TECHNISCHE UNIVERSITÄT ILMENAU
Institut für Praktische Informatik und Medieninformatik
Fakultät für Informatik und Automatisierung
Fachgebiet Datenbanken und Informationssysteme

Diplomarbeit

Optimiser-Based Recommendations for Physical Database Design

vorgelegt von

Alexander Thiem
Matrikel 36996

Betreuer:

Prof. Dr.–Ing. Kai-Uwe Sattler

Ilmenau, den 04. September 2008

Today's relational database management systems are made up of many complex components and managing these presents a growing challenge for database administrators. Every runtime environment can require different configurations to deliver adequate performance. Even within the same environment, demands can shift over time when workloads change. Keeping up with these demands requires continuous effort from the DBA. The goal of a modern DBMS must be to support the DBA in his work with automated processes and workflows that allow him to make quick and precise decisions. This work aims at describing and partially implementing a supportive system that will analyse the current DBMS configuration together with its workload to give recommendations on how to improve its performance and efficiency.

Die Komplexität aktueller relationaler Datenbank Management Systeme stellt eine immer größere Herausforderung an Datenbankadministratoren dar. Jede Laufzeitumgebung benötigt eine für sie angepasste Konfiguration, um performant zu operieren. Selbst innerhalb einer Umgebung können sich die Anforderungen im Laufe der Zeit ändern und eine erneute Anpassung erfordern. Dies zwingt den DBA sich kontinuierlich und intensiv mit dem System zu beschäftigen. Das Ziel eines modernen DBMS muss die Unterstützung des DBAs sein, um seine Arbeit mit automatisierten Prozessen und Handlungsabläufen zu erleichtern und ihm so stets schnelle und prezise Entscheidungen zu ermöglichen. Diese Arbeit zielt auf die Beschreibung und teilweise Umsetzung eines unterstützenden Systems, das die aktuelle DBMS Konfiguration zusammen mit dem aktuellen Anfrageverhalten analysiert und dem DBA Vorschläge unterbreitet, wie sich die Performanz und Effizienz des Systems verbessern lässt.

Contents

Eidesstattliche Erklärung / Affirmation

Ich versichere hiermit an Eides Statt, dass ich die von mir eingereichte Diplomarbeit selbständig verfasst und ausschließlich die angegebenen Hilfsmittel benutzt habe.
Diese Arbeit wurde bisher in gleicher oder ähnlicher Form nicht veröffentlicht und keiner anderen Prüfungsbehörde vorgelegt.

Hereby, I declare that I have written this work by myself without any assistance and that I have exclusively used the indicated literature and resources.
This or a similar work has not been published or presented to another examination board before.

Ilmenau, den 04. September 2008

Alexander Thiem

1. Introduction

In the last decade the ratio of hardware costs to human costs has changed dramatically. While hardware is becoming constantly cheaper the investment in qualified personnel represents the main part of most company's expenses (see e.g. [Hab03] and [IDC07]). With this background even relatively easy tasks become more and more expensive when highly qualified personnel need to spend time on them. Intelligent software tries to mitigate this effect by automating well-defined processes and by supporting the expert's decision making process. For example, a large number of supportive tools for system administrators is available such as tools to filter log files because with the increasing number of systems to monitor, manually going through log files becomes impractical. Pre-filtering based on heuristics dramatically reduces the quantity of data to observe and helps the administrator to concentrate on real anomalies.
The same applies to database administrators (DBAs) – without a proper aggregation of information the DBA is kept busy with constantly screening the system losing valuable time to actually maintain it. The complexity of a database management system creates the challenge of finding the best configuration for a given environment and workload. The DBA has to manually monitor the system over time to decide what needs to be done to either increase performance or to keep the performance at a constant level under given environmental restrictions.
Other than the application developer the DBA has no influence on the design of SQL queries that are executed on the database. He is limited to the configuration of the DBMS and the physical design of the database. These performance factors include DBMS settings such as caching and other memory options, the creation of secondary indexes and column statistics, finding the best storage structure for a table and the partitioning of tables. For all these options there is no universally best answer – there is a virtually infinite number of combinations that need to be investigated for one single environment and workload to find the appropriate contextual settings. The choice of the right configuration is driven by additional objectives, e.g. getting the highest possible performance at the cost of adding more hardware resources, finding the most efficient utilisation of available resources or even minimising the load and freeing resources.
To be able to perform such a model-based operation of a DBMS a far-reaching knowledge of its usage profile is required. The workload must be analysed over a period of time and evaluated to decide what might be beneficial to fulfill the requirements. It might even be necessary to apply certain changes to see the reaction of the system and to decide whether the change was beneficial or not. This is where automation can help to reduce the human effort allowing the expert to concentrate on numerically fewer but more important problems.
This work describes the concept of an advisory tool for the relational database management system Ingres that is able to monitor the workload of the DBMS and to recommend changes to the configuration that are believed to improve overall performance of the system. This design analyser was partially implemented with a number of changes in the DBMS core to monitor the system, a daemon tool to collect the data and a client to analyse the data and present recommendations to the DBA.

1.1. Motivation

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[1] or application frameworks like Ruby on Rails[2] that create their database schemes automatically the application developer can loose sight of the database and DBMS and may not even think about an efficient 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.
Although Ingres, as one of the first relational database management systems, was a very innovative system for a long time – e.g. being the first DBMS to use histograms for more precise cost estimates ([Koo80]) – it has suffered from a slowdown in active development lasting for more than a decade until recently. Hence, it is lacking features to automate processes and most of the DBA work still has to be done manually. Expanding Ingres with an advisor tool to recommend configuration and design changes will greatly enhance the value the DBMS offers to DBAs and developers. The fact that the Ingres source code has been released under the terms of the GNU Public License Version 2 makes it easy to develop new features for the DBMS and to contribute them to the project.

1.2. Scope

The main objective of this work is to describe the concept of a supportive system that enables Ingres to collect statistical data and to process this data to give recommendations on how to improve the physical database design of the system.

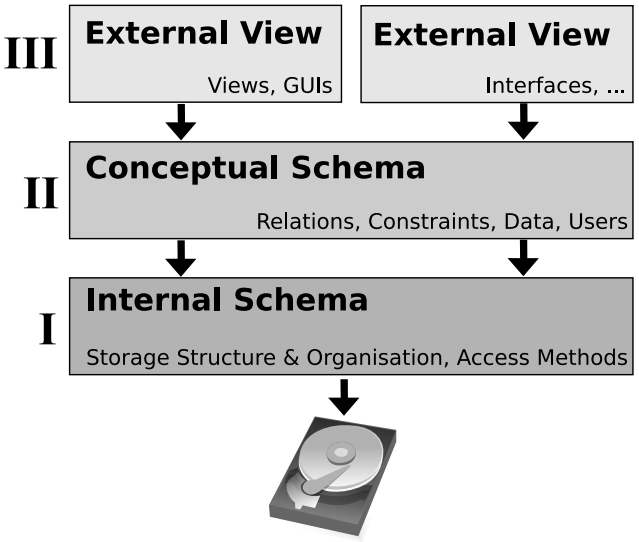


Figure 1.1: Three-Schema DBMS Architecture

[1] http://www.hibernate.org/
[2] http://www.rubyonrails.com/

Following the Three-Schema architecture shown in figure 1.1 the physical database design describes the internal level of the DBMS – the organisation and structure of data storage and the access methods (indexes, partitions, etc) on that data. This work will describe how information can be collected on both levels one and two, how it can be analysed and presented to the DBA who then can act on the recommendations provided by the system. The scope of this work is limited to the basics of what kind of information is of interest and how it may be retrieved and processed. Many sections of this work are focused on only one or a few parts of the whole to describe them in greater detail. The implementation of the system is experimental and should be considered as a proof-of-concept.

1.3. Structure of the Work

The rest of this work is structured as follows: Chapter 2 gives an introduction to the concepts behind Ingres and the principles of database tuning both manually and automatically. Chapter 3 presents research work and existing advisor tools in other systems and analyses how they perform. In chapter 4, the architecture of the proposed advisor is discussed in detail followed by the description of its implementation in chapter 5. Chapter 6 shows how the tool performs currently and chapter 7 provides an outlook on possible enhancements in future versions of the advisor.

2. Fundamentals

2.1. Ingres

Ingres originated at the University of California, Berkeley where Michael Stonebraker and Eugine Wong started a research project on implementing a relational database management system based on Edgar Codd's work ([Sto86], [Cod70]). 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 renamed to Ingres Corporation later, Stonebraker commercialised 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 came back as an independent company in 2005. The latest version, Ingres 2006, was released under the terms of the GNU General Public Licence Version 2. A more detailed look at Ingres' history can be found in [Pee07].
As [Thi08] already gives a detailed view on the internal architecture of the Ingres 2006 DBMS, this work will only briefly talk about the basics to give an understanding of Ingres' structure.

2.1.1. Architecture

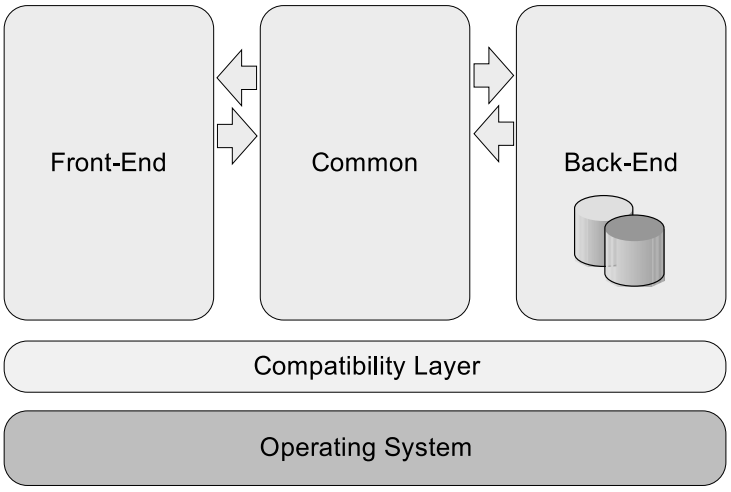


Figure 2.1: Ingres Code Architecture [Thi08]

Figure 2.1 shows the conceptual structure of Ingres. The DBMS is divided into four main parts: the front- and the back-end, the common part and the compatibility layer. The front-end holds all the user facing interfaces such as the command line tools and the forms based and visual tools. The back-end contains the actual DBMS with the parser, the optimiser, data storage, etc.

