or moving operations, the data ownership has to be considered, adding a new dimension to the physical design problem.

By scaling into the cloud, the physical design problem becomes even more difficult because the cloud employs *multi-tenancy* , i.e., data owned by different tenants co-reside in the same database. Multi-tenancy introduces less predictability in the self-tuning process, and opens a new research area that deals with the problems of efficient resource virtualization between different tenants that co-reside on the same physical machine. Furthermore, an additional problem the cloud providers have to cope with now is unpredictability in the workload variations and how those changes affect the performance of the running applications.

Despite introducing considerable challenges in the physical design, the cloud brings significant opportunities for scaling massive amounts of data, which, considering the data deluge problem, becomes the right direction for the future.

3.1.2 Querying Raw Data

For very large experiments such as the LHC, the scientific data is usually stored in a combination of systems, ranging from relational databases to plain files in a distributed file system. Practitioners, particularly in the scientific field, often opt to store large amounts of data outside relational database systems due to concerns regarding scalability, cost or long-term data conservation.

We expect that relational databases will continue to coexist with non-relational data. In fact, it is possible that a smaller percentage of data will be stored in classical relational databases, as seen with the growth in popularity of systems such as Hadoop [21] or NoSQL databases. Part of the reason is that classical database systems can only be used after data is loaded and physical design is performed. These tasks require a significant investment and cause a delay between the time data is collected and the first queries are submitted (data-to-query time). As data collections grow larger and larger, performing the traditional extract/transform/load/query cycle is becoming tedious, impractical and at the extreme, infeasible.

Additionally, it is often not clear what particular share of data scientists need to analyze. Loading entire datasets to discard them after a few queries is an undesirable state of affairs. It is hence no surprise that initialization – along with tuning – is frequently cited as one of the most deterring factors for the adoption of DBMSs. As a result, many communities, especially in the scientific domain, revert to home-grown solutions, which are easier to manage but usually do not employ the sophistication of an advanced DBMS [17].

Nowadays, a DBMS can only be used once the data is entirely loaded into it – a time consuming and often unnecessary process. DBMS have long provided rich functionality on datasets loaded into a database, but only

limited capabilities are offered to query external files of raw data. It has become apparent that the database community needs to reconsider the loading prerequisite and redesign database systems to query data in-situ, i.e., in its original location and original data formats (plain files or relational schemas). The in-situ data processing model can significantly reduce the data-to-query time and lead to user-friendlier systems. Nevertheless, renouncing the benefits of data loading while maintaining the performance of a classic DBMS comes with many challenges such as (a) reduce the extra costs of accessing and analyzing the raw data for every query (b) improve data retrieval in future accesses to the raw data files (c) store the data into the proper format and adjust its access methods according to the incoming queries (d) make all this evolving process as transparent as possible to the user via lightweight adaptation actions (e) provide support for multiple data formats with different characteristics (e.g. HDF-5, NetCDF).

3.2 Coping with Fast-Changing Environments

Fast-changing environments additionally exacerbate the physical design problem. With the goal of proposing an optimal design for a given workload, existing physical design tools require the training workload to be known beforehand. Unfortunately, in the scientific domain, it is usually difficult to obtain a set of training queries a priori. The latter is an indirect consequence of the scientific discovery process, where research directions change frequently based on recently gathered knowledge. Regardless of a performance improvement it may bring, in rapidly evolving scientific repositories the effort of finding the proper design may be worthless, because the researcher's focus may shift and the queries may change from the ones for which the system is tuned.

It is hence evident that continuous, automated and adaptive tuning is important in scientific disciplines. In an ideal scenario, scientists would rely on autonomous techniques such as database cracking [18] to gradually create indexes on commonly accessed data, without any user intervention.

3.3 Coping with Fine-grained Models

With ever-faster hardware and better tools, the models scientists simulate become increasingly complex. The complexity manifests itself in the size of the spatial data sets they use but more importantly in their density (the number of elements per unit of space). As discussed previously, several types of spatial queries can no longer be answered efficiently on increasingly detailed models. Coping with fine-grained models is not just a problem in neuroscience, but affects all simulation sciences where scientists work with models. Examples include the simulation of the human arterial tree in computational fluid dynamics research, earthquakes in geology and protein synthesis in systems biology.

Indexes supporting scientists in the simulation sciences in executing distance joins [9], range queries [10] and others all build on the R-Tree or related tree-like index structures [11]. By building all these approaches on the R-Tree, however, we also inherit its inefficiencies, namely performance deteriorates due to overlap. Overlapping bounding boxes on one level lead to ambiguous paths through the tree. When executing a query, all paths need to be followed and overlap thus leads to an excessive number of pages read from disk. The number of pages read from disk defines the query execution time and hence reading too many pages slows down query execution substantially. Several approaches have been developed to tackle the overlap issue, all of them, however, have drawbacks.

For example, neuroscientists frequently ask spatial range queries to determine what neurons of the brain model are in a particular region. The results of this very basic query is used for scrutinizing interesting parts of the brain, for the purpose of analysis (tissue density), visualization etc. Many approaches to support range queries on spatial data have been proposed. Yet, state of the art approaches (R-Trees and variants [11; 10]) do not scale well for data sets of increasing density because the overlap increases. Several approaches have been developed to tackle the problem of overlap, each of them, however, has drawbacks. The R+-Tree [12] for instance requires excessive disk space.

Today's increasingly dense data sets suffer even worse from overlap, because with the increasing density of the data set also the overlap in the R-Tree increases. Clearly new indexing approaches that scale with increasing data set density must be developed. In addition, new indexing approaches for fine-grained models will further complicate the automated physical design problem. Not only does a decision need to be reached whether or not to use spatial indexes but also what index is suited best for a particular spatial query.

3.4 Coping with Complex Queries

While the models scientists develop become increasingly complex, so do the questions they ask about them. While this is a problem across different disciplines in simulation sciences (computational fluid dynamics, systems biology, etc.), we illustrate it with two particular examples from neuroscience: while the problem of nearest neighbor queries has been studied before, the nearest neighbor queries problem with constraints has not yet been solved efficiently. Similarly, the problem of moving range queries, where a scientist issues several range queries along a trajectory has not yet been studied.

The neuroscientists, for example, need to execute range queries interactively in close succession on a spatial model to analyze and visualize the surroundings of neuron elements along a neuron branch. The state-of-the art is to use a spatial index (e.g., the R-Tree [10]) and execute range queries on it repeatedly. Executing range queries repeatedly, however, is inefficient because every time a range query is executed, the overlap in the tree-like index structure introduces substantial overhead.

Research has so far only developed approaches which calculate safe regions for moving k nearest neighbor queries ([13] and others). If the query point moves into these regions, then the result does not need to be updated, thereby avoiding reading data from disk unnecessarily. The approaches developed for moving objects, however, do not support moving queries. New indexing approaches thus have to be developed that improve the performance of moving range queries, by making the execution of spatially close queries more efficient and by prefetching data.

Similarly, in advanced scenarios k nearest neighbor queries only need to retrieve the spatially closest elements that satisfy additional criteria. Neuroscientists will for instance use this type of query to retrieve the nearest postsynaptic neuron branches to a particular neuron. State of the art approaches do not execute this query efficiently [19; 20]. They either find a candidate set of nearest neighbors with a traditional kNN approach and then eliminate the candidates that do not satisfy the additional constraints, or filter all elements with the additional constraint and then find in the remainder the k nearest neighbors. None of these approaches, however, consider the pruning power of the additional constraints; and in the worst case all elements need to be visited. New kNN approaches therefore must make use of the pruning power of the additional constraint as early as possible in the query execution.

Many more indexes need to be developed to answer complex questions efficiently. Adding many potential candidate indexes of course also further complicates the problem of automated physical design.

4 Conclusions

In this paper, we identify the data management problem faced by researchers in two particular scientific disciplines, and identify the research challenges that need to be tackled in order to help scientists to manage their data automatically.

Because of the way scientists work with data, we identify the problems and the research challenges of continuous automated physical design and in-situ querying of raw data files. Research in the former is needed because scientists constantly revise their hypothesis, and hence data collected and queries asked change constantly. As a consequence, the physical design of the data has to be updated perpetually, challenging today's automated physical design methods. The latter is necessary because scientists often do not query all data collected. Importing

all data into a database hence is inefficient and there is a need for new in-situ querying approaches for raw data files.

We furthermore identify the problems of ever more fine grained models and complex queries in the simulation sciences. Both trends challenge current indexing approaches and open new opportunities for research. Developing new indexes at the same time complicates the physical design problem, challenging the state-of-the-art approaches in this area as well.

None of these challenges is trivial and will require considerable research. Nevertheless, interesting data management research problems can be tackled when helping scientists to automate their data management. Doing so will ultimately help scientists to focus on their research again.

