

An Integrated Approach to Performance Monitoring for Autonomous Tuning

Alexander Thiem

Ingres Germany GmbH

alexander.thiem@ingres.com

Kai-Uwe Sattler

Ilmenau University of Technology, Germany

kus@tu-ilmenau.de

Abstract—With an ever growing complexity and data volume, the administration of today’s relational database management systems has become one of the most important cost factors in their operation. Dynamic workloads and shifting demands require continuous effort from the DBA to deliver adequate performance. The goal of a modern DBMS must be to support the DBA’s work with automated processes and workflows that facilitate quick and precise decisions.

In this paper, we present the concept of an integrated performance monitoring in the Ingres DBMS that provides long-term collection of information valuable for performance tuning, problem identification and prediction. The approach of enhancing the DBMS core with monitoring features rather than adding an additional watchdog on top of the system leads to a high data resolution while still having only a minimal overhead.

This concept was successfully prototyped in Ingres with a very small overhead for most usage scenarios. The prototype is able to collect and analyze data and to give useful recommendations on the physical database design to improve overall performance of the DBMS.

I. INTRODUCTION

The complexity of many of today’s DBMS setups, often with auto-generated database schemes containing dozens or even hundreds of tables, makes it nearly impossible to maintain a system without a certain degree of automation. With abstraction layers such as Hibernate or application frameworks like Ruby on Rails that create their database schemes automatically, the application developer can lose sight of the database and DBMS and may no longer think about an optimal physical database design. Most of the commonly used commercial DBMS products today provide tools and features either to automate processes or parts of processes to support DBAs in their work.

In this paper, we address the problem of collecting relevant information in the DBMS to provide a basis for well-founded decisions about changes to the system’s configuration. While most approaches to autonomous tuning are based on ad-hoc or short-term data or have a very low temporal resolution of their data, our concept aims at a continuous monitoring of the system over a long period of time that allows us to detect and even predict problems without stressing the DBMS. The evaluation of our system will show that we achieve a high data resolution with minimal overhead in most usage scenarios. The collected data can serve as the input for analysis modules that can range from simple reporting, the recommendation of changes and up to a fully automated self-tuning. We implemented additional mechanisms in the DBMS

to provide feedback on hypothetical changes to be able to discern their benefits. The platform we have chosen for this implementation is the Ingres DBMS that helps us with some of its unique features and which – as an open source project – makes it easy to add new features that can be made available to the public.

The Ingres DBMS originated at the University of California, Berkeley where Michael Stonebraker and Eugene Wong started a research project to implement a relational database management system based on Edgar Codd’s work ([1], [2]). Their first prototype was completed in 1974 and at that time the source code was available on tape so that Ingres could spread through the academic world. With the foundation of Relational Technology, Inc. in 1980, which was later renamed to Ingres Corporation, Stonebraker commercialized Ingres and started selling his DBMS. First acquired by ASK Corporation in 1990 and then by Computer Associates in 1994, Ingres’ active development was slowed down for years until Ingres Corporation re-emerged as an independent company in 2005. Since version 2006, Ingres has been released under the terms of the GNU General Public Licence Version 2. A detailed look at Ingres’ history can be found in [3].

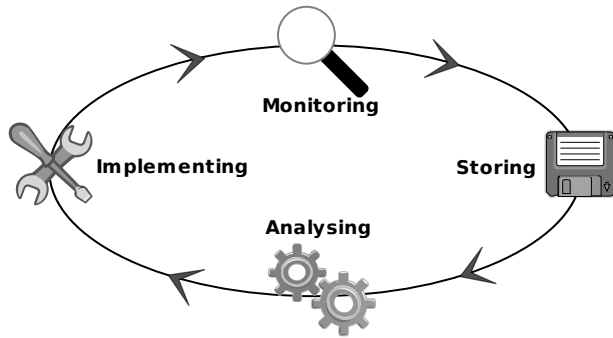
The rest of this work is structured as follows: Section II gives an introduction to the fundamentals of autonomous database tuning. Section III presents research work related to our project. In section IV, the architecture of the proposed concept is discussed followed by an evaluation of its performance in section V. Section VI then gives an outlook on possible enhancements in future versions of the tool.

II. FUNDAMENTALS

DBMS tuning can be described as a control loop: The DBA watches performance indicators such as response time, disk and CPU usage and can then choose from a number of control variables that may help to improve the current situation. Those can be indexes, statistics, DBMS settings amongst others. The DBA then implements the changes, monitors the DBMS’ reaction and plans the next round of changes.