Both parts share a number of functions in the common part which also includes the network interfaces over JDBC, ODBC or .NET. The compatibility layer is responsible for transforming Ingres' internal function calls into platform specific system calls. This way, the DBMS code itself can be kept platform independent and only the compatibility layer needs to be ported to support other architectures.
With version 6 of Ingres, released in 1990, the original monolithic source code was rewritten into a number of modules that should help to keep Ingres maintainable. Every code module represents a server facility that is responsible for a specific task – for example, there is an optimiser facility, a data management facility, a communication facility, etc. The idea was that facilities should only interact over a defined caller method to avoid cross-dependencies and side-effects.

2.1.2. Server Facilities

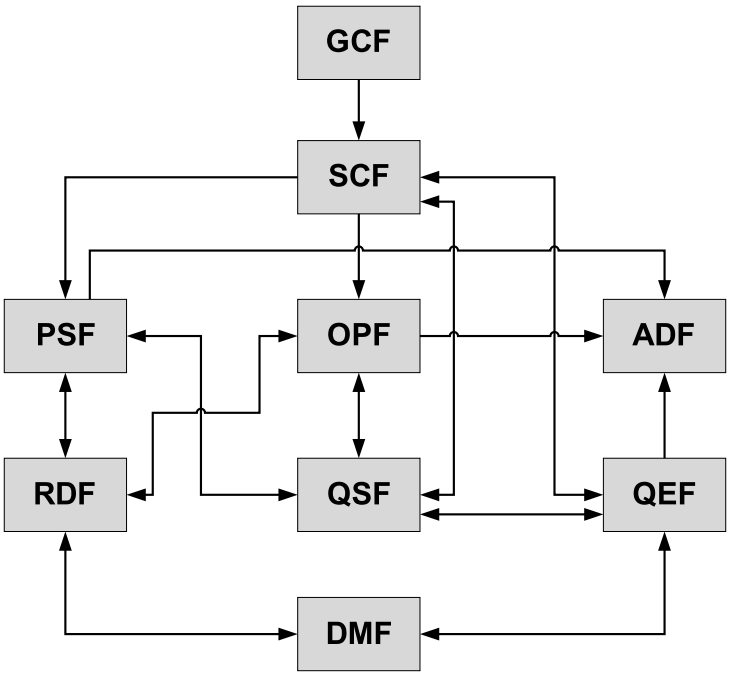


Figure 2.2: Ingres Server Control Flow [Thi08]

Figure 2.2 schematically shows the control flow between the most important server facilities within Ingres. The *general communications facility* (GCF) connects the back-end to the outside world passing all requests to and from the DBMS. The *system control facility* (SCF) is taking over and then calls all the other facilities needed to process the query. The *query storage facility* (QSF) acts as a global memory for the back-end holding all structures needed by more than one facility. SCF places the query in QSF and calls the *parser facility* (PSF) which reads the query from QSF and translates it into a parse tree. While parsing, PSF calls the *relation description facility* (RDF) to get information about database objects such as tables, columns and views. RDF itself calls the *data management facility* (DMF) to read the system catalogues that contain the requested information. PSF also uses the *abstract datatype facility* (ADF) to process constants and expressions in the query text. The parse tree is stored in QSF and PSF

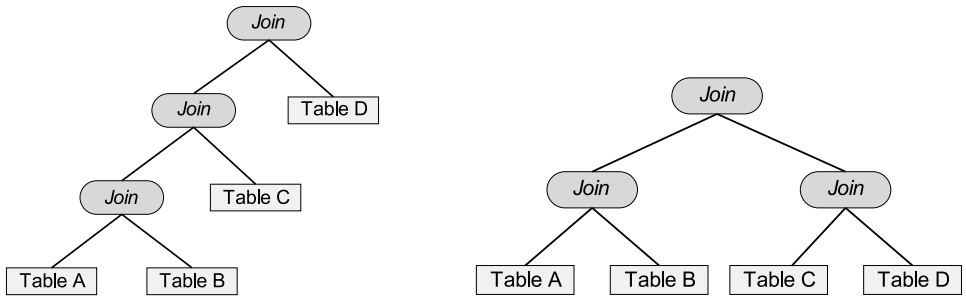
returns control to SCF. Next, the *optimiser facility* (OPF) is called to create an optimal execution plan. OPF decides which indexes will be used and enumerates the possible join orders of tables to find the cheapest way of executing the query. The resulting query execution plan is again put back into QSF and SCF continues with calling the *query execution facility* (QEF). QEF uses DMF to load and store data and ADF to compute results of expressions and functions. SCF then returns the final resultset to GCF and back to the incoming user interface. ([Ink04])

2.1.3. Relational Concepts

Ingres was designed around the idea of a relational data management. Hence, nearly everything in Ingres is handled as a relation. The system catalogues that contain all the meta data of database objects such as existing tables, columns, users and much more, are in fact stored in a database that can be accessed in the same way as any other database in Ingres. With this concept Stonebraker and Wong tried to minimize the effort of internal data handling because the DBMS can use its own data access and manipulation functionalities. Indexes are stored as tables with a column containing the key and a TID column. Therefore, indexes can be used by the optimiser by simply adding them to the list of joining tables. Even realtime monitoring data and DBMS runtime settings can be accessed and controlled over an SQL interface as described further below in section 2.1.5.

2.1.4. Optimisation

Query optimisation in Ingres is a three-phased process. First, the query is getting rewritten and simplified by applying a set of rules such as eliminating NOTs with DeMorgan's laws, transforming boolean expressions into conjunctive normal form and flattening subselects into a single statement. In this phase, the optimiser also adds the list of available indexes to the list of joining tables.
In a second step, the optimiser enumerates all possible shapes of the execution tree and calculates the cost of those trees to find the cheapest way to join the tables. Unlike most other optimisers, Ingres doesn't restrict the shape of the tree to only left- or right-deep ones but does allow every possible shape as seen in figures 2.3a and 2.3b. This greatly improves the chance to find an optimal plan, however, this also leads to a much larger search space and complexity for the enumeration algorithm.



(a) Left-deep (b) Bushy

Figure 2.3: Binary Join Tree Variations [Thi08]

As indexes are also tables the optimiser adds them to the enumeration process of table joins which adds even more possible combinations. [Koo80] computes the upper bound of trees to create to $\sum_{i=0}^{|I|} \binom{|I|}{i} * \frac{(2*n-2)!}{(n-1)!} * 4^{2*n-2}$ where I is the set of available indexes and n is the number of tables plus the number of used indexes i. This means that Ingres is enumerating all structurally unique tree shapes and permutes the order of tables to be joined through all tree leaves – this is done starting with no indexes added, then with one of the $|I|$ indexes, cycling through all available ones. Then with all possible sets of two indexes, then with three, etc, until the cost of every combination of tree shape, table permutation and index set has been calculated or a given timeout stops the enumeration. Ingres tries to reduce the search space by pre-filtering indexes that do not represent a matching predicate ($T.a = const$) or a joining attribute ($T.a = S.b$) but with a high number of tables to join this still makes it virtually impossible to find an optimal plan in reasonable time. Since Ingres r3, released in 2004, the optimiser was enhanced with a greedy algorithm ([Ink03]) that computes tree fragments of three tables at a time when handling with ten tables or more which greatly reduces the number of possibilities but even though a high number of indexes per statement can still lead to an extremely long runtime of the enumeration. In newer releases of Ingres a default enumeration timeout was introduced. The optimiser stops searching when the time it used so far exceeds the estimated execution time of the currently cheapest plan.
After a plan was found that is either the cheapest plan overall or the cheapest plan that was constructed before enumeration timed out the optimiser continues with the code generation phase in which the plan is being augmented with action and node headers. Action headers describe what needs to be done on a node when the result from the lower subtree is being received such as sorting the result or calculating an aggregate. Node headers, containing table and index control blocks, are added to nodes where disk access on tables is needed. The final query execution plan (QEP) is then stored back in QSF and SCF calls the query execution facility to execute the statement.

2.1.5. IMA

The *Ingres Management Architecture* (IMA) is a flexible framework that offers an extensible relational interface to read and write internal DBMS data not only of the local Ingres instance but also of remote servers. With IMA it is possible to dynamically access in-memory structures within the DBMS which is mostly used for realtime monitoring but also allows control of servers and sessions. IMA can be accessed over SQL, however, while the system catalogues are stored in a native database the IMA database (IMADB) is only a code construct simulating relations that contain the requested data.
Figure 2.4 shows the structure of IMA. A client application is communicating with the DBMS over a management protocol that primarily is SQL but can also be the *simple network management protocol* (SNMP) or the *common management interface protocol* (CMIP). The DBMS provides a *management information base* (MIB) which forms a set of information objects that can be monitored and manipulated. Depending on whether the IMA request is for a local or a remote object the MIB server either passes the request on to the local *management object* (MO) module which then calls internal code or the request is sent over the network using GCF to a remote Ingres instance where it gets processed by the remote MO module.
IMA is using a class concept where a class is the description of a set of data points together with methods to retrieve and store the data. Each class can have either one or more instances of

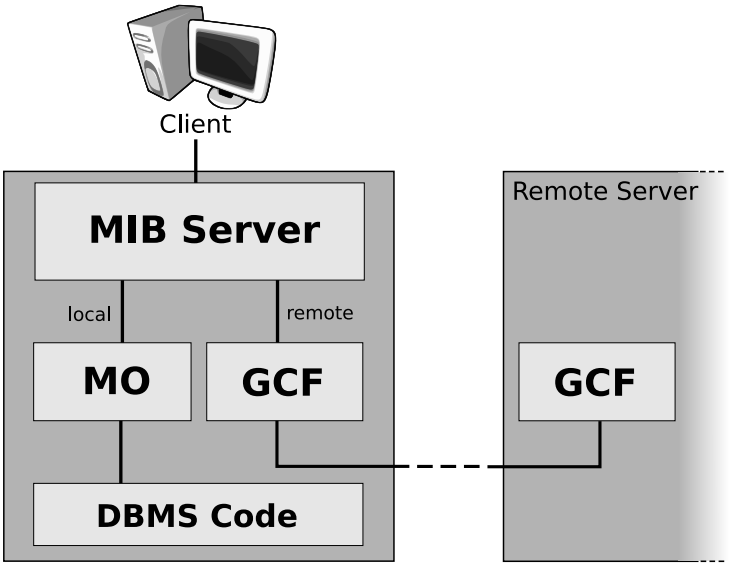


Figure 2.4: Ingres Management Architecture

objects that contain the actual data. A global counter for all current locks in the DBMS would have a single instance whereas a lock counter per session would have one instance per session. All classes attached to the MO module can be registered in IMA to represent a table in IMADB – details on how to create new IMA classes and objects are part of section 5.1.
IMA was created to allow fast and easy extension of monitoring tools for local and remote command line and visual interfaces. Ingres already provides a huge number of single data points from all the server facilities over IMA. The table ima_mib_objects in the IMADB lists all available managed objects and currently contains thousands of rows. ([Bro93])

2.2. Performance Tuning

This section gives an introduction to DBMS performance tuning. It will start with a list of principles that should be kept in mind while tuning a system. This is followed by a description of factors that influence performance in a DBMS and a description of indicators that can be used for monitoring a system's performance. The last part of this section will then talk about the idea of autonomous DBMS tuning.