References

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

[2] H. Markram. The Blue Brain Project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006.

[3] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. S. Szalay, D. J. DeWitt, G. Heber. Scientific Data Management in the Coming Decade. In SIGMOD, 2005.

[4] ATLAS Experiment. http://atlas.ch/.

[5] D. Malon, E. May. Critical Database Technologies for High Energy Physics. In VLDB, 1997.

[6] I. Bird. Computing for the Large Hadron Collider. In Annual Review of Nuclear and Particle Science, 2011.

[7] POOL - Persistency Framework. http://pool.cern.ch/.

[8] CERN Advanced STORage manager (CASTOR). http://castor.web.cern.ch/.

[9] T. Brinkhoff, H. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins R-Trees. *SIGMOD '93*.

[10] A. Guttman. R-trees: a Dynamic Index Structure for Spatial Searching. *SIGMOD '84*.

[11] V. Gaede and O. Guenther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), 1998.

[12] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. *VLDB '87*.

[13] X. Yu, K. Pu, and N. Koudas. Monitoring k-Nearest Neighbor Queries over Moving Objects. *ICDE '05*.

[14] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, N. Zeldovich. Relational Cloud: A Database-as-a-Service for the Cloud. In CIDR, 2011.

[15] D. J. DeWitt and J. Gray. Parallel Database Systems: The future of High Performance Database Processing. In Magazine Communications of the ACM, 1992.

[16] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellanom and S. Fadden. DB2 Design Advisor: Integrated Automatic Physical Database Design. In VLDB, 2004.

[17] S. Idreos, I. Alagiannis, R. Johnson, A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results?. CIDR 2011

[18] S. Idreos, M. Kersten, S. Manegold. Database Cracking. CIDR 2007

[19] G. Cong, C. S. Jensen, and D. Wu. Efficient Retrieval of the top-k Most Relevant Spatial Web Objects. *VLDB '09*.

[20] R. Hariharan, B. Hore, et al. Processing Spatial-Keyword (SK) Queries in Geographic Information Retrieval (GIR) Systems. *SSDBM '07*.

[21] D. Borthakur. The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/

Challenges and Opportunities for Managing Data Systems Using Statistical Models

Yanpei Chen, Archana Ganapathi*, Randy Katz
University of California, Berkeley, *Splunk
{ychen2,randy}@eecs.berkeley.edu, *aganapathi@splunk.com

Abstract

Modern data systems comprise of heterogeneous and distributed components, making them difficult to manage piece-wise, let alone as a whole. Furthermore, the scale, complexity, and growth rate of these systems render any heuristic and rule-based system management approaches insufficient. In response to these challenges, statistics-based techniques for building gray or black box models of system performance can better guide system management decisions. Although statistics-based approaches have been successfully deployed, a single model is often inadequate to capture intricacies of a single workload on a single system. The problem is exacerbated with multiple heterogeneous workloads super-positioned on a consolidated system. An even greater challenge is to translate the behavioral correlations found by statistics into insights and guidance for designing and managing even more complex data systems. In this article, we reflect on recent work on using statistics for data system modeling and management, and highlight areas awaiting further research.

1 Introduction

Large scale data systems are becoming ever more important. This is driven by increasingly economical access to large scale storage and computation infrastructure [4; 3], ubiquitous ability to generate, collect, and archive data about the physical world [11], and growing statistical literacy across many industries to consume, understand, and derive value from large datasets [1; 7; 15; 14]. The increasing ability to generate, record, and understand large scale data about the physical world is symbionic with the rising expertise to do the same for large scale data systems. Innovations in one area pioneer and inspire mirror innovations in the other. Statistical models have emerged as an essential tool for extracting knowledge about both physical and technology systems. This continues the historical relevance of statistics to scientific knowledge creation, in terms of inference, experiment design, hypothesis testing, and other ways to test and extend the limits of knowledge.

This technology context have led to the confluence of statistics and system design. In the past decade, a body of work emerged from both researchers and practitioners to use statistical techniques to mine data, model systems, understand systems, and manage systems [18; 9; 6; 13; 8; 12]. This effort is in response to the increasing complexity of data systems and the workloads they service, both composed of heterogeneous and often distributed components. Domain specific heuristics no longer scale, and no single domain expert can be consulted for all system-related decisions. An effort to scientifically understand these systems brings rewards far

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

outweighing the costs of re-engineering and the penalties of mis-engineering. Using statistical models becomes essential for designing and operating these systems.

There are a variety of commonly used statistical techniques, ranging from simple linear regression to more complex high-dimensional analysis. These techniques vary greatly in the amount of system specific knowledge they require and the system visibility they provide. A key challenge arises in identifying the boundaries of the system being modeled, what specific technique to use, how to adapt the statistical technique to serve the specific system management problem, how to evaluate the statistics-driven optimizations, and how to translate statistics to new knowledge about the data systems.

This article discusses these challenges and how they have been partially addressed by recent research. We focus on a few case studies to highlight in depth the challenges associated with defining system boundaries (Section 2), building statistical models (Section 3), evaluating statistical optimizations (Section 4), and turning statistics to new knowledge (Section 5). The case studies are mostly drawn from the authors' own work, because we are familiar with their limits. We highlight open problems and identify research opportunities that would benefit the general data system management community.

2 Defining a System

A *system* is a conceptual unit to which we can feed input, modify configurations, and observe performance under the given input and configuration. System input can be a single request, or a set of requests, often referred to as *workload*. Various metrics can characterize individual requests or entire workloads, ranging from request type, request size, to interarrival rate between requests, and others. The scale and mix of such requests often control how much work a system must perform, what kind of work they perform, and which configurations are the most appropriate. For example, in a relational database, queries serve as the input and sets of queries form a workload. The number, type, and arrival pattern of queries dictate the work required of the database, which can then be tuned for a particular workload.

System *configuration* often exposes tradeoffs to optimize system performance. For example, configuring a machine to use a subset of its cores for a particular application versus all its cores for that application would allow users to trade response time for resource sharing across different workloads. For another example, data systems often have logical configuration parameters to control data layout. One can control the layout of data among the nodes on a parallel relational database system, and different layout policies have an impact on resource utilization and query execution time.

System *performance* is an umbrella term that can be used for any observed behavior that is a consequence of running an input workload under a particular configuration. Some examples of performance metrics include execution time, CPU utilization, cache hits/misses, disk I/Os and latency. These metrics may be collected periodically, every minute for example, aggregated over a time window, such as average CPU utilization over a one minute window, or aggregated for an entire workload, as in the case of a performance summary for a database.

This simplistic definition of a system is extensible. A system composed of multiple subsystems has a set of input and configuration parameters for the entire system, which divides into input and configuration parameters for each subsystem. The performance of the entire system is a composition of the performance of each subsystem. Although the definition is simplistic, modeling such complex multi-component systems is non-trivial.

3 Modeling the System

The increasing complexity of data systems and the workloads they service create challenges for modeling system behavior. We illustrate some challenges with three case studies. The first seeks to predict storage subsystem performance at the IO request level and aggregate workload level [18]. The second analyzes two enterprise

(a) 15-dimensional description of storage system workloads in [18]

1. Average arrival rate
2. Read ratio
3. Average request size
4. Percentage of sequential requests
5. Temporal and spatial burstiness
6-15. Correlations between pairs of attributes

(b) 41-dimensional description of directory-level access patterns in [9]

1-3. Number of hours with 1, 2-3, or 4 file opens
4-6. Number of hours with 1-100KB, 100KB-1MB, or >1MB reads
7-9. Number of hours with 1-100KB, 100KB-1MB, or >1MB writes
10-12. Number of hours with 1, 2-3, or 4 metadata requests
13-15. Read request size - 25th, 50th, and 75th percentile of all requests
16-18. Write request size - 25th, 50th, and 75th percentile of all requests
19-21. Avg. time between IO requests - 25th, 50th, and 75th percentile of all request pairs
22-24. Read sequentiality - 25th, 50th, and 75th percentile of files in the subtree
25-27. Write sequentiality - 25th, 50th, and 75th percentile of files in the subtree
28-30. Read:write ratio - 25th, 50th, and 75th percentile of files
31-33. Repeated read ratio - 25th, 50th, and 75th percentile of files
34-36. Overwrite ratio - 25th, 50th, and 75th percentile of files
37. Read sequentiality - aggregated across all files
38. Write sequentiality - aggregated across all files
39. Read:write ratio - aggregated across all files
40. Repeated read ratio - aggregated across all files
41. Overwrite ratio - aggregated across all files

(c) 100,000-plus-dimensional description of datacenter state in [6]

1-100: Machine 1 CPU 25th, 50th, 75th percentiles, memory 25th, 50th, 75th percentiles, ... active threads, page swaps, ...
101-200: Machine 2 CPU 25th, 50th, 75th percentiles, memory 25th, 50th, 75th percentiles, ... active threads, page swaps, ...
...