Figure 1 shows the steps of an auto-tuning control loop: The monitoring of the system, the storage of the monitored data over a period of time, the analysis of this data and the presentation or automatic implementation of changes based on the analysis.

The performance indicators used in auto-tuning can be categorized into: workload information, catalog information



1: Auto-Tuning Control Loop

and system statistics.

Information about executed statements reveals the usage pattern of the system which is most important for making useful design decisions. Every statement is recorded together with the time it took to execute and other variables such as which secondary indexes were used. Such a statement history shows which tables, attributes and indexes are frequently used or which statements are the most expensive ones and therefore allows the configuration changes to be matched to the actual workload of the system.

The system catalogs store definitions of all tables, attributes, users, etc together with information about storage structures, histograms, indexes and much more. This information is needed to give accurate recommendations for physical design changes such as changing a tables storage type or the collection of data statistics.

System-wide statistics contain all kinds of data points including the number of database connections, locks and cache usage, etc. Recording those values continuously over a longer period of time allows the identification of patterns and peaks, possible bottlenecks and, to a certain degree, the prediction of future problems. This can help to find the optimal set of system settings.

III. RELATED WORK

In recent years, several approaches have been developed to address autonomous tuning tasks. The most prominent approaches are devoted to the index selection problem [4], to the problem of finding an optimal set of materialized views [5], and the optimal fragmentation strategy [6]. In all of these solutions, the query optimizer is invoked to estimate the benefit of hypothetical objects (indexes, partitions etc.) for a given workload.

More recent approaches consider the tuning tasks as an online optimization problem where feedback from the query optimization and/or execution is used to alert the DBA if a better configuration exists [7] or to adjust the current configuration automatically to the workload [8], [9], [10].

However, all these approaches focus more or less on a single tuning problem and collect the necessary data explicitly and on-demand from the query optimizer. In contrast, only a few approaches described in the literature follow a more

comprehensive monitoring strategy. Oracle 11g [11] provides an Automatic Workload Repository (AWR), a repository for collecting performance-related data. This data is collected from in-memory statistics and written to a snapshot in the AWR. The list of data contains base statistics such as the number of physical reads and writes, times spent in specific processes, e.g. parse times, wait events as well as derived metrics, i.e., per query/transaction statistics, top (problematic) objects and queries. Snapshots are captured periodically allowing to compare statistics from different times. Furthermore, the snapshots are used by an advisory framework to derive recommendations for the physical design, SQL tuning, and memory management parameters as well as by the Automatic Database Diagnostic Monitor (ADDM) for identifying performance problems in the DBMS operation.

IBM DB2 provides a similar approach called DB2 Performance Expert [12] which consists of a DB2 monitoring component, a performance warehouse, and exception processing.

Both systems are general purpose solutions for integrated system management and are, therefore, rather heavy-weight. In contrast, the goal of our work presented here is a lightweight, optimizer-integrated approach addressing the needs of autonomous tuning.

Another approach related to our work is SQLCM [13], a framework consisting of a monitoring component and an ECA rule engine. SQLCM follows also a server-centric solution for low overhead on server execution. In SQLCM, several classes of objects are monitored, such as queries, transactions, blockers (queries blocking other queries due to incompatible locks), as well as timer objects. The attributes of these objects (start time, duration, costs) are collected in so-called lightweight aggregation tables (LAT) which are in-memory structures for aggregating the monitored data grouped on the objects. The LATs are used to feed the ECA rule engine for triggering specified actions. Contrary to this, in our work the monitoring component is used to collect long-term statistics which can be analyzed by external tools.

IV. CONCEPT

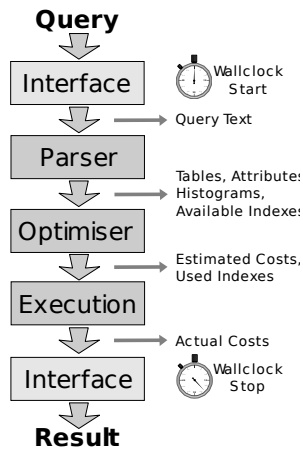
Our approach was designed with the following set of requirements: i) Data monitoring and collection must be lightweight so that the DBMS performance is not affected and the overall performance does not suffer. ii) For all cost based decisions the internal cost model of the DBMS should be used to ensure that design changes will actually be used by the DBMS. iii) All parts of the system should be easily expandable to allow for future enhancements.