2.2.1. Tuning Principles

Tuning in general can be described with a control loop as shown in figure 2.5: The subject to tune needs to be observed to see how it currently acts. Based on the observed control variables and their deviation from the target a decision has to be made how the actuating variable needs to be adjusted. Then, the actual change is performed and observation of the subject continues.
In the same way the DBA tunes his DBMS – he watches 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 and others. The DBA then implements the changes to see how the DBMS reacts and what changes he needs to

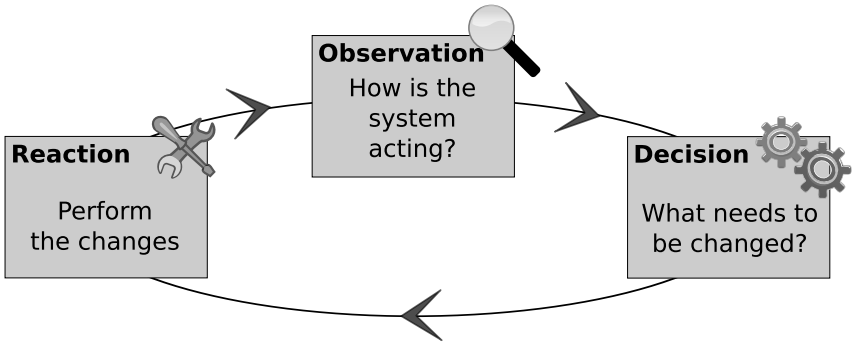


Figure 2.5: Tuning Control Loop

do next.
A comprehensive view on database tuning can be found in [SB03]. From that book the following principles can be derived:

1. The solution to high load on the DBMS in terms of high disk I/O, CPU or RAM does not necessarily have to be additional hardware resources – it is more important to reveal the cause of the load first to see if the problem can be resolved in software.

2. The time a statement takes to execute should always be considered together with its frequency because a slow query that is executed only once or twice shouldn't get more attention than a slightly faster query that is executed dozens or even hundreds of times.

3. The components of a system are never under full load all at once – there is always one component slowing down the rest. Finding and eliminating bottlenecks can speed up things tremendously.

4. Every tuning measure causes costs – additional hardware costs money, creation of additional database objects costs disk space and time. Costs and benefits need to be weighted against each other.

5. Objects such as indexes and materialised views require maintenance (when issuing updates, inserts, deletes), therefore, unused objects have negative influence on performance and should be considered for removal.

2.2.2. Performance Factors

Factors that influence the performance of a DBMS have already been mentioned in this work without further explanation. Although this may be basic knowledge for most of the readers this section gives a list of the most important factors together with a rationale why they influence a DBMS in general and how they work for Ingres in special.

Secondary Indexes

The data in a table is organised along the primary key of the table. Depending on the underlying structure of a table the primary key is used as the index of a tree or a hash map. A lookup on the

primary key in a tree is fast and efficient because the tree is sorted in such a way that the DBMS can easily find the requested key and its corresponding data page. For example, the primary key of an employee table may be the employee number because most queries will use that number to find an employee. However, queries on name or address can also be very common but since the table is sorted by employee number a lookup for the last name requires a full scan of the whole table. This is were secondary indexes (or for the ease of use just indexes) can help to improve performance. An index is a pair of two values: The attribute that is being indexed – in the employee example this would be the last name – and a tuple identifier (TID) which is a pointer to the data page that contains the row that belongs to the last name. This index is sorted by last name which enables the DBMS to answer queries for last names quickly.
As described in section 2.1.3, secondary indexes in Ingres are implemented as tables with two columns for the attribute and the TID. As a consequence indexes can also have different storage structures and can even directly be used in SQL statements. The Ingres optimiser as described in section 2.1.4 uses indexes by adding them to the list of tables for which the enumeration will find a join order – this is an important difference to most other DBMS. Other systems such as the MS SQL Server optimiser (see [CN07]) explicitly request indexes to build a plan, so they can have a large number of indexes to choose from. With the enumeration in Ingres, however, a large number of indexes leads to a more than exponentially growing search space for a query execution plan. Therefore, it is crucial to find the number of useful indexes without adding too many of them. Even when no indexes are added explicitly by the DBA Ingres implicitly creates them for constraints such as a unique column or a foreign key column so that checking the constraint can be done efficiently.
To give an example: Given are the two tables

employee *(emp_id, name, last_name, social_no, dep_id → department)*
department *(dep_id, building, floor, director → employee)*.

The two foreign keys will create two secondary indexes. As mentioned before, an index on employee.last_name would make sense. The social security number is definitely a unique value throughout the employee table – a unique constraint adds another index. In a normalised schema department.building and department.floor are probably also foreign keys to other tables adding two additional indexes to the picture. Using the formula provided by [Koo80] a join between the two user tables and the six index tables has $\sum_{i=0}^{6} \binom{6}{i} * \frac{(2*(2+i)-2)!}{((2+i)-1)!} * 4^{2*(2+i)-2} \approx 4.65 * 10^{15}$ possible QEPs. This is a very simple example and the Ingres optimiser is smart enough to exclude those indexes that are useless for the query so that the number of combinations in this case may be much smaller but on more complex schemes and queries the number of user created indexes and constraint indexes can be the cause for bad performance.
Therefore, finding the best index set for a database in Ingres means creating the right number of indexes to speed up query execution without causing query compilation to exceed a reasonable runtime. Even indexes that are not considered while creating a QEP can negatively influence a system's performance – as stated in tuning principle #5, modification on the indexed data by inserts, updates and deletes requires an index to be updated accordingly. For indexes that are never used by the optimiser this leads to an unnecessary overhead.

Statistics

[Koo80] proposed and implemented the use of histograms in Ingres to allow more accurate estimates of cardinalities. The knowledge of cardinalities supports the optimiser in his decision to

efficiently join tables together. Let's pretend, the employee table from the example above holds 500 rows while the department table has only 20 rows. Let's also pretend that there are three employees with last name Smith in the records. Without that knowledge the query

```
select department.dep_id, employee.name
  from employee, department
  where employee.dep_id = department.dep_id
        and employee.last_name = 'Smith'
```

could be executed by first joining department and employee resulting in $500 * 20 = 10000$ rows and then searching those 10000 rows for the occurrences of employees with last name Smith. However, when the optimiser knows that there are only three Smiths in only 20 departments the query would probably executed by first searching the 500 employees for Smith and then join only those three results with the 20 departments touching only $500 + 60 = 560$ rows instead of 10000.

With the use of table statistics based on histograms the optimiser knows the overall number of tuples and the distribution of values in a table. In Ingres, statistics need to be created manually – for tables without statistics the optimiser assumes an equal distribution of all values and estimates that an exact match ($T.a = const$) will always return 1% of the tuples and a range qualifier ($<, \leq, >, \geq$) will always return 10%. Of course, those assumptions probably miss the real distribution of values by far – here, the creation of statistics can heavily improve performance.

Other than indexes, statistics require no active maintenance when the content of a table changes – they remain static and do not cause additional overhead. However, this also means that when the data distribution significantly changes over time the histogram becomes outdated and provides wrong estimates. If that is the case the statistics need to be updated or recreated to reflect the new distribution.

Partitions

For a full table scan, partitioning of large tables into smaller chunks of data can help to distribute I/O load on several hard disks taking advantage of multiple read heads scanning the table in parallel and is therefore a good way to eliminate bottlenecks as stated in principle #3. Partitioning in Ingres can be done either by using a key column or with a random distribution. When using a *random* distribution new rows are added to the different chunks randomly, however, no known sort order means that the system needs to touch every partition chunk for every query. When using a key column the decision in which chunk a new row is inserted depends on the value of the column used to create the partition and therefore the system knows in which chunk the requested rows are. A *range* distribution will put all values in the interval $[v_0, v_1)$ into the first chunk, values in $[v_1, v_2)$ go to chunk two, and so on. With a *list* distribution every chunk is associated with a user-defined list of values. New rows are inserted into the chunk where the key column matches one of the values in the chunk's list. The third key column method is a *hash* distribution where every chunk has a hash value – key columns of new rows are being hashed and put into the corresponding chunk.

As with indexes, the sort order of partitions only helps when the query is actually using the sorted column – when that is not the case even the column key partitioning requires all partition chunks to be read. This can be countered by the creation of secondary indexes on the partitioned table.

Materialised Views

A database view can help to make accessing a schema easier by hiding complex joins behind a one-table query. It also allows adding an additional security level to the database by presenting the user a different view on the real tables. However, computing the results of a view every time it is referenced in a statement can be quite expensive. As a kind of cache, Oracle introduced the concept of materialised views – the results of a view are computed once and stored in a separate table which is then used instead of the view. The materialised view requires maintenance as it becomes outdated when the underlying tables of the view change. Depending on the DBMS and the settings the materialised view may be updated immediately, on a regular basis or even manually. As with indexes, this overhead is worthwhile only when the materialised view is actually used – if not, it should be removed.
The current release of Ingres doesn't support the concept of materialised views, yet.

Storage Structures

The storage structure of a table can have significant influence on the performance. The default structure for new tables in Ingres is a *heap* where new rows are simply appended to the end of the file – hence, inserts in the table are fast, however, thinking of a lookup in the table the system needs to scan the whole file every time because of a missing sort order. Also, freed space after deletes can't be reused until the table structure is getting reorganised. [Cora] recommends heap only for very small tables or tables where no sort order is required. Heap is also the most efficient structure for bulk loading of data.
A more sophisticated way of storing data is by using a key to determine the location of rows on their data page on the disk. The simplest key structure in Ingres is the *hash* storage where the value of the primary key of each row is getting hashed and put onto the corresponding page on the disk. An exact match ($T.a = const$) can be processed most efficiently with a hash structure as the value only needs to be hashed to know on which page the data is stored. However, range queries or a LIKE operator with a wildcard can't benefit from the hash and the system needs to perform a full table scan again. Another disadvantage is the way hash reacts on growing tables – when a page is full the system creates overflow pages to store new rows. Those overflow pages form a linked list connected to the main page of the particular hash key – now even an exact match forces a scan of the complete list. Overflow pages can be removed by reorganising the table where the hash function will be adapted to address all pages directly.
Using the *ISAM* storage structure the data is organised in a static tree. The primary key is used to build an index tree with the actual data in the leaf pages. The index allows efficient lookup of exact matches as well as range scans and LIKE prefix searches ($T.a\ like\ 'S\%'$), however, as the index is static ISAM also uses overflow pages when the main data pages are full. As with hash, a lookup on overflow pages becomes inefficient and the table needs to be reorganised when there are too many.
The fourth storage structure available in Ingres is the *B-Tree* which is the most flexible kind of structure. B-Tree uses an index similar to ISAM but it is not static which eliminates the problem of overflow pages when the table grows. Instead, the tree is dynamically rebalanced when pages get added or removed. The overhead that is added by an automatic reorganisation is outweighed by the performance that is in most situations better than with the other storage structures.