Table 2: Model complexity for three studies. The number of dimensions corresponds to the complexity of the systems and workloads being modeled

network storage traces to extract common access patterns, which facilitates identifying highly targeted system optimizations [9]. The third constructs "fingerprints" of datacenter performance crises from historical traces, and uses them to automatically classify future performance crises in real time [6].

The common challenge all these studies confronted is *model multi-dimensionality*. This challenge arises from the simple fact that the systems and workloads being modeled are complex, and therefore need to be described in multiple ways, i.e., multiple dimensions. In other words, the number of dimensions corresponds to the complexity of the systems and workloads being modeled. For example, study [18] models block level storage workloads on disks and disk arrays. The models there used 15 dimensions (Table 1(a)). Study [9] models session, application, file, and directory level storage workloads on client-server network storage systems, an increase in both system and workload complexity. The models there involved up to 41 dimensions (Table 1(b)). Study [6] models general datacenter workloads on the entire datacenter, a further increase in system and workload complexity. The models there involved more than 100 dimensions for each machine, combined across 1000s of machines (Table 1(c)). Multi-dimensionality is *inevitable* for any rigorous attempts to model complex systems and workloads.

Working in multiple dimensions creates a heavy cognitive load, and one instinct is to use as few dimensions as possible, or even resort to heuristics. This approach carries great risk because it introduces *designer bias*, another common challenge. Bias arises from the human system designer's potentially incorrect assumptions and mental models, built up by historical experience, but needs constant re-evaluation as systems and workloads

rapidly evolve. Any subjective choice of which dimensions to keep and what heuristical reasoning to apply inevitably involves some assumptions about how systems and workloads behave. All three studies in [18; 9; 6] consciously avoided this approach. They begin with a large number of dimensions, then perform *dimensionality reduction* through classification, clustering, regression, correlation analysis, and other rigorous statistics techniques. Doing so controls the cognitive load in interpreting the models, while minimizing human designer bias.

A further challenge is the necessity to deal with *outlier behavior*. Complex systems servicing complex workloads are prone to introduce rare but regularly appearing outliers. These outliers cause "classical" statistical models to be influenced by rare data. For example, a sudden spike in workload arrival patterns may cause a single system query to take several orders of magnitude longer, causing the arithmetic average query duration to double. For many use cases, this inflated kind of average is not appropriate. The solution is to use *robust statistics*, an active field of statistical research that is gaining in prominence. Some examples of robust statistics include quantiles (general case of percentiles, used in [6]), cluster medians (instead of arithmetic means, used in [9]), and piece-wise models (instead of global models, used in [18]).

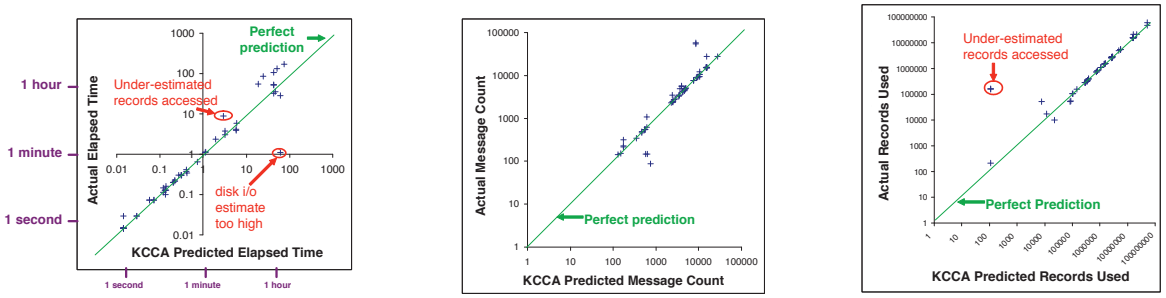
Generally speaking, most well-constructed statistical models can be shown to be effective for some highly specific use cases, even if those models do not adequately address the challenges mentioned here. However, such models create doubt regarding whether they generalize to other use cases, and remain useful as both systems and workloads evolve. The challenges mentioned here require some effort to address, even though solutions to them are more or less known. The rewards of addressing these challenges represent models that survive transient behavior pathologies, are easier to understand and accept, and have a greater likelihood of keeping up with the continuous growth in system and workload complexity.

4 Evaluating a Proposal

Once the system has been defined and a model chosen, there are two possible next steps. If the research contribution is the system model itself, then the quality of the model needs to be evaluated. Alternately, if the contribution is applying the model to improve system performance, then the evaluation needs to demonstrate both the quality of the model and the improvement in system performance.

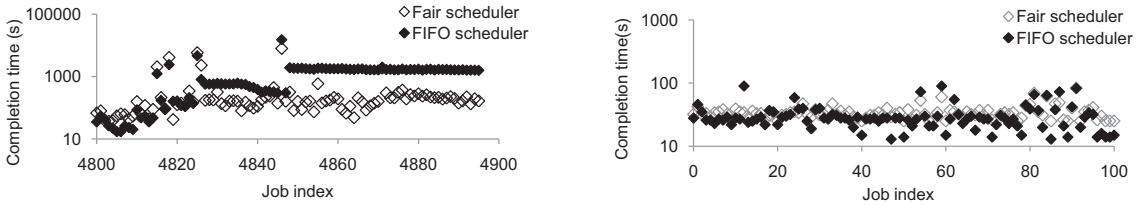
An example of evaluating statistical system models is in [13]. The work seeks to develop a multi-dimensional system performance model for database queries. The key statistical tool used is kernel canonical correlation analysis (KCCA). At a high level, KCCA finds dimensions of maximal correlation between an input dataset of query descriptions and an output dataset of query behaviors [5]. Both the input and output datasets are multi-dimensional. Evaluating the system model requires demonstrating that KCCA predicts system behavior that approximates actual system behavior. System behavior traces originate from running an extension of the TPC-DS decision support benchmark [16]. The trace then divides into training and testing datasets, a standard model evaluation technique in statistics. Graphs of predicted versus actual behavior demonstrate the accuracy of the models across multiple dimensions, including query time, message count, and records used (Figure 1).

An example of demonstrating improved system performance is in [8]. The work introduces realistic workload suites for MapReduce [10], and uses them to compare the performance of the default MapReduce FIFO task scheduler versus the MapReduce fair scheduler [19]. A fixed workload is replayed under each MapReduce scheduler, and the system performance behavior is observed and compared. In this case, the statistical model is about the input workload, not on the system, and the research contribution is on accurately capturing system behavior subject to realistic workload variations. Such variations complicate performance comparison, in that some conditions favor one system setting, while some other conditions favor other settings. This is true even for simple performance metrics such as job completion time. Figure 2 shows the fair scheduler gives lower job completion time than the FIFO scheduler under some workload arrival patterns, and vice versa under other arrival patterns. Thus, the choice of scheduler depends on a rigorous understanding of the workload.



(a) Query elapsed time (b) Message count (c) Records used

Figure 1: Evaluating system model - predicted vs. actual behavior. Graphs reproduced from [13].



(a) Fair scheduler has lower job completion time. (b) FIFO scheduler has lower job completion time.

Figure 2: Evaluating system performance under two settings - MapReduce FIFO scheduler vs. fair scheduler for two different job sequences. Graphs reproduced from [8].

Both of the above studies confront similar challenges. First is the challenge of *workload representativeness*. If the evaluation covers workloads that do not represent real life use cases, then there would be little guidance on how evaluation results translate to real life systems. This challenge is especially relevant for the study in [8], where the choice of the optimal MapReduce scheduler depends on the particular mix of certain arrival patterns. For the study in [13], the standard TPC-DS benchmark is augmented with additional queries that are informed by real life decision support use cases. Such knowledge about real life use cases arises from either empirical analysis of system traces, or from system operator expertise. As data management systems become more complex and more rapidly evolving, it is likely that operator expertise alone would become insufficient, and good *system monitoring* becomes pre-requisite for good system design.

Another challenge is the need for *continuous model re-training*. In [13], both the training and testing datasets come from the trace. This setup does not translate to a real life deployment, in which only the training dataset is available and the test set is generated in real time as queries are submitted to the system. For example, the query prediction model is trained once per configuration and the system takes some action based on the predicted resource requirements for a new query. However, the resulting behavior of the system's response to the new query is not accounted for when the model is static. The concept of a "test dataset" is ad hoc and should be constantly updated by subsequent queries. This is an inherent shortcoming of system behavior models, well-studied in the Internet measurement literature [17]. Consequently, it is desirable to have statistical models that can *constantly re-train parameter values* or even *discover new parameters*.