Following the concept of the control loop, the system is split into the four phases of monitoring, data storage, analysis and implementation.

A. Monitoring

The goal of the monitoring phase is to collect as much data as possible during the day-to-day usage of the DBMS with little or no impact on the system's load, response time and performance. As shown in figure 2, the DBMS core is used

to collect data from the three categories of workload, catalog and statistics information by adding local sensors along the path a statement takes through the DBMS.



2: Data Monitoring

While most other systems are time- or event-triggered and often use a watchdog sitting on top of the DBMS, monitoring its activities, our approach is fully integrated into the DBMS core to avoid any kind of unnecessary overhead during the collection of data. Everything that is logged is known to the DBMS anyway – for example, when a statement text is parsed, the table and attribute names need to be read from the system catalogs. This data is logged right at its source so that no further access to the catalogs (and a possible disk access) is required for the monitoring. The time to execute, estimated costs, indexes used in the execution are logged by adding simple function calls at various places in the core rather than running an extra monitoring thread that would cause not only communication overhead but may also need to handle more complex data structures instead of logging only very simple data. Therefore, our system does not call the DBMS modules such as the optimizer or parser but it is part of each of those modules.

The data that is collected in the DBMS core is stored in main memory and is made available over the Ingres Management Architecture (IMA). IMA is a flexible framework included in Ingres that offers an extensible relational interface to read and write internal DBMS data. With IMA it is possible to easily access in-memory structures within the DBMS over standard SQL which allows remote monitoring of the DBMS without having to implement a new interface or communications protocol. Because IMA objects reside only in main memory, there is no disk access required to store or read the data. Each class of IMA objects can be registered as a virtual table in an Ingres database which then offers the data over any supported SQL interface.

To limit the overall memory requirements for the monitoring, all data structures were implemented as ring buffers that contain a moving window of data with a configurable size. By default, the monitoring can capture up to 1000 different

statements until the buffer wraps around.

Figure 3 shows the database schema of virtual tables used to populate the monitored data over IMA. The *statements* table holds all unique statements that were executed on the DBMS together with a hash of the statement text that is used as the referencing key to the other tables. The *workload* table keeps the history of when which statement was executed and what costs were involved to store the actual workload of the system. The *references* table keeps track of which database objects (tables, attributes, indexes) were used in which statement. System-wide statistics are stored in the *statistics* table. Recording these values over a long period of time allows trend analysis of the system's state.

Statements

(Database name,
Unique hash key,
Query text,
Frequency)

Workload

(Database name,
Hash key → Statements,
Optimiser CPU time,
Optimiser disk I/O,
Execution CPU time,
Execution disk I/O,
Estimated CPU time,
Estimated disk I/O,
Wallclock to execute)

Tables

(Database name,
Table ID,
Table name,
Frequency,
Storage structure,
Data pages,
Overflow pages)

Statistics

(Current sessions,
Maximum sessions,
Locks per transaction,
...)

Indexes

(Database name,
Index ID,
Index name,
Table ID → Tables,
Attribute ID → Attributes,
Frequency)

References

(Database name,
Hash key → Statements,
Object type,
Object ID,
Table ID → Tables)

Attributes

(Database name,
Attribute ID,
Table ID → Tables,
Attribute name,
Frequency,
Histogram)

3: IMA Table Schemes

B. Data Storage

Data storage is performed by a lightweight daemon running in the background. The tool periodically wakes up and queries the IMA database to get the newest data. Because of the volatility of the data in IMA, the time between wake-ups must be short enough to capture changes in a detailed enough resolution – but in order to reduce overhead on the DBMS the period must not be too short. Collecting up to 1000 statements within an interval of 30 seconds has proven to be enough to get a proper data resolution without significantly loading the DBMS. If needed, the resolution can be increased to capture more statements per second.

The daemon polls IMA a given number of times to get the most current data and then appends the collected data to the workload database. The workload database is a native Ingres database that contains the same table schema as the one used in IMA. Updates on tables are appended and provided with a timestamp to allow trend analysis over a longer timespan. This delayed way of persistently storing the data has only a slight impact on the DBMS load because disk accesses are performed only every few minutes instead of with every executed statement as it would happen when the collected data is directly stored on disk.

While we implemented the data storage in this prototype in an external tool, this task can also be integrated into the DBMS core to avoid polling IMA and instead writing to workload DB only when the main memory buffers are full. We believe this can further reduce the overhead seen in the evaluation in section V.