Configuration

While the configuration – or the settings – of the DBMS don't necessarily belong to the physical database design they still have a big impact on the system's performance. A typical configuration file of an Ingres installation holds more than 400 lines but there are even more settings available. Every facility in Ingres provides a number of settings that range from simple and straightforward things like the user connection limit to less transparent things like the threshold for the logging facility that specifies when a log buffer is written to disk and when the disk access will be delayed. Most settings can be changed by hand but some depend on other settings and may be overwritten based on rules like the maximum number of users in OPF is 0.2 times the user connection limit of the DBMS. Not all configuration settings are fully documented and for most performance affecting settings there are only rules of thumb that apply to certain environments.

2.2.3. Performance Indicators

Before the DBA can react on a performance problem, he needs to know that there actually is a problem. To get to know about a system's performance there is a number of indicators that can be watched. This section lists the most promising performance indicators and how they can be used with Ingres.

CPU Time and Disk Input/Output

While CPU time and disk I/O are two different indicators they can never be interpreted independently – when one is in idle it is probably waiting for the other as stated in principle #3. The experiments that will follow in chapter 3 have shown that a system can easily spend 80% and more of its time with reading data from disk while the CPU is in idle state waiting for the read to finish. The internal cost model of the Ingres optimiser is based on CPU time and disk I/O – every action, a sort, a join, is associated with CPU costs and for every disk operation the number of accessed pages is estimated to calculate the I/O costs. Those cost values can be seen in the visualisation of a QEP (as described in appendix B) – part of every node in the tree are two values prefixed with *D* and *C* representing the two individual cost estimates.

```
1                Cart-Prod
2                Heap
3                Tups 89235168165888.000 Pages 89235168165888.000
4                D137709846528.000 C892353249280.000
5              /                            \
6            Proj-rest                      Proj-rest
7            Heap                           Heap
8            Pages 1133493 Tups 1133493     Pages 971924 Tups 78725824
9            D141689 C11335                 D98409 C787258
10         /                               /
11  protein                             neighboring_seq
12  Heap                                Heap
13  Pages 1133512 Tups 1133493          Pages 787273 Tups 78725824
```

Figure 2.6: QEP of a Cartesian Product

Figure 2.6 shows an Ingres QEP of a cartesian product – as it can be seen, the estimated costs of the cartesian product in the top node are higher in the order of magnitudes compared to the costs of the lower nodes.
The less pages that need to be accessed on the disk the less disk I/O will block the system – the number of pages may be reduced by an index that can replace the underlying table in a query or a B-Tree structure where lookups can be performed without a full table scan. To reduce high CPU time the QEP may be searched for expensive aggregates or unnecessary sorts caused for example by a DISTINCT that often can be left out.

Outdated Statistics

As explained in section 2.2.2, statistics in Ingres are static and must be updated or renewed manually when the data distribution of a table significantly changes. One possible indicator for the need to renew the statistics can be a high number of inserts and updates on the table – the probability is high that the distribution changed when a lot of new data was added or existing data was modified. The Ingres trace point QE90 (see appendix B for the usage of trace points) looks similar to a QEP but lists estimated and actual results per tree node.

```
1                                  ----------
2                                  HJOIN
3                                  at        0
4                                  et        9
5                                  ad        6
6                                  ed       11
7                                  ac        9
8                                  ec      684
9                                  ----------
10                               /             \
11                 ----------                   ----------
12                 ORIG                         ORIG
13                 at      478                  at       12
14                 et        9                  et       39
15                 ad        6                  ad        0
16                 ed        7                  ed        4
17                 ac        7                  ac        0
18                 ec      478                  ec      157
19                 ----------                   ----------
```

Figure 2.7: Output of Trace Point QE90

Figure 2.7 shows a simplified example of QE90 with a join of two tables. The values in lines with *et*, *ed* and *ec* show the estimated tuple count, disk I/O cost and CPU cost respectively. The lines with *a* show the actual value that is known after execution of the query. As it can be seen, the estimated number of tuples, especially in the left table, doesn't match the actual result – with the creation of statistics this situation can be improved.

Index Usage

As described above, indexes in Ingres can significantly improve performance when they are used appropriately – but in the same way they can slow down a system dramatically. There are only rules of thumb of where the border is between enough and too many indexes on a table or

within a database so that a good knowledge of the workload is required to tune a system with the creation of indexes. A QEP as in figure 2.6 will reveal what indexes Ingres has chosen to execute a query – an index node will be marked with *I(Table Name)*. In addition, trace point OP161 prints the list of all tables and indexes that are considered for the execution of a query, including indexes that are not part of the final QEP. This list can be taken as an indication for indexes that are never used by the optimiser and therefore only slow down the enumeration phase.

2.2.4. Autonomous Tuning

The manual tuning of a DBMS is a time-consuming task with the biggest part being the constant observation of the system. Here, the DBMS can support the DBA with the automation of reoccurring processes and even with automated changes in the system. When looking at embedded systems where the DBMS is only one part of another application, hidden from the user, there may not be a database administrator who can adapt the system to a changing workload or environment. Autonomic computing ([KC03]) aims at creating self-managing applications that know how to maintain, optimise and even heal themselves. Autonomous tuning as one part of the autonomic computing still follows the same control loop as manual tuning does – the system needs to be observed, problems need to be identified and decisions need to be made to perform the necessary changes. For an automated system these steps can be separated in I) the monitoring of the system, II) the collection of the monitored data over a period of time, III) the analysis of this data and IV) the presentation or automatic implementation of changes based on the analysis.
There are several approaches of autonomous database tuning that implement those four steps at different levels. The most sophisticated approach certainly is the independence of human intervention by building a self-sustaining and self-maintaining system. The internal layer, the physical database design, of such a system is completely hidden from the user and the DBMS controls storage structures, access methods and settings all by itself. The identification of problems, the decision how to solve the problems and the implementation of changes is completely automated and done during normal operation of the system. Because there is an infinite number of scenarios and environments a fully autonomous system must be either able to dynamically react to unknown situations or it can only operate within a fixed environment. The first may produce a high load on the system caused by the overhead to find the best design for every possible configuration – the latter will be impractical in many situations because of the large variety of environments. Requirements, coming from users and applications either need to be formalised in a way the system can turn them into design decisions or they will need to be guessed based on heuristics with the risk of making wrong decisions.
In the next lower level of autonomous tuning the DBA doesn't get replaced by the system but instead he is supported in his day-to-day work. The human factor remains and the DBA can keep maintaining the system based on informal knowledge. The DBMS offers semi-automated support in form of scheduled maintenance tasks and event alerters. Most of the tasks can be performed without intervention while for bigger tasks and problems the DBA is being alerted and provided with details and possible solutions. Those systems can identify problems and recommend changes to overcome them. To a certain degree, they also allow prediction and prevention of problems by analysing trends in the data they monitor.
One step further below is the system which supports the DBA by offering wizards and advisors

that are started manually to scan and analyse the DBMS. The DBA either waits for the problem to happen or he needs to run the tools periodically to prevent problems.
The less automation the less overhead and resource consumption is caused in the day-to-day usage of the system. However, when problems occur the automation in online systems leads to quicker resolutions or it can even prevent problems beforehand while offline systems can only report the past.
The tuning tools that are delivered with most of today's commercial DBMS (as seen in the next chapter) are somewhere between the lowest level of autonomous tuning in form of passive wizards and the more sophisticated level of background tasks and automated information gathering used to build alerters. All those tools offer a mixture of plain reporting of data, problem prediction and automatic problem identification and solving.

Autonomous Tuning in Ingres

Except the basic monitoring, as described in section 2.1.5, Ingres does not yet offer the features that are needed for an autonomous database tuning. It lacks a more comprehensive monitoring, the collection and persistent storage of the captured data and a way to process and analyse this data. This work aims at describing a concept to expand Ingres with these features to initially allow the creation of a passive advisor tool but also to provide the data that is needed to implement the next steps of problem prediction and automatic problem solving. Code contributions of this work will be I) an enhanced workload monitoring that records not only plain statements but also details such as estimated and actual execution costs and much more, II) a data collector that allows long-term storage of the monitored data and III) an advisor tool that performs analysis on the data to recommend changes to the physical database design to improve the workload performance.

3. Related Work

This chapter looks at existing approaches of autonomous database tuning as described in section 2.2.4. First, an overview on past and current research in the area of autonomous tuning will be given. Then, three different implementations of autonomous tuning in commercially available DBMS will be presented.

3.1. Research

The research area of autonomous database tuning is a field of growing interest both in the academic world and in the industry. Early work dates back to the 1970's when [CH76] proposed a concept for a self-adaptive database management system that was able to recommend secondary indexes on single-relation queries. The system was monitoring access patterns on the database to gather statistics that were then used to predict the need of indexes. [RS91] and [FON92] then used the today most common approach of recording a real workload during normal operation of the system to "*find a good physical database design, i.e. one as good as one that a competent human database designer with the same information would produce*".
While early works such as [BPS90] created their own cost model to identify an optimal configuration, this task was later turned over to the internal optimiser of the DBMS. The *What-If* approach ([CN98], [LL02], [Sch06]) allowed the instrumentation of the optimiser by feeding it with hypothetical – not yet materialised – physical structures. Accurate cost estimates were now possible without having to copy or imitating the cost model of the DBMS in an external tool.
With a growing dependency on automatic designing there is also the risk of making wrong decisions. [CBTM05] bemoans the lack of meaningful benchmarks for auto-tuning systems and states that most works show the efficiency of their tool itself but not the usefulness of its design recommendations. [GA08] describes the risks of a *What-If* or optimiser-based approach as design recommendations closely depend on the estimates of the optimiser which could be far off the real costs and which would therefore lead to wrong decisions. He tries to find a metric for a physical design to measure its quality and to minimise the risks.
The following is a small selection of noteable research projects based on current database management systems.