A third challenge is imposed by the *limitations of system replay*. Replay here implies executing a workload on a real system, with the system making control actions using statistical models. The study in [8] demonstrates this approach. Replay allows designers to explore the interaction between model re-training and system actions that affect the model. However, as data management systems grow in size, replaying long workloads at full scale and full duration becomes a logistical limit. For example, the experiments in [8] required using a 200-machine cluster for several days. Preparing the experiment required additional days of debugging at scale. Even

then, the replay is not at the full scale of the original system that was traced. There is a spectrum of replay fidelity, from comparison experiments on the actual front-line, customer-facing systems, using production-scale data and covering long durations, to scaled-down experiments using artificial data and covering short durations. Ultimately, the system designer needs to judiciously select the *appropriate replay fidelity*, balancing the need for quality insights, logistical feasibility, potential for improvements, and risk of negative system impact.

5 Statistics to Knowledge

The discussion so far has covered the challenges associated with modeling systems and evaluating those models. There is a deeper issue that must be confronted - how to distill knowledge from statistical analysis. Fundamentally, statistical analysis only leads to correlations between workloads, system behavior and performance characteristics. This fact is true even for rigorous hypothesis testing experiments. Statistics help us understand *how* systems behave. There is an inevitable methodological and scientific "leap of faith" to translate from system behavior statistics to some deeper insights about *why* systems behave thus.

To illustrate, consider the study in [12]. This is a follow-up study to [13], and applies the KCCA prediction technique to predict ad-hoc queries running on MapReduce extensions, an increasingly popular complement to traditional databases. The difference in the nature of systems considered in [12] and [13] requires two different model formulations, in that the same KCCA technique need to be applied on two different kinds of system descriptions. While the KCCA technique helps draw statistical correlations between multi-dimensional workload and performance vectors, there is considerable pre-requisite knowledge involved with how to formulate the modeling problem, where to draw the system boundaries, and what actions to take based on the KCCA prediction outputs. Such insights are not supplied by statistical models directly, rather by human designers interpreting the significance of statistical models.

It is non-trivial to interpret the significance of statistical models, especially as systems increase in complexity, and the dimensions of the statistical models likewise grows. Standard dimensionality reduction techniques help somewhat, but ultimately shift the problem from interpreting multiple dimensions to interpreting the dimensionality reduction process.

To illustrate, consider the study in [9], introduced earlier in Section 3. Recall that the study analyzes two enterprise network storage traces to extract common access patterns, which facilitates identifying highly targeted system optimizations. One analysis computes multi-dimensional descriptions of files accessed, and uses k-means clustering [2] to identify the most common types of files. The output of k-means clustering is a set of multi-dimensional cluster centers, as in Table 3. Even though k-means reduced a multi-dimensional space to just six common access patterns, human expertise is needed to interpret the significance of these clusters. Deriving the human applied labels in the lowest row in Table 3 required both examining the numerical values of cluster centers, and computing additional information, such as file types and file co-location within the directory structure. This interpretation step is far more cognitively demanding than applying the k-means algorithm, and arguably represents a transition from a scientific description (statistics) to a scientific taxonomy (knowledge) of system behavior.

More generally speaking, it is desirable to leave multiple opportunities for *human intervention*, both while building statistical models of systems and while using these models to control systems. Human intervention is especially important when multiple performance metrics are involved, and multiple human users have different metrics. The goal of statistical models and statistically managed systems would be more to help mediate potentially conflicting requirements from human users, rather than optimizing for some abstract multi-dimensional performance metric to which the human users cannot relate. Human intervention also offers additional opportunities to detect "black swan" phenomena, which are rare system events that have large performance impact, yet are difficult to statistically identify. A balanced approach would augment statistical methods with human expertise, and vice versa.

File access patterns	Pattern 1	Pattern 2	Pattern 3	Pattern 4	Pattern 5	Pattern 6
% of all files	59%	4.0%	4.1%	4.7%	19%	9.2%
# hrs with opens	2hrs	1hr	1hr	1hr	1hr	1hr
Opens per hr	1 open	2-3 opens	2-3 opens	2-3 opens	1 open	1 open
# hrs with reads	0	0	1hr	0	1hr	1hr
Reads per hr	-	-	100KB-1MB	-	1-100KB	1-100KB
# hrs with writes	0	1hr	0	1hr	0	0
Writes per hr	-	100KB-1MB	-	1-100KB	-	-
Read request size	-	-	4-32KB	-	2KB	32KB
Write request size	-	60KB	-	4-22KB	-	-
Read sequentiality	-	-	70%	-	0%	0%
Write sequentiality	-	80%	-	0%	-	-
Read:write ratio	0:0	0:1	1:0	0:1	1:0	1:0
Human applied labels	Metadata only	Sequential write	Sequential read	Small random write	Smallest random read	Small random read

Table 3: Multi-dimensional descriptions of enterprise storage file access patterns from [9].

6 Challenges and Opportunities

Managing data systems using statistical models is becoming an increasingly important topic. The sheer size and complexity of data systems today necessitates some sort of statistical technique to help design and operate such systems. Also, as data systems support essential day-to-day services, it becomes increasingly important to develop accurate behavior and performance models. We believe the broad effort to manage data systems using statistical models remains very much in a nascent stage. We highlight below some opportunities for future research.

One current logistical challenge is the limited availability of large-scale system traces. As discussed in Section 4, the ground truth of constructing and evaluating statistical models arises from monitoring the relevant data systems. However, there are few publicly available traces of large scale, consumer facing systems. This challenge also offers an opportunity, especially for practitioners in industry. The lack of publicly available traces means that organizations who publish their traces stand to influence the direction of the field. Conversely, researchers should keep in mind that the availability of certain kinds of traces does not indicate that those systems are representative. As system tracing capabilities improve and data anonymization tools become prevalent, this challenge can gradually subside.

Another opportunity arises in accelerating the statistically informed data system design loop. Better statistical models lead to better designs and better insights, which in turn lead to better models. As discussed in Section 5, it is non-trivial to make the cognitive advance from each step to the next. One solution would be to identify a small set of robust, scalable, general-purpose statistical tools, with some best-practices and good-defaults on how to apply them, while leaving ample opportunities for human intervention. Another solution would be to devise intuitive visualizations of the modeled statistical behavior. When such visualizations are possible (Figure 2), the human cognitive load vastly decreases. Conversely, it becomes burdensome to interpret multi-dimensional data in a purely numerical fashion (Table 3).

Managing data systems using statistical models requires collaboration between computer system designers and statisticians. An additional challenge lies in the various research incentives for each field. Traditionally speaking, computer system designers get rewarded for building better systems, while statisticians get rewarded for discovering new algorithms or new modeling techniques. The overlap between the two fields is increasing, and both communities are broadening their research agenda. Some examples covered in this paper merely seek to develop a statistical understanding of existing systems instead of constructing new systems. Others merely apply well-known, relatively straightforward statistical methods to assist system design instead of developing new statistical techniques. Practitioners of both fields have much to learn from one another. We are hopeful that the emerging body of work on managing data systems using statistical models helps highlight the growing necessity and rewards for bridging the two fields.

The scale and heterogeneity of today's data systems and the workloads they service complicate system design and operation. As more people are required to build and maintain these systems, statistical techniques have served the systems community well in quantifying the intricacies of data systems. This article discusses some challenges and opportunities that hopefully help inform and guide future studies in the area. We are confident that the confluence of systems and statistics research provides a launching pad for moving the art of system design towards becoming a science.

References

[1] Hadoop World 2011 Speakers. `http://www.hadoopworld.com/speakers/`.

[2] E. Alpaydin. *Introduction to Machine Learning*. MIT Press, Cambridge, Massachusetts, 2004.

[3] Amazon Web Services. Amazon Elastic Computing Cloud. `http://aws.amazon.com/ec2/`.

[4] Apache. Apache Hadoop. `http://hadoop.apache.org/`.

[5] F. R. Bach and M. I. Jordan. Kernel independent component analysis. *J. Mach. Learn. Res.*, 3:1–48, March 2003.

[6] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *EuroSys 2010*.

[7] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop goes realtime at Facebook. In *SIGMOD 2011*.

[8] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *MASCOTS 2011*.

[9] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *SOSP 2011*.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[11] EMC and IDC iView. Digital Universe. `http://www.emc.com/leadership/programs/digital-universe.htm`.

[12] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *ICDE Workshops 2010*.

[13] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE 2009*.

[14] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP 2009*.

[15] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. In *VLDB 2010*.

[16] R. O. Nambiar and M. Poess. The making of TPC-DS. In *VLDB 2006*.

[17] V. Paxson and S. Floyd. Why we don't know how to simulate the internet. In *WSC 1997*.

[18] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with cart models. In *MASCOTS 2004*.

[19] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys 2010*.