As with the IMA tables, the amount of data in the workload DB needs to be limited as the tables would infinitely grow. By default, all entries are kept for seven days to allow recording the workload of a typical work week.

Because the workload DB is in fact a user database, handling the collected data is most simple and can be done with standard SQL. This also allows to create triggers and procedures that automatically react to changes such as exceeding or dropping below a given threshold for a data point. The daemon provides an active alerting mechanism that informs the DBA in case of a defined database event such as reaching the maximum number of users on the system. The DBA can easily set up his own alerts by creating more triggers. While [13] states that using triggers and procedures is too slow to perform good enough for an online monitoring, we believe that in our situation, where the overhead of executing a trigger is only seen every few minutes when the daemon appends new data, the benefit of using well-known standard techniques is the preferable solution.

C. Analysis

There are three basic levels of how to deal with the collected data. The first is a simple report: The data gets aggregated and visualized and it is up to the DBA to identify problems and to find solutions for them. In the next level, the data will be analyzed based on pre-defined rules. A set of known facts is compared with the data to identify the problems which are then mapped to possible solutions. This can happen either locally, where each data point is examined independently, or globally, where all of the information is seen as a whole. This helps to reveal side effects and to avoid conflicting configuration changes. The third level of analyzing the data is to interpret its meaning, to identify trends and patterns and to start predicting potential problems in advance. This more sophisticated approach aims to replace or at least to minimize the human factor in the analysis of the DBMS.

In our implementation we concentrate on data collection and storage, however, we have created a working analyzer tool that is able to scan the collected data and to recommend changes to

the current database configuration. The result of this analyzer is a mixture of plain reports and rules-based recommendations such as:

- Actual and estimated costs of a statement differ significantly: This may be caused by missing or outdated statistics such that the optimizer may not be able to find the cheapest plan.
- One or more attributes of a table have no statistics: Histograms should be created.
- A table with a fixed amount of main data pages has already more than 10% overflow pages: The table should be restructured or modified to storage structure B-Tree.

In addition to these simple rules, the analyzer also recommends the creation of new indexes. In Ingres, secondary indexes are stored as tables that have columns containing the indexed keys and a pointer to the data page. Therefore, indexes can be used by the optimizer by simply adding them to the list of joining tables. This fact allows us to feed the Ingres optimizer with a number of hypothetical, or virtual indexes, as introduced by [14], exploiting its decision about which indexes will actually be used to find an optimal index set for the workload.

Our analyzer tool is only meant to show a possible way of analyzing the data we collect from the DBMS. It does not yet use all potential information such as the interdependencies between local sensors or a sophisticated trend analysis of time series of data. However, the architecture of the system with its standard SQL interface makes it easy to define more rules or to create an alternative analyzer.

D. Implementation

The implementation of the recommended changes either happens automatically or manually by the DBA. While the DBA can easily schedule the changes for a maintenance window or decide to perform only some changes now and some others later, an automatic implementation requires efficient ways of, for example, creating a secondary index during normal operation of the DBMS without blocking other users. [15] describes the possibility of exploiting full table scans to create indexes on the fly while executing a statement and [16] breaks with traditional indexing and aims to crack a database into self-organizing pieces.

For our system, we restricted ourselves to a manual implementation of changes. Results and recommendations of the analyzer are presented in textual and graphical form to support the DBA in his decision as to which changes need to be performed.

V. EVALUATION

This section presents a set of experiments that were performed, first, to see the impact of the monitoring on the DBMS and second, to show the usefulness of the collected data for the workload performance. The machine that was tested on was a Gentoo Linux 32-Bit desktop system with a two-core Intel Pentium D processor with 3 GHz clock frequency, 4 GB of main memory and a 150 GB hard drive. The system was

tested against a database populated with data from the *Non-Redundant Reference Protein* (NREF) database as described in [17]. This database consists of six tables filled with a total of 100 millions of rows of real, non-synthetic data. As plain text files the NREF database takes about 6.5 gigabytes. In the DBMS, depending on the storage structure, it takes up to 30 gigabytes or more, ensuring that the database is significantly larger than the system's main memory.

A. Monitoring Tests

The monitoring tests show the impact on the overall system performance caused by the overhead of the monitoring code in the DBMS core and the daemon that writes the data to the disk. We expect that for a workload of expensive statements that take several seconds or minutes to complete, the impact of the monitoring is negligible because the additional function calls in the core are all of subsecond runtime and are only executed once per statement. The impact of the daemon would be apparent because it wakes up periodically to read data from IMA and write it to the workload DB. This adds additional disk accesses during the execution of statements. With a higher throughput of statements, a growing impact on performance is to be expected. For statements that execute in a fraction of a second the overhead of the monitoring code becomes substantial. The overhead of the daemon increases with the number of rows that need to be written to the workload DB.