Microsoft AutoAdmin

Since the mid 1990's the Microsoft Research project AutoAdmin is the origin of features for the autonomous database tuning in the Microsoft SQL Server. It started as an index recommender based on the creation of *What-If* indexes and was later expanded to the physical structures of materialised views and table partitions. The selection of index candidates is done by intercept-

ing the explicit index requests in the MS SQL optimiser and augmenting the query execution plan with details about these requests. This plan is finally translated into an AND/OR tree that contains all index requests of a single query where orthogonal requests are AND'ed and mutual exclusive requests are OR'ed. The trees of all queries in a workload are then AND'ed to result in an overall index request tree that can be used to find an optimal index set ([BC06]). A look at the development of the AutoAdmin project and its more recent state can be found in [CN07]. This work also discusses the different tuning models described in section 2.2.4: The offline tuning with a passive advisor tool which is how the AutoAdmin project began, an alerter model that continuously monitors the system and alerts the DBA when action is needed and the online tuning in which the system automatically decides and performs the tuning all by itself.

DB2 – SMART and QUIET

The DB2 SMART project is part of IBM's vision of an autonomic computing ([KC03]). It does not only focus on the physical database design but also includes advisors and wizards for the system configuration, maintenance and task scheduling, system health and data recovery ([LL02]). Instead of looking at indexes, materialised views and partitions one after each other, the DB2 design advisor incorporates dependencies between the various physical structures to get better results. A materialised view itself is stored in a table so that creating an index on it makes sense or the view can even be partitioned. The advisor defines strong and weak dependencies between the features that are used to form the search space for possible tuning measures on a given workload ([ZRL+04]).
The QUIET approach ([SGS03], [SGS04]) aims at the fully automated management of indexes by monitoring the workload, finding and evaluating index candidates and then creating them online during normal operation of the DBMS. The work mentions an idea of [Gra00] with the possibility of exploiting full table scans to create indexes on the fly while executing a query.

Oracle Automatic SQL Tuning

As presented in [DDD+04], Oracle aimed at including all the tuning features right into the internal optimiser that is controlled by special SQL commands. The optimiser can be put into a tuning mode where it has more time than usual to find an optimal execution plan and also includes tree shapes that otherwise wouldn't be considered (see figures 2.3a, 2.3b). The tuning includes the already known *What-If* approach but with the concept of SQL Profiles, Oracle added a new way of performance optimisation. Table statistics do not always lead to an optimal execution plan because the optimiser may still be wrong with estimates in intermediate results. An SQL Profile is a set of statistics specific for one single statement. These special statistics are collected by partially executing the statement and gathering cardinalities of inner nodes of the plan. The Profile is then transparently used by the optimiser everytime the statement is executed.
Oracle also suggests to restructure statements when it detects badly written SQL based on heuristics. These include syntax issues, that prevent certain optimisations of the statement, semantic issues, that can cause poor performance and design issues, where the developer probably used the wrong SQL.

PostgreSQL

Because the PostgreSQL code is open source there is quite a number of academic projects that aim at adding features for autonomous tuning. [Sti04] and [Sch06] discuss the implementation of virtual or *soft* indexes that allow the application of the *What-If* approach by feeding the internal PostgreSQL optimiser with hypothetical structures. [Lue06] then focuses on the automatic selection and creation of these soft indexes following the idea of a self-sustaining system.
Similar to the SMART project, [ML05] suggests a global tuning system for PostgreSQL that respects side-effects and interdependencies between tuning measures instead of tuning only locally and risking an overall performance drop when changes conflict. Sensors throughout the system perform a local monitoring of a single DBMS component and report problems when the observed data differs from the expected target. A central coordinator then either has the global knowledge to decide for the cause and solution of the problem or he passes the alert to software agents that are local in the several components. The knowledge is distributed amongst those agents which then suggest possible solutions to the same problem and the central coordinator decides which solution wins.

3.2. Commercial Systems

This section will show design advisors in commercial database management systems and their ability to improve the system's performance. The ones that were looked at are *Oracle 11g*, *Microsoft SQL Server 2005*, and *IBM DB2 9.5*. All three DBMS include features to monitor the system, record the workload, analyse the data and recommend changes to the physical design of the database. The available tools were tested for their effectiveness by analysing a given workload to improve the performance of the system.

3.2.1. Experimental Setup

The machine that was tested on was a Windows Server 2003 R2 desktop system with a two-core Intel Pentium D processor with three gigahertz clock frequency, four gigabytes of main memory and a 150 gigabyte hard drive.
The advisor tools were tested against a database populated with data from the *Non-Redundant Reference Protein* (NREF) database as described in [CBTM05]. The schema 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 20 gigabytes or more, making sure that the database is significantly larger than the system's main memory. For this experiment a mixture of the NREF2J and NREF3J query sets with 50 simple structured selects was used[1], measuring the time needed to execute those statements on the DBMS. For each test all queries were executed in one batch. This batch was repeated three times to minimise influence of local anomalies. The tests were first performed on the initial load of the NREF data with only primary keys applied. Then again after the implementation of the recommendations provided by the advisor tools. As a reference, the tests

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

were executed a third time on the initial load but with a predefined set of 33 indexes proposed by [CBTM05]. Every DBMS was left at its default settings. After a default installation there were no changes being made to the configuration other than creating a database and filling it with the NREF data. Not only because of that the results of these tests can not be used for a comparison between the various DBMS but only for indicating a relative change resulting from the advisor tools. Therefore, there will be no confrontation of results at the end of the chapter.

3.2.2. Oracle

Founded by Lawrence Ellison, Robert Miner and Edward Oates in 1977, Software Development Laboratories, later known as Relational Software, Inc. and then as Oracle Corporation, released its first official version of Oracle called Oracle V2 in 1979. Oracle 10g, released in 2003, introduced a new feature set for workload monitoring to perform automatic and semi-automatic changes to the database to improve the performance.

Monitoring and Data Collection

Based on the Automatic Workload Repository (AWR) which is used to collect statistics of the current workload, Oracle 10g and up are able to monitor the system in the background by recording data such as the usage of database objects, connection and scheduling statistics to identify time expensive tasks. The Automatic Database Diagnostic Monitor (ADDM) uses that data to make snapshots of the system every hour keeping them over a period of seven days to enable analyses of the workload in different time frames. Two different snapshots can be compared to identify changes and trends. This can be used to find causes of possible performance bottlenecks by analysing, for example, CPU and memory load together with the top n most expensive SQL statements.

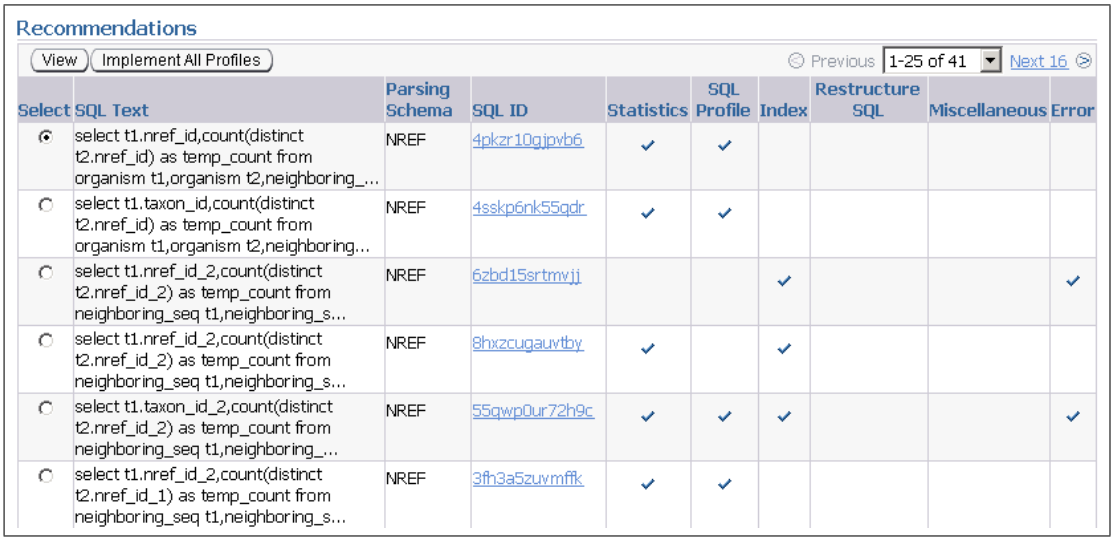
Analysis

The two tuning tools that were used to test Oracle's tuning capabilities are the SQL Tuning Advisor and the SQL Access Advisor. Both analyse a given workload coming from either AWR/ADDM or from an SQL Tuning Set (STS) that represents a static group of statements. The advisors can be used directly from the SQL command prompt but Oracle also provides easy-to-use wizards included in its Oracle Enterprise Management (OEM) webinterface.
To be able to compare the effectiveness of the advisors the NREF queries were executed before optimisation as described in section 3.2.1 resulting in an average execution time of 13924 seconds on an Oracle 11g database.

SQL Tuning Advisor

The Oracle SQL Tuning Advisor takes one or more SQL statements from an STS, from a time period between two ADDM snapshots or from current AWR data, being the last n executed queries that are still in the cache. Recommendations include creation of statistics and indexes, restructuring the SQL statement and creation of SQL Profiles. The SQL Tuning Advisor only

looks at one statement at a time without considering the context and workload the query is being executed in.
To test the SQL Tuning Advisor an SQL Tuning Set was created with the 50 NREF queries. The Tuning Advisor can either be run in limited (no SQL Profiles being created) or comprehensive mode (SQL Profiles included). The time limit for optimisation was set to 15000 seconds and a five minute limit per query but the task was already completed after about 6300 seconds.