SQL Azure as a Self-Managing Database Service: Lessons Learned and Challenges Ahead

Kunal Mukerjee, Tomas Talius, Ajay Kalhan, Nigel Ellis and Conor Cunningham
Microsoft Corporation

Abstract

When SQL Azure was released in August 2009, it was the first database service of its kind along multiple axes, compared to other Cloud services: shared nothing architecture and log-based replication; support for full ACID properties; providing consistency and high availability; and by offering near 100% compatibility with on-premise SQL Server delivered a familiar programming model at cloud scale. Today, just over two years later, the service has grown to span six hosting regions across three continents; hosting large numbers of databases (in the order 100s of thousands), increasing more than 5x each year with 10s of thousands of subscribers. It is a very busy service, clocking more than 30 million successful logins over a 24 hour period. In this paper we reflect on the lessons learned, and the challenges we will need to face in future, in order to take the SQL Azure service to the next level of scale, performance, satisfaction for the end user, and profitability for the service provider.

1 Introduction

These are exciting times to be working on cloud services. The buzz has seeped beyond technical forums, and is now a common and recurring theme in the popular media [1]. Economic conditions seem to also favor widespread interest in leveraging cloud services to start new businesses and grow existing ones with frugal spends on infrastructure [2]. Since it went live in August 2009, SQL Azure, which was the first large scale commercial RDBMS server (Amazon's Relational Database Service, or RDS, which is a distributed relational database service, was first released on 22nd October, 2009), has been growing steadily to accommodate both external and (Microsoft) internal [e.g., see [13]) customers and workloads. Given that the SQL Azure service was the first of its kind along several different axes [3], [4], and few others have ventured into providing this specific mix of architectural orientation and service guarantees [5], we feel that the time is right to reveal what have been our internal pain points, lessons learned, and perception of what the future holds, as we continue to grow the service to support a larger customer base, scale to store and query bigger data, and push ahead on performance and SLAs. The rest of the paper is organized as follows: The rest of Section 1 first introduces the canonical application pedigree and customer expectations currently employing the service; and then sets the SQL Azure architectural context, specifically for detecting and handling failures throughout the service and underlying infrastructure. From the very outset, SQL Azure has been serving industry strength workloads. Its most widely publicized incarnation has been the SQL Azure service [7], in which a large number of independent databases are served. Additionally, we hosted the Exchange Hosted Archive [9] on it, in which the infrastructure

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

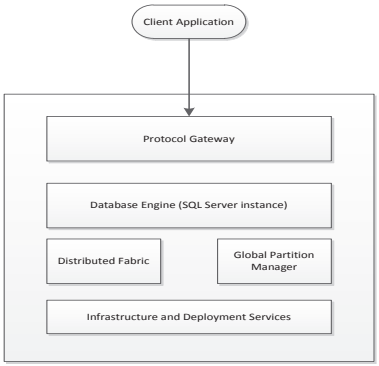


Figure 1: Overall system architecture

serves a massively scaled, multi-tenant, email archival service. As a result, we hit the ground running in terms of handling scale on all imaginable areas of the architecture, and that taught us a number of lessons in these past two years. We will describe some of these in Section 2. We describe challenges and future directions for the service in Section 3. Section 4 has our conclusions.

Canonical SQL Azure database: OLTP consolidation. In order to facilitate understanding of the material presented in this paper, it would be useful to introduce the reader to the kinds of databases, application contexts, and primary customer expectations our service aims to serve. OLTP consolidation is a predominant category of database applications being currently deployed on SQL Azure. The interesting characteristics of these are: 1) Each database's resource requirements are smaller than a single node in the cluster; 2) Relatively small nodes are used; and 3) Customers want to pay-as-they-go. Additionally: What pieces of the service experience do the bulk of SQL Azure customers place the most premium on? Our customers want a great deal. This is corroborated by the study conducted in [14], where Kossmann et al. concluded that Azure is currently the most affordable service for medium to large services. Our customers expect, and get this "good deal for the money" on commodity hardware, automatic management of core pieces of the system, and built-in High Availability (HA). We should note here, that SQL Azure is being used for non-OLTP scenarios as well. There are at least two Microsoft teams that are using it as their "OpsStore". TicketDirect is a good representative case study [15] of OLTP consolidation applications that are running on SQL Azure. Exchange Hosted Archive [13] is a representative of a non-OLTP workload (email archiving).

SQL Azure: Fault Tolerance Architecture. Cost factors have required delivering highly reliable services over inexpensive hardware with an operational model that is fully automated, in SQL Azure. Furthermore, failures of individual components are literally an everyday occurrence at this scale. To be a viable business for us, as service providers, the system must handle most faults without operator intervention. Figure 1 shows the top level view of SQL Azure's system architecture. As we explain below, the fundamental pillars of self-management and fault tolerance permeate each level of the system architecture.

Typically all user data is copied on three replica partitions. These data partitions are spread through multiple fault domains (racks and switches in the data center) and upgrade domains (units of software rollout), so that multiple replicas of the same partition are unlikely to go down at the same time. Each partition has at most one primary replica at any given moment of time, the other replicas are either secondary or are inactive. All user reads and writes originate at the primary replica.

The *Protocol Gateway* enables client applications to connect to the SQL Azure service. It coordinates with the distributed fabric to locate the primary, and renegotiates that choice in case a failure or system-initiated reconfiguration causes the election of a new primary. The *Database Engine*, which is an instance of SQL Server, typically manages multiple locally attached disks, each of which is private to that instance. If one

of the instance's disks fails, the watchdog processes which monitor hardware health will trigger appropriate failure recovery actions. In the SQL Azure database server, partitions are known as the *failover unit*. To attain high availability, we designed a highly reliable system management layer that, among other things, maintains up/down status of servers, detects server failures and orchestrates recoveries. These capabilities are built into the *Distributed Fabric* layer, which runs as a process on every server that runs a SQL Server instance. There is a highly-available *Global Partition Manager (GPM)* architectural module, which maintains a directory of information about partitions. For each partition P, there is a cluster-wide unique id. And for each replica R of P, it knows R's location and its state. If a hard failure is detected at any secondary, a new node is appointed and a copy of the partition is created on that node. If a primary dies, the GPM will elect one of the remaining secondary nodes to become the primary. Finally, the *Infrastructure and Deployment Services (IDS)* layer is responsible for hardware and software lifecycle management and monitoring. Hardware lifecycle includes such activities as taking bare metal machines and preparing them to play a role on the cluster, monitoring machine health, and taking it through repair cycles: reboot, reimage and return to manufacturer (RMA). The software lifecycle runs workflows to clean a cluster, to specify machine roles [4], deploy application software, deploy OS patches, etc.

Self-healing vs. self-managing. Our experience with boxed products shows that humans are the largest factor in operational cost. In a conventional setting you need humans to set up High Availability, orchestrate patches, plan upgrades, deal with hardware failures, etc. Cloud database systems need self-healing capabilities in order to scale to large numbers of clusters and large number of machines in each cluster. Without this we don't get efficiencies of scale. Current SQL Azure self-managing capabilities are in the following areas: 1) There are a number of cases in which SQL Server detects hardware failures. We convert these high severity exceptions into node failures; 2) System managed backups; 3) Periodic snapshots of SQL Server state on each node. These are used for trending and debugging; and 4) Throttling to prevent user requests from overwhelming the machine.

Experience from the box product also shows that managing database performance requires human intervention. In this context we will need to invest in traditional self-tuning capabilities. This includes auto-defragmentation, auto-indexing, etc. The multi-tenant nature of SQL Azure requires self-tuning capability in the form of resource governance. We have to ensure that workloads don't adversely affect each other and a workload which requires resource guarantees can be satisfied. Database systems will require a fair bit of investment to make them behave well in a multi-tenant setting. This is therefore, a rich area for research.

2 Lessons in Self-Management and Fault Tolerance

In the previous section, we outlined how the overall system design is fundamentally geared to support fault tolerance, and to favor automatic, rather than manual mitigation and fix of any failure. In this section we discuss a number of specific dimensions that are identifiable towards this general theme, leveraging the architecture, as well as service operations (i.e., people and processes).

After running the SQL Azure service over two years, we have come to realize that everything in hardware and software fails sooner or later, and sometimes in bizarre and unpredictable ways! And so the emphasis of the monitoring and process layer surrounding the hardware and software architecture outlined in Section 1 is primarily concerned with generating early warnings, detailed metrics at different granularities, and stopping contagion from failures, rather than trying to come up with ways to avoid failures from ever happening. Here we outline the major thrusts of this policy.

Define the metrics to instrument and monitor the cluster effectively. The right mix of health metrics should be detailed enough, provide aggregate views across logical and/or physical groups of machines, and yet be able to drill into failure cases and hotspots to effectively monitor from a compact dashboard.