Three test setups were used to show the influence of the monitoring: i) One Ingres instance compiled from the latest available source code without any of the changes described in this paper. All default settings were kept. The NREF database was created and filled with data using only primary keys and no other indexes. ii) A second Ingres instance with the monitoring code compiled in. The same DBMS configuration and NREF database are used. iii) The monitoring Ingres instance but with the daemon running in the background. These variations show the influence of the monitoring code in the core alone and of the daemon running on the same machine. On each of these setups three different tests were performed: First, a mixture of the NREF2J and NREF3J query sets with 50 selects¹ that stress the database with expensive joins and many full table scans. Second, a set of 50,000 simple joins of two tables was executed. All statements were in the form of

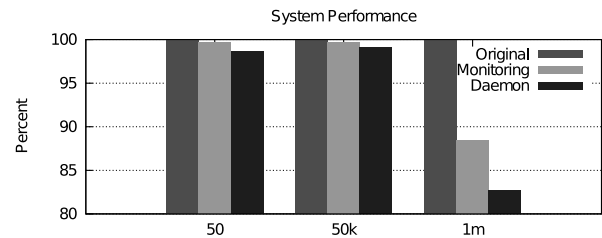
```
select p.nref_id, sequence, ordinal
from protein p join organism o
on p.nref_id = o.nref_id
where p.nref_id = 'NF00000001'
```

with the where clause cycling through 50,000 different nref_ids, forcing the monitor to log each statement as a new one. The third test was performed with 1,000,000 even simpler statements in the form of

```
select p.nref_id
from protein p
```

¹The query set and other necessary files to reproduce the tests can be found at <http://www.thiem-net.de/ida/>

where p.nref_id = 'NF00000001' to see how the monitor lowers the pure throughput of query processing. All tests were repeated three times to minimize local anomalies.



4: System Performance

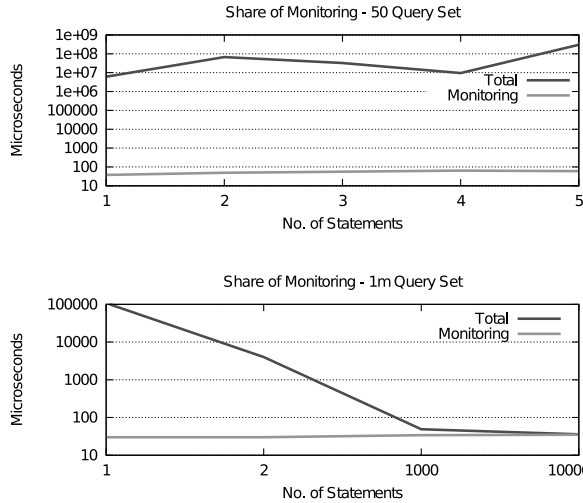
Figure 4 shows the results of the three tests on the three setups. The setups are called *Original*, *Monitoring* and *Daemon* referring to the three Ingres instances described above. The tests are called *50*, *50k* and *1m* referring to the number of statements executed. Results are shown as relative values to the time the test needed to execute on the untouched Ingres instance. The graph starts at 80% to highlight the differences.

The tests show that the monitoring code in the DBMS core has very little influence when executing complex and expensive statements. The DBMS needs several seconds or even minutes to execute one of the 50 NREF queries so that the overhead of the monitoring code becomes negligible and stays far below 1%. A slightly greater impact can be seen with the daemon which is writing system statistics to the disk during the query execution. However, the difference is still only at about 1%.

The *50k* test with 50,000 simple joins shows an even smaller difference between the three setups than the *50* test. The 50 complex queries put a very high load on the disk because many full table scans are required. This is not the case for these simple queries and the daemon doesn't slow down the execution as much as before, probably by forcing the read/write head of the hard disk to move between two different locations all the time. Still, it can be seen that the DBMS without the daemon has a slightly higher throughput of statements.