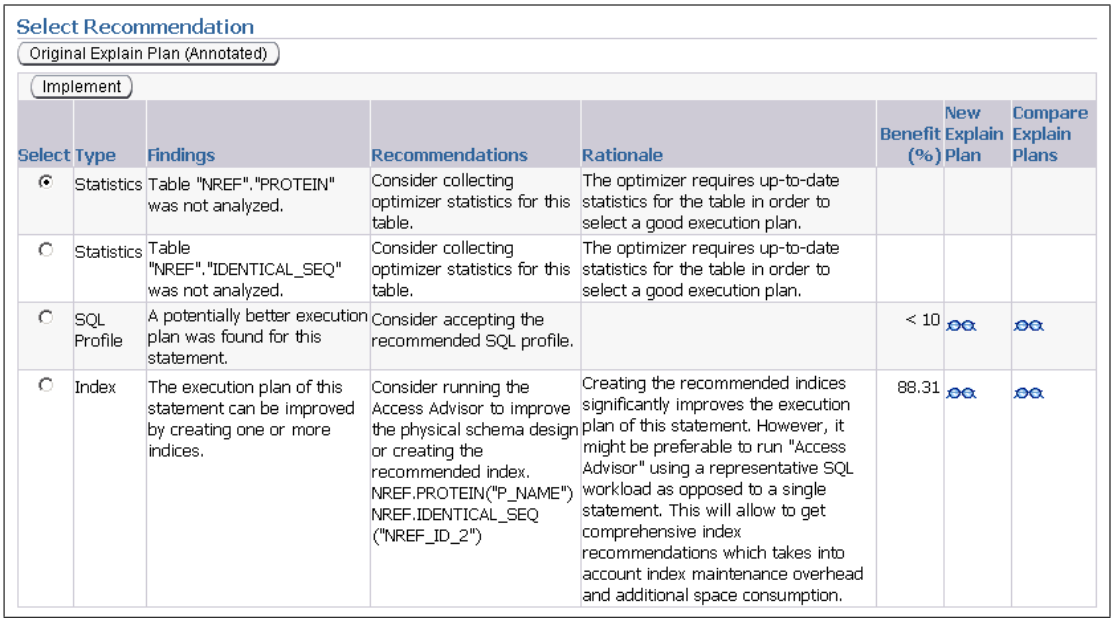
Recommendations

View | Implement All Profiles | Previous 1-25 of 41 | Next 16

Select	SQL Text	Parsing Schema	SQL ID	Statistics	SQL Profile	Index	Restructure SQL	Miscellaneous	Error
◉	select t1.nref_id,count(distinct t2.nref_id) as temp_count from organism t1,organism t2,neighboring_...	NREF	4pkzr10gjpvb6	✓	✓				
○	select t1.taxon_id,count(distinct t2.nref_id) as temp_count from organism t1,organism t2,neighboring...	NREF	4sskp6nk55qdr	✓	✓				
○	select t1.nref_id_2,count(distinct t2.nref_id_2) as temp_count from neighboring_seq t1,neighboring_s...	NREF	6zbd15srtmvjj			✓			✓
○	select t1.nref_id_2,count(distinct t2.nref_id_2) as temp_count from neighboring_seq t1,neighboring_s...	NREF	8hxzcugauvtby	✓		✓			
○	select t1.taxon_id_2,count(distinct t2.nref_id_2) as temp_count from neighboring_seq t1,neighboring_...	NREF	55qwp0ur72h9c	✓	✓	✓			✓
○	select t1.nref_id_2,count(distinct t2.nref_id_1) as temp_count from neighboring_seq t1,neighboring_s...	NREF	3fh3a5zuvmffk	✓	✓				

Figure 3.1: Oracle SQL Tuning Advisor Recommendations

Figure 3.1 shows a part of the recommendations that were created on the given STS. The user is presented with the list of queries together with possible changes that may help to improve the performance. In total, the Oracle SQL Tuning Advisor recommended the creation of nine distinct indexes, 19 SQL Profiles and statistics on all tables.



Select Recommendation

Original Explain Plan (Annotated)

Implement

Select	Type	Findings	Recommendations	Rationale	Benefit (%)	New Explain Plan	Compare Explain Plans
◉	Statistics	Table "NREF"."PROTEIN" was not analyzed.	Consider collecting optimizer statistics for this table.	The optimizer requires up-to-date statistics for the table in order to select a good execution plan.			
○	Statistics	Table "NREF"."IDENTICAL_SEQ" was not analyzed.	Consider collecting optimizer statistics for this table.	The optimizer requires up-to-date statistics for the table in order to select a good execution plan.			
○	SQL Profile	A potentially better execution plan was found for this statement.	Consider accepting the recommended SQL profile.		< 10	oo	oo
○	Index	The execution plan of this statement can be improved by creating one or more indices.	Consider running the Access Advisor to improve the physical schema design or creating the recommended index. NREF.PROTEIN("P_NAME") NREF.IDENTICAL_SEQ ("NREF_ID_2")	Creating the recommended indices significantly improves the execution plan of this statement. However, it might be preferable to run "Access Advisor" using a representative SQL workload as opposed to a single statement. This will allow to get comprehensive index recommendations which takes into account index maintenance overhead and additional space consumption.	88.31	oo	oo

Figure 3.2: Oracle SQL Tuning Advisor Details

Figure 3.2 exemplarily shows the recommendation details for one of the NREF queries. The user can see the estimated benefits for each recommendation together with a rationale of the suggested change. It is even possible to compare the query execution plans before and after the implementation of the change. In this case, the advisor expects the creation of an index to improve performance of this single query by over 88%.
After implementing all of the nine recommended indexes, six of the SQL Profiles and creating statistics on the six NREF tables the size of the database has grown from 17 to over 28 gigabytes. The tests were repeated resulting in an average execution time of 7263 seconds which is a performance win of about 48%.
As the SQL Tuning Advisor is designed to analyse only one query at a time the user has to select every single recommendation to decide what he wants to implement. With a large workload this becomes very unhandy and the user risks losing overview on what indexes and statistics already have been created.

SQL Access Advisor

The Oracle SQL Access Advisor is a second tool to scan a workload to get recommendations about how to improve the performance. Other than the SQL Tuning Advisor the Access Advisor tries to get a more global view on the database and does not only focus on a single statement. The advisor recommends the creation of indexes, partitions, materialised views and materialised view logs. A materialised view log is used by Oracle to incrementally update a materialised view after the master table has changed instead of re-creating the view from scratch.

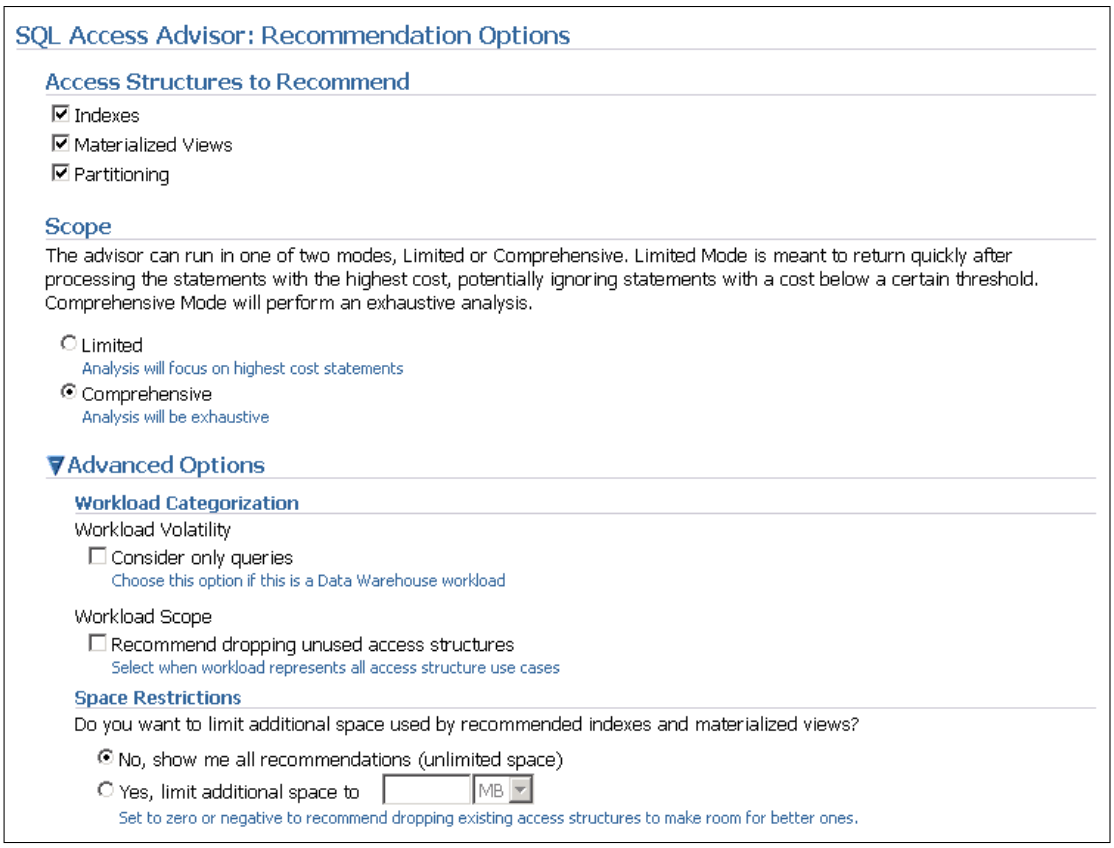


Figure 3.3: Oracle SQL Access Advisor Options

As shown in figure 3.3, the SQL Access Advisor wizard offers a number of options to choose from. For example, it is possible to provide space restrictions to limit the disk space used by new database objects or to get a list of unused objects that may be dropped from the database. The time limit for the optimisation process was defaulted to 10000 minutes giving the advisor nearly seven days time to analyse the workload. However, even after several tests the advisor completed its work after approximately 20 seconds.