An overall emphasis of our monitoring strategy is to accurately detect and react rapidly to black and white failures every time, as well as detect "gray area" failures, and coerce them to either flag a serious enough condition where the component should be shut down completely, or to realize that it is a benign condition. A philosophical explanation of the above strategy is that we accept that there will always be gray areas in the failure landscape, and so canned responses can never be truly comprehensive. Our approach to this problem is to install redundant checks to increase confidence in the type and severity of each failure.

We use a mix of logging, automatic watchdogs that trigger on failure and near-failure conditions, as well as alerts that trigger emails and/or phone calls for severe and/or sustained failures. The machines originating failure are also taken through a three step escalation protocol, in which increasingly aggressive steps are taken to try and fix the problem: 1) reboot; 2) reimage; and 3) return to manufacturer (RMA). This provides a hierarchy of escalation, where if the service, by virtue of its intrinsic and designed resilience to failure can recover from a failure state, would not require human intervention – this is what is required in order to scale the service up, keep costs down for our end users (see also Section 1.1), and for us to sleep at night!

Take a calibrated approach to dealing with serious failures. Coming up with graded responses to failures of different varieties is something of a black art, which practitioners of a service learn over time.

To use specific examples from running SQL Azure: serious software issues like a program hitting an Access Violation (AV) are typically highly localized to a single machine and in response to a very specific mix of workloads. Under these failure conditions that are highly localized, it is generally appropriate to bring the machine down quickly, because otherwise, persisting in the AV state where wild accesses to memory may be a possibility, even more serious consequences like index corruption are reachable. This may be summarized into the following rule of thumb: *fail fast when the failure conditions are localized*.

However, in other situations, failing fast isn't the best approach, because it could lead to cascading failures. For example, one ought not to bring machines down due to bad or invalid configuration, because configurations are common across many machines, and that could result in (automatically) leading to more widespread failures such as quorum loss [4], or even bringing the entire service down.

Due to the above intricacies of taking carefully calibrated responses to failures, we have a role for central command and control, which allows humans in the loop to assess the impact of the failure/repair action(s), before initiating them. Another best practice that has evolved, simply as a result of running a database service, is to disallow combinations of input data and SQL statements that are already known (because they generated failure either on the live service or on a test cluster) to be generate problems or failures, until a fix (QFE) has been rolled out. SQL Azure has a mechanism known as *statement firewall*.

To generalize, the lessons we learned with respect to calibrated responses to failures is that *cascading failures are to be avoided* at all cost, and that safeguards need to be put in place to stop them. Once the scope of the failure exceeds some standard deviation (automation can help to figure out that is happening), the recommended policy is to get humans involved, rather than continuing to try to auto-resolve. We actually have an explicit abstraction known as the *Big Red Switch* in the system, which is our mechanism for turning off automatic repair or compensation operations when the system is clearly experiencing an unplanned problem.

Software-based defense in depth is appropriate when building on commodity hardware. As explained in Section 1.1, [3] and [4], one of the fundamental design points of SQL Azure was to build the entire service on top of commodity hardware, in order to provide an attractive price point for our customers, as well as improve our margins, as the service provider. However, the flip side of running on commodity hardware is the need to create higher service reliability by investing more on the software side, in terms of hardening the stack towards common hardware and network errors. Only then can the overall system have the level of resilience and recoverability needed to run a highly reliable and available service like SQL Azure.

Here are some of the things we needed to support in software as our *defense in depth* strategy – they would

not be needed when building a modern database system at a different price point on the hardware stack, e.g., RAID5 or RAID10, or SANs. 1. Using checksums to verify data on disk – we learned this lesson the hard way, after observing some bit-flips on disks; 2. Periodically checking (by sampling) data on disk for disk rot, and ensuring that log handshakes are honored; 3. Signing all network communication, to guard against bugs in the network interface card (NIC) firmware – this was actually done after discovering faulty behavior of some NICs; and 4. Storing information (such as identity metadata) redundantly, thus allowing us to reconstruct / rebuild in the case of catastrophic failure. Examples are GPM rebuild and DB identity (server + database name).

To summarize our design philosophy with respect to defense in depth: when building on commodity hardware and network stacks, *trust nothing, verify everything*. Sometimes if you don't trust the stack below you and check everything, then one needs to also measure the cost of each such check. If a check is very expensive at a higher level of the stack then it might be possible to push it down the stack. In these cases, guarantees of reliability need to be negotiated between providers of different levels of the stack, and suitable levels of acceptance testing should be instituted to maintain the trustworthiness of these contracts, or "stack-internal SLAs". A simplification that helps implement the defense-in-depth policy cheaply is to simply take the view that we will *enforce homogeneity* across the cluster. This can be done through a multitude of relatively inexpensive ("hammer") implementation strategies, such as polling, diff-ing, etc., to explicitly check for homogeneity, and interpret that as an indicator of system health.

Build a failure classifier and playbook. Let each failure teach you something for the future. To offer examples, here are some of our most commonly recognizable failure types, and what we do with them:

Planned failures. These are the most common. Examples are application software upgrades, OS patches, firmware upgrades, and planned load balancing events. Handling updates without compromising availability and without breaking any application is a very important design tenet of SQL Azure, and is mediated by the IDS layer (Figure 1). Our *two-phase upgrade* scheme is described in [4]. One must also be careful to throttle upgrade actions, to maintain availability of the overall service. Also, where possible, it helps greatly if one is able to deconstruct a monolithic service or major component into several smaller services, and if these smaller services can be deployed independently of one another. This reduces the strain on the overall system by minimizing the scope of updating pieces of the service in small increments, and provides good opportunities for load balancing.

Handling upgrades well carries paramount importance because if they can be done very successfully and efficiently then we are able to release more often; and if we are able to release more often, then we can improve the service faster. Therefore, quality of upgrades translates directly into continuously improving the quality of the service, as well as adding to it in terms of features and functionality – these translate into a higher quality user experience, as well as better business health for the service provider.

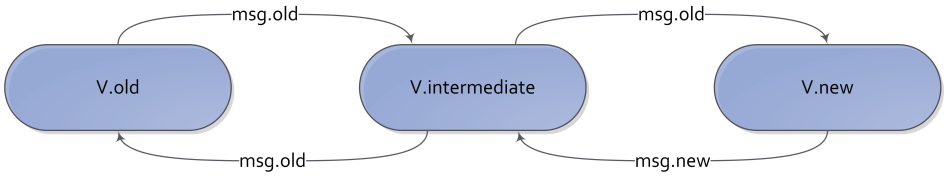


Figure 2: Two-phase rollout strategy

In our view, building towards efficient upgrades, as well as the ability to rollback an update are both equally important. Examples of when we have needed to rollback an upgrade are situations where a rollout resulted in changing network or data protocols in incompatible ways. The standard practice used by SQL Azure is a two phase upgrade, with each of the phases being a fully monitored roll-out. In the first phase the new binaries must be capable of understanding both old and new protocols, but should not initiate the new protocol. This allows the new code to communicate with the old version of code and also allows rolling back safely to the old version. During the second phase, the software can initiate the new protocols because we know the first phase

has completed by the time the second phase starts. If there is a rollback, then it only needs to take one step back to the first phase which is still capable of using the new protocols and data formats. The two phase roll-out pattern (Figure 2) is widely applied in SQL Azure in both database and gateway services, and is a key piece of our fault tolerant system design.

Another thing worth mentioning is that our overall approach to system design is to aim for simplicity, because robustness usually follows. For example, rather than building upgrade support on a feature by feature basis, we chose to build a single upgrade scheme and use this as a "hammer" to solve all issues.

To summarize: *perfect your upgrade scheme; it is the goose that lays golden eggs for the service*.

Unplanned failures. These fall into the following buckets: 1) Hardware and network errors, e.g., top-of-rack switch failures, disk failures, rare failures such as bit flips in NIC, memory, etc. – for these we use software-based defense in depth policy, as outlined in Section 2.4; 2) Software bugs – for these we take a *mitigate first, fix next* policy. For example, if a known combination of statements and inputs are generating a type of failure, we temporarily install a statement firewall, effectively banning the problem combination, until a fix can be rolled out; and 3) Human error – an example of this is an error in configuration management.

In general, we have found handling configuration failures to have been extremely painful, because they may impact multiple machines and workloads, and may quickly spread their contagion before we are able to roll back. Longer running test runs that check for any possible regression on large integration and test clusters is one preventative measure. In a similar vein, having to roll back OS patches has also been painful. Byzantine failures are hard to debug. NIC errors have fallen into this category. Software hangs are comparatively easy to debug, because the SQL org is intimately familiar with the database software and tool chain.