The *1m* test reveals the impact of the monitoring. The one million very simple statements need about 11% more time to complete with the monitoring code and 17% more when the daemon is running. It can be seen that the main part of the overhead here comes with the monitoring code in the core as this is added for every single statement while the daemon still only wakes up every 30 seconds and writes to disk every few minutes. With over 1,000 statements per second, the default data resolution of the monitoring of 33 statements per second has been exceeded by far so that the daemon always writes the same amount of rows per interval, no matter how high the throughput in the DBMS is.

To understand this drop in performance, the time needed for the monitoring was measured and compared to the overall



5: Share of Monitoring

time a statement needs to execute. The first graph in figure 5 shows the share of the monitoring for the first five queries of the 50 test. It can be seen that the time spent in the monitoring code is negligible compared to the overall execution time. This situation continues for all of the 50 statements because each query consists of expensive full table scans, joins and sorts.

The second graph shows the share observed in the 1m test where all the very simple statements only differ in the where clause. For the very first statement, the DBMS needs to initialize its caches and needs to read catalog information from the disk. Here, the monitoring takes only a fraction of the overall execution time. The second statement already shows the impact of caching in the DBMS. While the monitoring remains constant, the overall time to execute drops down to only 5% of what was needed for the first statement. With the 1000th statement, the share of the monitoring takes 90% of the overall execution time and at 100,000 statements, the monitoring is at nearly 98%.

This shows that, while the DBMS is able to significantly reduce the time to execute many similar and very simple statements, our monitoring always takes more or less the same amount of time, keeping the lower boundary of execution time up. The measurement revealed that each call to a monitoring function takes about one or two microseconds. Depending on the complexity of the query (number of referenced tables, attributes), this added between 30 and 70 microseconds per statement in our tests, while the 1m statements alone took less than 30 microseconds to execute. We believe that by adding a better caching strategy to the monitoring code, we are able to further reduce this overhead and the number of function calls so that the monitoring scales better when dealing with most simple queries such as the ones in the 1m test.

Another performance factor is of course the access to disk when writing the data to the workload DB. At its highest throughput of logging 33 statements per second (which can

be configured to a higher or smaller number as needed) the workload DB grows at a rate of about 28 megabytes per hour. This data is kept for seven days by default, so that the size of the workload DB is limited in total to about 4.7 gigabytes. While we did not include this in our experiments, we believe that moving the workload DB to another physical disk will reduce the overhead added by the daemon.

B. Analyzer Tests

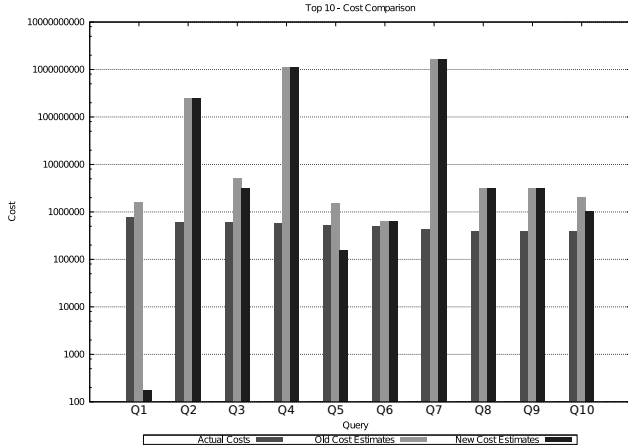
While this paper focuses on the monitoring of data, we also have implemented a basic analyzer tool that scans the collected data and recommends changes to the physical database design. The experiment in this section shows whether the recommendations of our analyzer are actually useful to improve the performance of a system. To test the usefulness of the analyzer the two Ingres instances from the previous section were used. The original instance with no code changes applied was manually optimized. A set of 33 reference indexes recommended by [17] was applied, statistics on all tables were collected and all tables were modified to storage structure B-Tree.

The second instance, with the monitoring code, was used to record the workload of the 50 NREF statements. The analyzer was executed to scan the workload and to recommend changes. Those changes were applied to compare their performance win with the one of the manual optimization. We expect that the performance of both systems will be similar but, due to the fact that the analyzer tries to find an optimal index set, the size of the resulting database should be smaller.

The analysis took about 40 seconds to scan the workload of the 50 NREF statements and to test possible new indexes on the DBMS. For 31 statements the analyzer reported that the estimated cost values differ significantly from the actual execution cost and suggested to collect statistics. In addition, all of the six tables had a high number of overflow pages on disk because of the default storage structure heap and the analyzer recommended modifying them to B-Tree. For the whole workload, the analyzer recommended 12 secondary indexes.