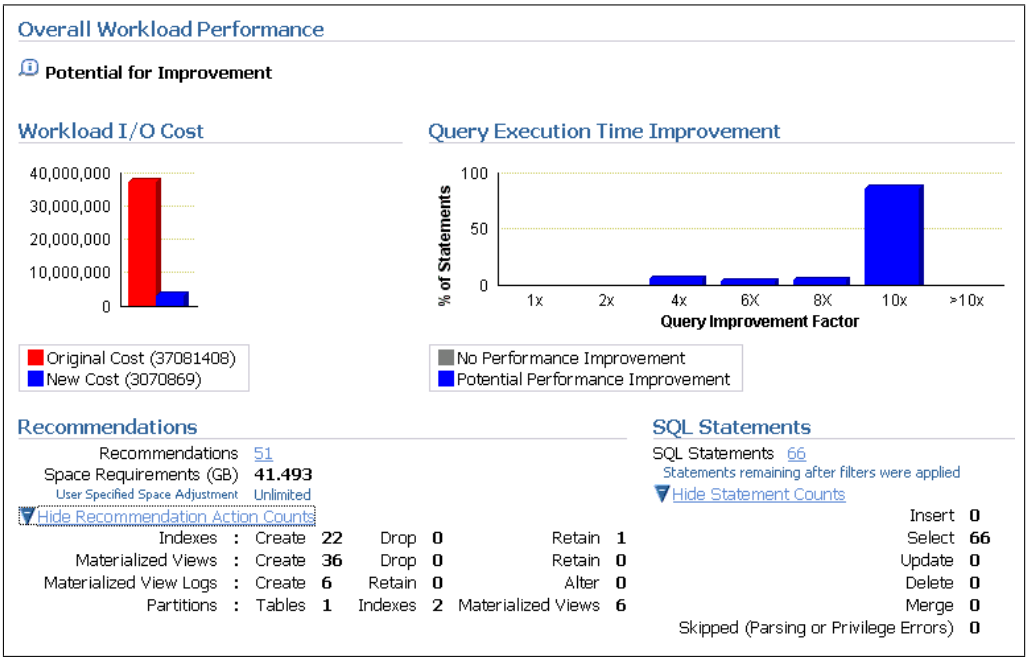


Figure 3.4: Oracle SQL Access Advisor Results

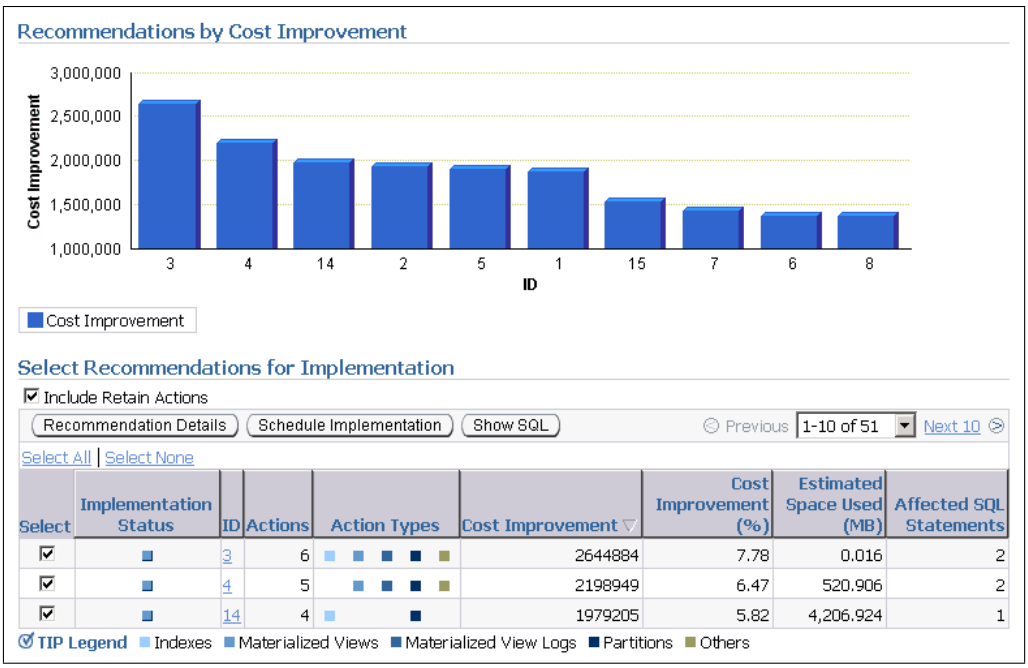


Figure 3.5: Oracle SQL Access Advisor Recommendations

Figures 3.4 and 3.5 show the results of the SQL Access Advisor run. The user is presented with the estimated performance improvements together with a detailed breakdown of each suggested change. On the given workload the advisor recommended a total of 13 indexes, 13 materialised views, six materialised view logs and one partition that were all implemented and again enlarged the database to about 28 gigabytes. The resulting execution time was cut down to an average of 1721 seconds - a performance win of 88%.

Results

Run #	**Non-optimised**	**Optimised** (SQL Tuning Advisor)	**Optimised** (SQL Access Advisor)	**Reference**
1	14149 s	7353 s	1727 s	1489 s
2	13519 s	7219 s	1717 s	1531 s
3	14105 s	7217 s	1721 s	1485 s
Average	**13924 s**	**7263 s**	**1721 s**	**1501 s**

Figure 3.6: Oracle Advisor Results

Figure 3.6 gives an overview of the results observed in this experiment. Although the concept of SQL Profiles with statement-specific statistics promised to be a good way of improving the performance of queries it seems that the approach of the SQL Access Advisor leads to better results by trying to find a set of recommendations for the whole workload and including materialised views and partitions.
Both advisor tools can be controlled from within the Oracle Enterprise Management interface so that the user doesn't need to handle the complex SQL that needs to be created to start the tools. The results are presented in great detail and the user can decide to see either only a summary or all information even down to a comparison of query execution plans. Even with materialised views and partitions the 33 reference indexes couldn't be beaten, however, creating those enlarged the database from 17 to over 36 gigabytes on the disc taking nearly 10 gigabytes more than the recommendations of the Access Advisor which led to an only slightly longer execution time.

Other Tools

Oracle offers a number of additional tools that either provide the user with recommendations or automatically perform changes in the background. The Automatic Optimizer Statistics Collection is a DBMS task that is scheduled to automatically gather statistics on tables with no or outdated statistics. The Automatic Shared Memory Management takes care of memory allocation. There are other advisor tools such as the Segment Advisor which analyses the fragmentation of database files or the Undo Advisor that gives recommendations about Oracle's undo system. Detailed information about the various possibilities in the current version of Oracle can be found at [Corb].

3.2.3. MS SQL Server

The first release of the Microsoft SQL Server is dated back in 1989 with version 1.0 for OS/2 and the first 32-Bit Windows NT based release followed in 1993. Until 1994, Microsoft was partnering with Sybase keeping more or less the same code line for Unix and VMS on the one side and Windows and OS/2 on the other. Version 6 of the SQL Server was then released in 1995 without work of Sybase. In 1999, version 7 introduced tools based on the AutoAdmin project such as the MS SQL Server Profiler for monitoring the database, the Query Analyzer which is used for performance analyses of queries and the Index Tuning Wizard which recommends a set of indexes for a given workload. With the SQL Server 2005 the Index Tuning Advisor was replaced by the Database Engine Tuning Advisor (DTA) which now also recommends to drop unused indexes as well as the creation of partitions and of materialised views.

Monitoring and Data Collection

The MS SQL Server Profiler monitors all activities on a database and records the workload. However, other than the Automatic Workload Repository in Oracle, the Server Profiler needs to be started and controlled manually. While running the tests the profiler recorded a trace file containing all statements that were executed.

Analysis and Presentation

The resulting trace file can then be used with the Database Engine Tuning Advisor.

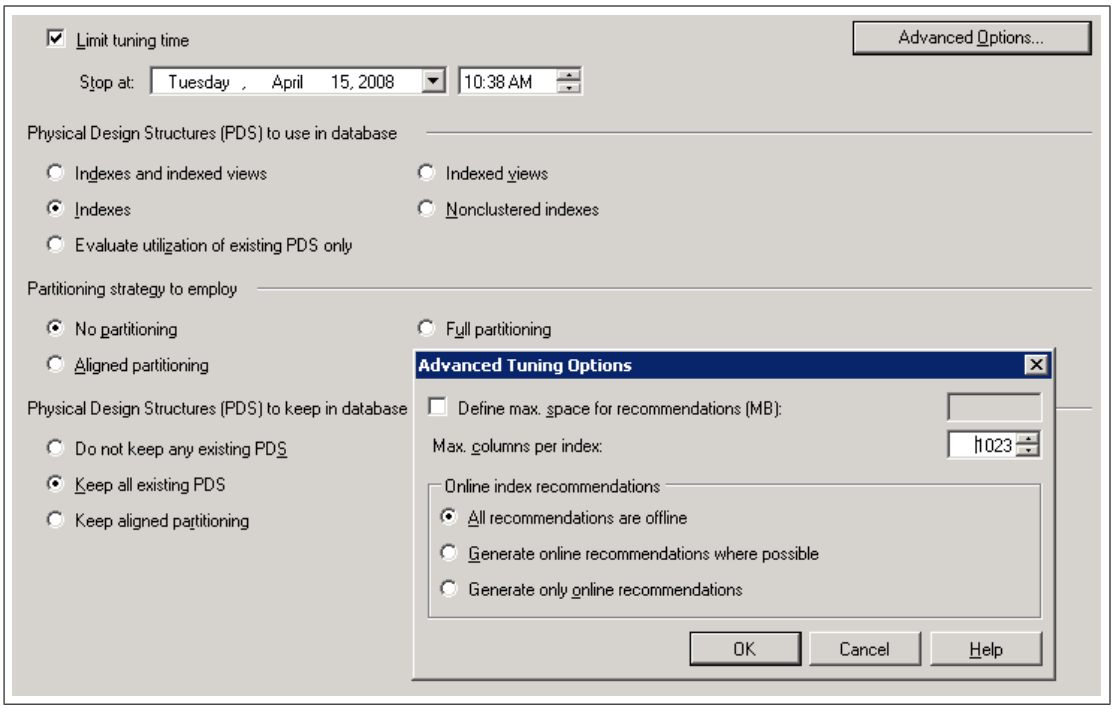


Figure 3.7: MS SQL Server DTA Options

Figure 3.7 shows the options available for DTA. As with Oracle's advisor tools, the user can decide what objects will be recommended and he can set a limit to the disk space required by those objects.