Design everything to scale out. The key theme here is to favor decentralized service architecture, and avoid any choke points. The GPM (Figure 1) is the only centralized piece of the SQL Azure architecture. It is a scaled out single point of failure, with seven redundant copies across seven instances. We have designed and modified this piece extremely carefully, running somewhat exhaustive stress tests before deploying modifications. We also enforce machine roles [4], because that ensures that the service follows a loose coupling paradigm, and that there are effectively firewalls across machines belonging to separate roles, which arrest cascading failures. But one does need a balanced approach towards creating roles, because they tend to eat into cluster size, and eventually, into our business margin.

The lessons we have learned during the two years of deployment are to constantly assess what are the access patterns and network pathways. By doing so, we become aware of hotspots in access patterns, and are able to adapt partitioning strategies to mitigate them and release stress across the system. In practice, this turns out to be a rather effective early warning and failure avoidance strategy, e.g., we observed that overloaded partitions lead to throttling that may ripple across significantly large portions of the cluster, and overloaded VIPs (virtual IP addresses) may lead to connectivity issues. It is best to avoid these types of failures by *constant monitoring and adaptation*.

Protect the service with throttling. SQL Azure is not a VM-based solution; therefore the onus is on the service, and not the user, to provide sensible throttling and resource governance, in a manner that tries to balance supply and demand of scale from the users' perspective, while maintaining high availability and reliability guarantees on behalf of the service.

SQL Server offers a varied set of functionality. When combined with customers' varying workloads, the pattern of stress that may be caused across various parts of the architecture, e.g., connectivity, manageability, memory, IO, disk usage, etc., quickly becomes very complex. However, we take a relatively simple view of how to handle this complexity: *no one request should be able to bring the service (or a significant portion of it) down*. This translates into always ensuring that there will be some spare capacity in the system, and that in turn translates into enforcing thresholds, soft and hard limits, and quotas on almost every shared resource throughout

the system.

Resource governance [8] is an intrinsic feature provided by the SQL Server instance. We leverage this to ensure that the system should be able to control the minimum and maximum resource allocations, and throttle activities that are trending towards exceeding quotas. At the same time, we are able to use the resource governance framework to guarantee that demanding users will get performance guarantees, within what the system can sustain.

3 Challenges Ahead

Much of the work that has gone into designing a self-managing and self-healing system architecture (Section 1), are geared to ensure that the service runs smoothly, and mostly without any human intervention, except under conditions of severe failure. Nevertheless, these measures are mostly to maintain the service infrastructure in a highly available and reliable state 24x7. In order to take the service to the next level in terms of providing ever-improving customer value, and also continuing to improve our own business margins, we need to invest along a number of different dimensions, some of which are outlined in this section.

Execute crisp and granular SLAs that offer more cost/consistency/performance options. Florescu and Kossmann [11] argued that in cloud environments, the main metric that needs to be optimized is the cost as measured in dollars. Therefore, the big challenge of data management applications is no longer on how fast a database workload can be executed or whether a particular throughput can be achieved; instead, the challenge is on how the performance requirements of a particular workload may be met transparently to the user, by the service performing internal load balancing of available capacity. Indeed, one of the fundamental points of Cloud PaaS is to have customers no longer have to worry about physical units such as machines, disk drives, etc. At the same time, it is also important for the service provider to play inside of reasonably attractive price:performance sweet spots.

In addition to optimizing on hardware cost, we are also very motivated to offer more options to our customers, and at a finer level of granularity, in order to appeal to ever more numbers of customers to bring their varied workloads to our platform.

One area of investment would be to formulate crisper contracts (SLAs) with customers and partners, which provide a finer granularity of setting expectations. For example, these advanced and granular SLAs would agree on monitoring mechanisms, expected load characteristics, and types of issues that the customer might be willing to debug, with help from the system. We would also negotiate more detailed SLAs with our internal and external providers, and use those to optimize parts of the architecture, e.g., error checking, as well as streamlining Ops, with a view to optimizing price points of running the service.

In a multi-tenant environment, users need to get the resources they pay for, while the system needs to load balance users across servers. Part of the solution may be defining the parameters of a service level agreement in a way that can be supported with high probability.

It would be interesting to investigate algorithms as in [9], which aim to jointly optimize choice of query execution plans and storage layout, to meet a given SLA at minimum cost. Kraska et al. [10] describe a dynamic consistency strategy, called *Consistency Rationing*, to reduce the consistency requirements when possible (i.e., when the penalty cost is low) and raise them when it matters. The adaptation is driven by a cost model and different strategies that dictate how the system should behave. In particular, they divide the data items into three categories (A, B, C) and treat each category differently depending on the consistency level provided. Algorithms like these easily map on to granular SLAs that offer more fine-tunable options to our customers, with respect to their workload *du-jour*.

Leverage economies of scale; break bottlenecks through strategic hardware investments; optimize database layout vs. cost. An obvious win in the area of leveraging economies of scale, would be to converge on the same hardware SKU as Windows Azure. This should automatically result in more favorable unit prices, and therefore, improve our margins.

An example of moving opportunistically with timely hardware investments to break observed performance and scale bottlenecks is by upgrading to 10 Gb Ethernet connections from rack to rack in a SQL Azure cluster. This helps us break through an IOPS bottleneck we were observing, with intense data traffic across the cluster, e.g., with distributed query workloads. Another possible hardware investment geared towards offering an attractive price:performance tradeoff point for the customer would be to adopt Solid State Disk (SSD) technology. If the price is not right to move every single disk on the cluster to SSD right away, it might still be at the right price point to selectively start the move, e.g., on disks with high traffic, like log disks (multiple virtual databases share a common transaction log in SQL Azure, therefore making log drives a bottleneck).

Generalizing, the first step is to instrument the cluster really well, and monitor it closely with good metrics, to understand where the common bottlenecks lie. Then, as hardware price points evolve, we can latch on to the sweet spots in prioritized fashion, striving to improve the price:performance ratio for the customer, while maintaining comfortable price points of running the service. It would be interesting to investigate query optimizers that adapt storage usage patterns to the relative performance of different volumes, if we were to pursue a hybrid disk strategy in future, e.g., [9]. Adding such adaptivity into the overall architecture would, by itself, generate fresh avenues for cost savings. As in [9], we could formulate the database layout problem as an optimization problem, e.g., by placing smaller on-disk data structures at high QoS levels we can approach a better price:performance point.

React rapidly to maxing out capacity and availability + affordability of new hardware. A successful service must be able to balance the time to deploy a new cluster vs. the declining cost curve of hardware over time. If you buy hardware before there is demand, you waste it. If you buy hardware too late, then your service will not be able to deliver capacity on demand.

A similar set of challenges is brought about when new hardware SKUs become available. To not move fast enough hurts both our customers' value proposition compared to alternatives on the market, as well as hurts our business model. The trick here is to anticipate trends in hardware and price:performance a little bit ahead of the curve, get the certification process in top gear, and roll it out at a point where we are competitive, as well as leverage improving hardware efficiencies to benefit the business.

In order to sustain currently accelerating rates of growth, we need to balance utilization profiles across currently active data centers, as well as gear up to deploy into more data centers with growing demand. To do this efficiently, we need to streamline our processes along multiple dimensions, e.g., acquisitions, hardware and vendor partner chain, gearing internal processes such as testing to move more rapidly on certification, etc.

There is obviously a non-linear cost function involved in deploying a new cluster, just like building a new power-plant is a large CapEx expense for a power company. At times, there may be benefits in incentivizing customers to change their usage pattern so that the service can optimize its own trade-offs in deploying new hardware. This could manifest as a "sale" on an under-utilized data center vs. a full one or perhaps as a rebate for customers that upgrade to a new, resource-conserving feature.

Bake TCO into our engineering system. It has been one of our fundamental principles of system design, that packing more databases per machine reduces our TCO. Moving forward, we need to bake TCO into our software design process, so that it becomes one of the fundamental pillars of system architecture. Here are some examples of how this might be achieved, by asking the following questions: 1) Is the design of the new feature significantly reducing the percentage of total disk capacity that is monetizable, e.g., by reducing the effective cluster-wide capacity to carry non-redundant customer data? 2) Is the design of the new feature significantly reducing the total CPU horsepower of the cluster? 3) Is the design channeling away IOPS capacity in a way that

slows down monetizable customer queries, with respect to negotiated SLA? 4) Are the Resource Management policies and protocols in good alignment with the pricing model?

The cost vs. value of new feature/functionality needs to be considered for each resource dimension. Once considered, the feature may need to be designed differently to minimize large, non-monetizable elements. When there are large costs, giving end users control and visibility over that cost will help maximize the use of shared resources.