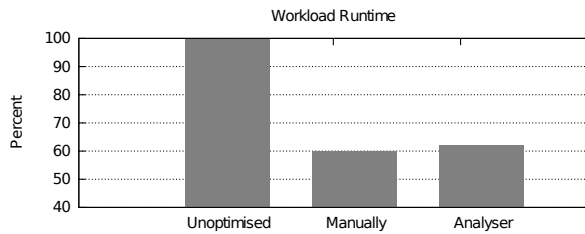
Figure 6 shows the resulting cost diagram of the ten most expensive statements of this workload. The first bar represents the actual costs needed to execute the statement, the second bar shows the costs the optimizer estimated and the third bar shows the estimated costs based on the creation of new, yet virtual, indexes. To draw this graph, our analyzer currently only considers performance win through index creation and doesn't factor in storage structures or statistics, which is why only a few statements seem to benefit from the recommended changes. The results of the experiment will show that the performance win is much higher. However, the graph also shows that the cost estimations of, for example, statements Q2, Q4 and Q7 significantly differ from their actual costs which indicates a poor query execution plan – here, the analyzer recommended the collection of statistics.

To test the usefulness of the analyzer all the recommended changes were implemented: All tables were modified to B-



6: Cost Diagram

Tree, the 12 recommended indexes were created and statistics were collected.

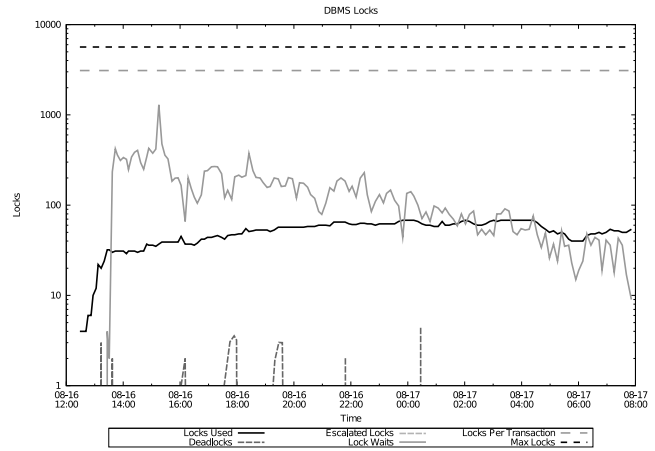


7: Analyser Results

Figure 7 shows the results observed in this experiment. The manual optimization of the NREF database grew the size of the data files from 33 to 65 gigabytes on disk while the time to execute the query set was reduced to about 60% of the original runtime. The recommended changes of the analyzer grew the size to 53 gigabytes, saving 12 gigabytes over the manual optimization with 33 indexes. The execution time was cut down to 62% (without taking the overhead of the monitoring into account).

Our analyzer mainly concentrates on the level of reporting of aggregated data and the identification of common causes for performance problems. It makes presumptions such as that an index that was recommended for many statements is more useful than an index that has less recommendations. This – as with some of the other rules – is only true for certain situations. But still, the presented results can already help to identify bottlenecks and to solve performance problems that otherwise could have caused time-consuming work for the DBA. While the creation of statistics and the modification of table structures to B-Tree can be considered to be standard tuning tasks, the identification of an optimal index set for a given workload requires in-depth knowledge about the system. The DBA easily risks creating too many indexes that could even slow down the system. The experiment shows that the analyzer was able to

recommend useful changes to the physical database design with a performance win that was comparable to the manual optimization. The recommended index set was only half as big as the reference index set, saving disk space of over ten gigabytes.



8: Locks Diagram

In addition to textual recommendations, our analyzer also provides graphical feedback of the system’s state. Figure 8 shows an example of how the analyzer visualizes DBMS statistics such as from the locking system. The graph shows the number of used locks together with indicators for lock waits and deadlocks to help the DBA identifying problems.

VI. CONCLUSIONS

Studies such as [18] state that today the smallest fraction of overall costs to operate an IT system are costs for software and hardware and that by far the biggest fraction is the cost of personnel using and maintaining the system. Therefore, a selling point of growing importance is the ability of software to keep the maintenance effort at a minimum. Software systems should thus become more and more self-sustaining and independent of human intervention.

In this paper, we presented a concept for an integrated performance monitoring for autonomous tuning in the relational database management system Ingres that had not yet all of the features of monitoring, collecting and analyzing data to improve the performance of the system. The DBMS core was expanded to monitor the workload on the system. The continuous data collection is performed by a daemon that reads the data from the DBMS and stores it in a workload database. The collected data is then scanned to find possible performance improvements and to recommend changes to the physical database design.

In contrast to most existing auto-tuning systems, our approach is based on a tight integration with the DBMS core to keep the overhead at a minimum while still providing a high data resolution.

With the experiments in this paper, we were able to show that the overhead added by the monitoring is negligible for

complex statements that take several seconds or more to execute. The impact becomes apparent for subsecond statements where – at over 1000 statements per second – the throughput was lowered by about 17% for our first experimental implementation. We believe that, with a more sophisticated cache strategy built into the monitoring, this overhead can be further reduced to serve well enough for high-performance systems.

To test the usefulness of the collected data, we created a basic analyzer tool, that was executed on a workload to see if the recommended changes could actually improve the performance of the system. The experiment showed that the analyzer results in a performance win that is comparable to a manual optimization while requiring less disk space and saving the time a DBA would have needed to tune the system.

The presented implementation is still experimental and should be considered as a proof-of-concept rather than ready for production use. Besides the extension of the monitoring to collect even more useful data from the DBMS we mentioned a set of other possible improvements throughout this paper, such as the integration of the data collection into the DBMS core or the need to improve the analyzer tool. Dependencies, not only between the various physical structures but between all configuration changes, need to be identified. With a dependency graph, the analyzer could actually search for an optimal set of recommendations that would be a better fit for the given workload.

A next step would then be the autonomous implementation of changes without interaction of the DBA. For this to be achieved, more efficient ways of creating physical structures are needed, allowing them to be created online while the system is in normal operation without slowing down or blocking user traffic.

REFERENCES

- [1] M. Stonebraker, Ed., *The INGRES Papers: Anatomy of a Relational Database System*. Addison-Wesley, 1986.
- [2] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970.
- [3] J. Peel, "Ingres: Re-Opened, Innovating and in Business," *Datenbank-Spektrum*, vol. 7, no. 22, pp. 13–19, 2007.
- [4] A. Skelley, "DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes," in *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2000, p. 101.
- [5] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated Selection of Materialized Views and Indexes in SQL Databases," in *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 496–505.
- [6] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden, "DB2 design advisor: integrated automatic physical database design," in *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*. VLDB Endowment, 2004, pp. 1087–1097.
- [7] N. Bruno and S. Chaudhuri, "To tune or not to tune?: a lightweight physical design alerter," in *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 499–510.
- [8] S. Chaudhuri and N. Bruno, "An Online Approach to Physical Design Tuning," in *ICDE '07: Proceedings of the 23rd International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 826–835.
- [9] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis, "Colt: continuous on-line tuning," in *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2006, pp. 793–795.
- [10] K. Sattler, I. Geist, and E. Schallehn, "QUIET: Continuous Query-driven Index Tuning," in *Proceedings of 29th International Conference on Very Large Data Bases (VLDB), Berlin, Germany*, 2003, pp. 1129–1132.
- [11] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood, "Automatic Performance Diagnosis and Tuning in Oracle," in *CIDR*, 2005, pp. 84–94.
- [12] W.-J. Chen, U. Baumbach, M. Miskimen, and W. Wicke, *DB2 Performance Expert for Multiplatforms V2.2 (IBM Redbooks)*. IBM Press, 2004.
- [13] S. Chaudhuri, A. C. König, and V. Narasayya, "SQLCM: A Continuous Monitoring Framework for Relational Database Engines," in *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, p. 473.
- [14] S. Chaudhuri and V. Narasayya, "AutoAdmin "what-if" index analysis utility," *SIGMOD Rec.*, vol. 27, no. 2, pp. 367–378, 1998.
- [15] G. Graefe, "Dynamic Query Evaluation Plans: Some Course Corrections?" *Bulletin of the Technical Committee on Data Engineering*, vol. 23, no. 2, pp. 3–6, 2000.
- [16] S. Idreos, M. L. Kersten, and S. Manegold, "Database Cracking," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2007.
- [17] M. P. Consens, D. Barbosa, A. Teisanu, and L. Mignet, "Goals and benchmarks for autonomic configuration recommenders," in *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2005, pp. 239–250, <http://www.cs.toronto.edu/consens/tab/>. Link visited 30 Apr 2008.
- [18] G. Haber, "Economic Impact of the Austrian Software Industry," 2003, <http://www.econ.uni-klu.ac.at/eis2003a/>, summary of the study, page 4. Link visited 16 Aug 2008.