System help on performance tuning and Management. As we approach a point where the underlying infrastructure just works, then the main point of failure will shift closer to the customer, i.e., be mostly attributable to a combination of workload, functionality, scale, access patterns, etc. If we can improve the level of self-help that the system can provide to the customer, then in many cases it will become more economical for both the customer, as well as for us as service providers, if the customers are able to undertake recommended remedial actions of their own accord, with existing help and hints from the system.

Pre-cloud era research focused on auto-tuning mechanisms such as automatic index or statistics creation to improve the overall quality of a given plan. These core issues do not change in a simple hosted environment or a platform database service. Various techniques and tools have been created to tune workloads or assist in physical database design, and these same techniques can often be used on a database platform (perhaps with small modifications).

Costs influence performance tuning in cloud environments, automatic or otherwise. For the individual purchasing computing resources, there are incentives now to reduce the recurring cost. It may become very important not just to tune logical and physical database design but to do so against a cost function with multiple price points. For example, properly indexing an application and validating that each OLTP query does seeks instead of scans will be easily measurable in the service bill. Performance evaluation will need to understand the size of the database workspace in addition to traditional core database concepts. If a business decides to increase the computing capacity for a busy season, this will influence performance evaluation and optimization techniques in different ways than on fixed hardware.

From the perspective of the tenant hosting the service, there will be incentives to reduce the cost of delivering the service. This may force re-evaluation of computationally expensive items such as query compilation to both reduce time spent generating plans as well as to find plans that perform well in a multi-tenant hosted environment. This will bias some engineering choices towards reducing internal costs since buying new computing capacity is expensive. Features including physical plan persistence, automatic physical database design tuning, and automatic identification of suboptimal queries now become ways to reduce service delivery costs.

Current platform database systems largely avoid traditional large database-tuning problems by focusing on small query workloads which are easy to tune manually and because database sizes are currently more limited than non-hosted counterparts. As the cloud hosting platforms mature, these differences should largely disappear and the same core challenges will remain.

Performance tuning will also become somewhat more complicated due to multi-tenancy. If hardware is shared by many customers, isolating performance issues may become difficult if they are not isolated from each other. Cloud database platforms will need to invest in mechanisms to isolate problem applications both to troubleshoot them and to manage their service to prevent one database application from negatively impacting others if there is a problem.

4 Conclusions

In this paper we have reflected on the lessons we have learned while ramping up the SQL Azure database service these past two years. The main themes emerging from past lessons we have learned have been to design everything for scale-out, carefully instrument and study the service in deployment and under load, catalog

failures, take a calibrated approach towards responding to them, and to design both the architecture, as well as failure response protocols and processes to contain the scope of failures and mitigate their effects effectively, rather than focusing too heavily on failure avoidance.

With regards to ongoing challenges and investments, the focus is on moving towards a higher level of sophistication in empowering the user of the service to get help from the system to figure their way out of localized failures. We would also like to move towards more granular and load-tunable SLAs. As service providers, it is important for us to move with greater agility to combine hardware trends and algorithmic innovations in areas like optimized database layout and query plans, to approach price:performance sweet spots. In response to accelerating adoption of SQL Azure, it will become important to deploy new data centers rapidly, and to continuously improve TCO, efficiency and utilization, in order to improve the business model of running the service. Finally, we have work to do in terms of improving our tools, both to attract and migrate large and varied customers with mission-critical data and computations to the SQL Azure service.

References

[1] The Cloud: Battle of the Tech Titans, Bloomberg Businessweek, March 3, 2011 (http://www.businessweek.com/magazine/content/11_11/b4219052599182.htm).

[2] Weinman, J., Cloud Economics and the Customer Experience, InformationWeek, March 24, 2011 (http://www.informationweek.com/news/cloud-computing/infrastructure/229400200?pgno=1).

[3] Campbell, D. G., Kakivaya, G., and Ellis, N., Extreme scale with full SQL language support in Microsoft SQL Azure, In Proc. 2010 International Conference on Management of Data, pp. 1021-1024, 2010.

[4] Bernstein, P.A., Cseri, I., Dani, N., Ellis, N., Kalhan, A., Kakivaya, G., Lomet, D.B., Manne, R., Novik, L., Talius, T., Adapting Microsoft SQL Server for Cloud Computing, ICDE 2011, pp. 1255-1263.

[5] Sakr, S., Liu, A., Batista, D.M., Alomari, M., A Survey of Large Scale Data Management Approaches in Cloud Environments, IEEE Communications, 2011.

[6] Microsoft Corp.: Microsoft Exchange Hosted Archive. http://www.microsoft.com/online/exchange-hostedservices.mspx

[7] Microsoft Corp.: Microsoft SQL Azure. http://www.microsoft.com/windowsazure/sqlazure/

[8] Microsoft Corp.: Introducing Resource Governor (http://technet.microsoft.com/en-us/library/bb895232.aspx)

[9] Reiss, F.R., Satisfying database service level agreements while minimizing cost through storage QoS, in SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing.

[10] Kraska, T., Hentschel, M., Alonso, G., Kossmann, D., Consistency Rationing in the cloud: Pay only when it matters, PVLDB, 2(1):253-264, 2009.

[11] Florescu, D., Kossmann, D., Rethinking cost and performance of database systems, SIGMOD Record, 38(1):43-48, 2009.

[12] Thakar, A., Szalay, S., Migrating a large science database to the cloud, HPDC 2010, 430-434.

[13] Microsoft Corp.: Microsoft Case Studies. EHA. http://www.microsoft.com/casestudies/Case_Study_Detail.aspx?casestudyid=4000003098.

[14] Kossmann, D., Kraska, T., Loesing, S., An evaluation of alternative architectures for transaction processing in the cloud, In SIGMOD, 2010.

[15] Microsoft Corp.: Microsoft Case Studies. TicketDirect. http://www.microsoft.com/casestudies/Case_Study_Detail.aspx?CaseStudyID=4000005890.

ICDE2012

Washington DC, USA

April 1-5, 2012

http://www.icde12.org/

Sponsored by

IEEE

IEEE computer society

The 28th International Conference on Data Engineering

The annual International Conference on Data Engineering (ICDE) addresses research issues in designing, building, managing, and evaluating advanced data-intensive systems and applications. It is a leading forum for researchers, practitioners, developers, and users to explore cutting-edge ideas and to exchange techniques, tools, and experiences.

We invite the submission of original research contributions and industrial papers as well as proposals for workshops, panels, tutorials and demonstrations.

IMPORTANT DATES

Research papers

Abstracts Due	JULY 12, 2011
Full Papers Due	JULY 19, 2011
Author Feedback	SEPTEMBER 8-13, 2011
Author Notification	OCTOBER 11, 2011
Final Versions Due	NOVEMBER 8, 2011
Conference Dates	ARPIL 1-5, 2012

Workshop Proposals due: **June 15, 2011**

Seminar/Tutorial Proposals due: **July 20, 2011**

Demo Proposals due: **July 26, 2011**

Panel Proposals due: **July 26, 2011**

Venue

Renaissance Arlington Capital View Hotel

2800 South Potomac Ave, Arlington, Virginia 22202 USA

GENERAL CHAIRS
X. Sean Wang (US NSF, Univ. of Vermont)
Nabil R. Adam (US DHS S&T, Rutgers Univ.)

GENERAL VICE CHAIRS
Alex Brodsky (George Mason Univ.)
Vijay Atluri (Rutgers University)

PROGRAM CHAIRS
Johannes Gehrke (Cornell University)
Beng Chin Ooi (National U of Singapore)
Evaggelia Pitoura (U of Ioannina, Greece)

INDUSTRIAL PROGRAM CHAIRS
Nicolas Bruno (Microsoft Research)
Liang-Jie Zhang (IBM Research)
Kenneth Ross (Columbia University)

SEMINAR/TUTORIAL CHAIR
Aryya Gangopadhyay (UMBC)

WORKSHOP CHAIRS
Sharad Mehrotra (Univ. of California, Irvine)
Anupam Joshi (UMBC)

PANEL CHAIRS
Alex Tuzhilin (New York University)
Michael Gertz (University of Heidelberg)

POSTER CHAIRS
Jaideep Vaidya (Rutgers University)
Zachary Ives (Univ. of Pennsylvania)

DEMO CHAIRS
Christof Bornhoevd (SAP Research)
Richard Goodwin (IBM)
Mirek Riedewald (Northeastern University)

PROCEEDINGS CHAIRS
Anastasios Kementsietsidis (IBM)
Marcos Vaz Salles (University of Copenhagen)

LOCAL ORGANIZATION CHAIRS & SPONSORSHIP CHAIRS
Carlotta Domeniconi (George Mason Univ.)
Huzefa Rangwala (George Mason University)

FINANCE CHAIR
Hui Xiong (Rutgers University)

PUBLICITY CHAIR
Soon Ae Chun (City Univ. of New York)

WEB CHAIR
Micah Sherr (Georgetown University.)

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398