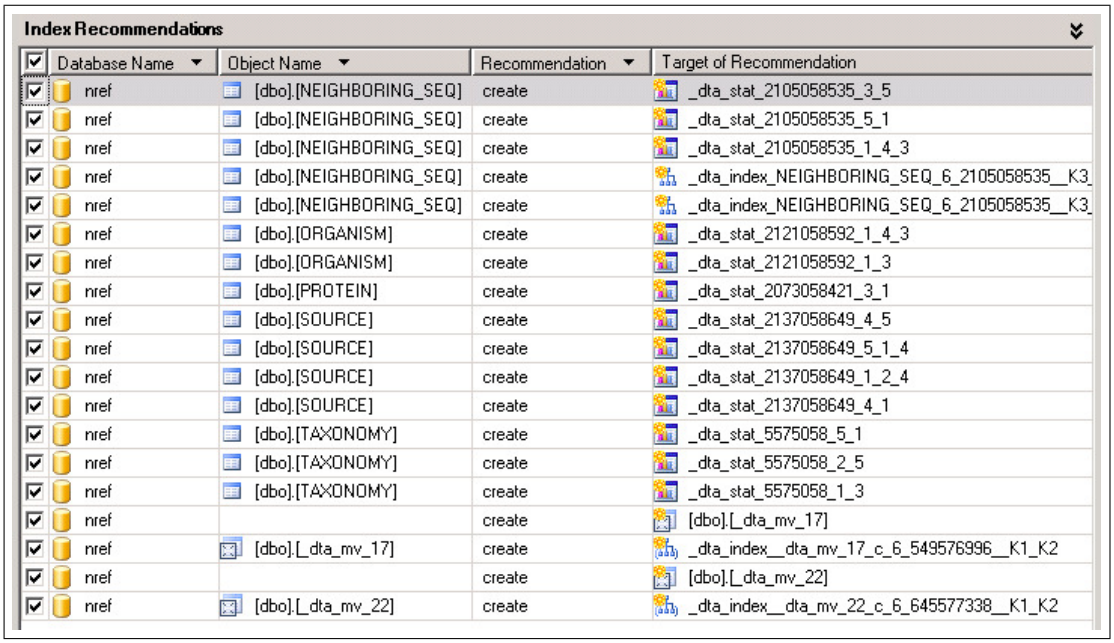


Figure 3.8: MS SQL Server DTA Recommendations

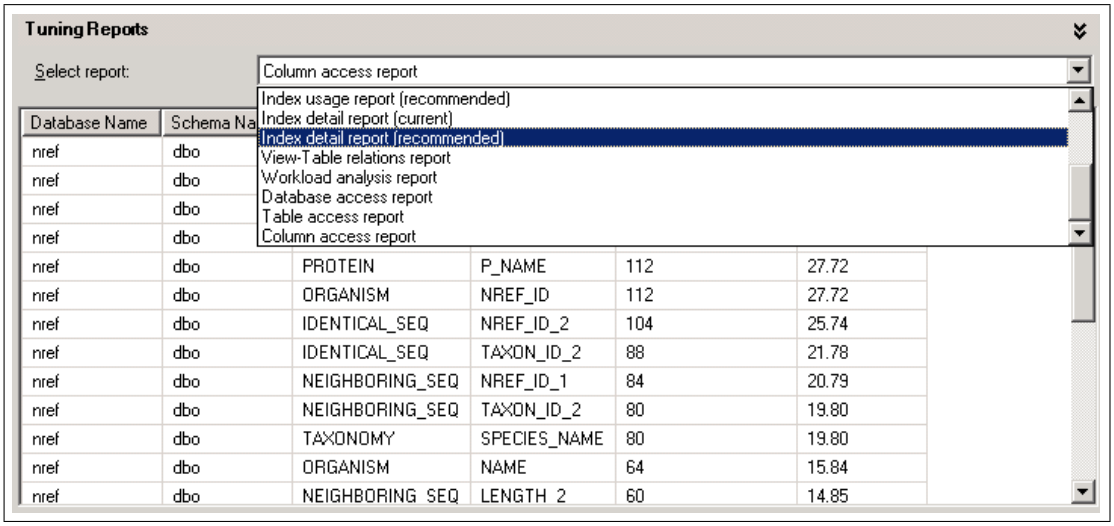


Figure 3.9: MS SQL Server DTA Reports

Although giving a time limit of one hour, the advisor was already done after about eight minutes. In figures 3.8 and 3.9 the results of the advisor run can be seen. DTA offers a number of detailed reports on the optimisation that should help the DBA to decide which recommendations to implement if he doesn't want to apply them all.
For the test all recommended changes were applied. MS SQL Server created four indexes, 13 statistics, two materialised views and one partition enlarging the size of the database from about 13 to 17 gigabytes.

Results

In figure 3.10, the results of the MS SQL Server tests can be seen. On an unoptimised MS SQL Server 2005 database the NREF query set took an average of 12102 seconds to complete. DTA was able to cut down the average execution time by 60% by creating indexes, materialised views and partitions. As with Oracle, the 33 reference indexes doubled the size of the database and could speed up execution only a few percent over the advisor results. The advisor is being controlled over a simple, self-explaining graphical interface but there seems to be no option to go into a more detailed view as it was with Oracle where the user was even able to compare execution plans.

Run #	**Non-optimised**	**Optimised**	**Reference**
1	12215 s	4867 s	4089 s
2	12207 s	4757 s	4036 s
3	11884 s	4785 s	4042 s
Average	**12102 s**	**4803 s**	**4055 s**

Figure 3.10: MS SQL Server Advisor Results

Other Tools

Besides the Database Engine Tuning Advisor there are no other advisory tools integrated in MS SQL Server version 2005 but in the new version 2008 Microsoft included more self-tuning and advisory features to support the DBA such as a best practice design alerter that can help to avoid typical design issues for common usage patterns. More information about performance tuning in MS SQL Server 2005 can be found in [WIG+06].

3.2.4. IBM DB2

Developed for IBM's MVS mainframe in the early 1980s, DB2 is directly based on the 1970's research project System R which tried to implement Codd's ideas of a relational model for databases. The first version of DB2 was released in 1983 and during the 1990s IBM ported the DBMS to other non-mainframe platforms. In version 6.1, IBM added the DB2 Advisor which was capable of recommending multi-column indexes. In version 8.2, this was replaced by the DB2 Design Advisor which can also recommend materialised views, partitions and multidimensional clustering tables which are mostly used for data warehousing.

Workload Obtaining

The Design Advisor can be invoked with a single statement or with a set of statements in a file or a database table. This allows analysis of a static and maybe theoretical workload. To work with the actual workload the advisor can be sourced with the DB2 statement cache or the Query Patroller that comes with the data warehousing edition of DB2. When working with a

large amount of queries in the workload the Design Advisor can compress it by merging similar queries to reduce the time required for the analysis.

Analysis

DB2 again allows the user to set a disk space limit for the new database objects that will be recommended as well as the kind of objects to recommend as seen in figure 3.11. The DB2 9.5 test installation had problems with the NREF query set. One query took an extreme of more than 72 hours to complete, hence it was removed from the set. The remaining queries took an average of 9233 seconds. The analyser results can be seen in figure 3.12. On the NREF workload the Design Advisor recommended 11 indexes but no materialised views or multidimensional clustering tables. Creating those indexes added about 1.5 gigabytes to the database enlarging it from 8.2 to 9.7 gigabytes on the disk.

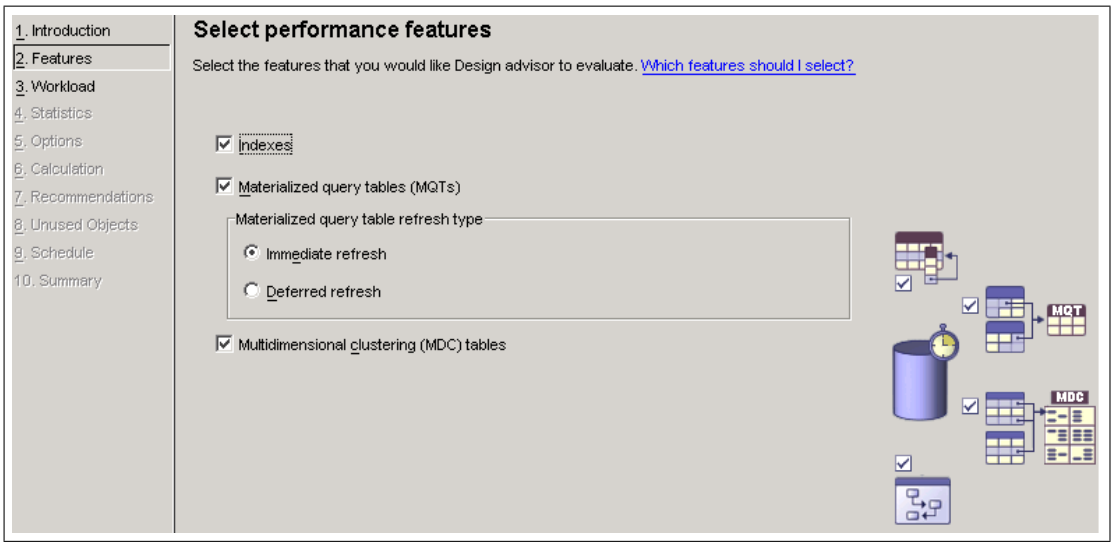


Figure 3.11: IBM DB2 Design Advisor

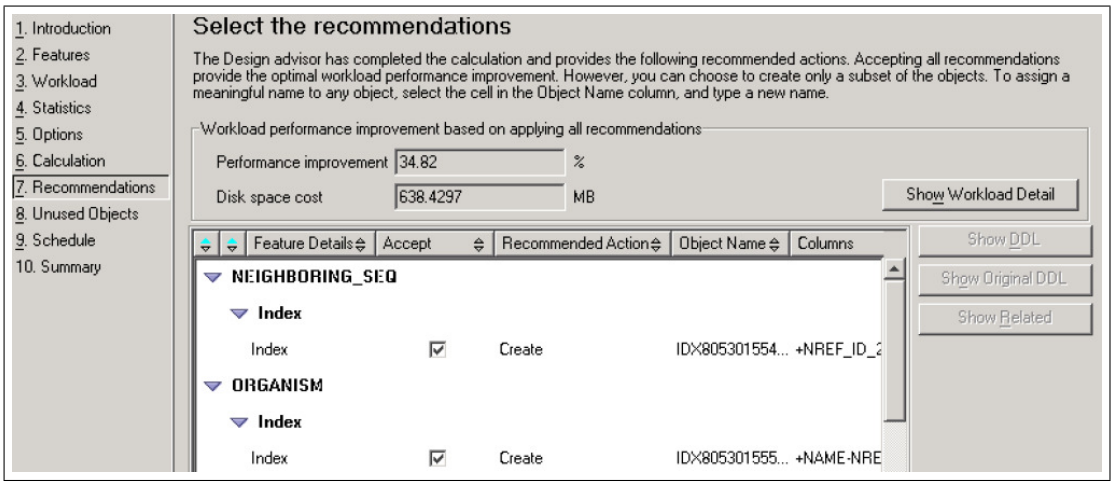


Figure 3.12: IBM DB2 Advisor Recommendations

Results

The DB2 Design Advisor could cut down the execution time to 6324 seconds. As with MS SQL Server and Oracle the recommendations of the Design Advisor couldn't beat the set of reference indexes however the 33 indexes doubled the size of the database to about 17 gigabytes while performing only slightly better than the recommended smaller index set.

Run #	**Non-optimised**	**Optimised**	**Reference**
1	9241 s	6360 s	5590 s
2	9135 s	6242 s	5078 s
3	9325 s	6370 s	5594 s
Average	**9233 s**	**6324 s**	**5420 s**

Figure 3.13: IBM DB2 Design Advisor Results

Other Tools

DB2 also includes a Configuration Advisor which can recommend changes to DBMS and database settings for a specific usage scenario. As seen in figure 3.14, the user is guided through a wizard describing the usage pattern in terms of the ratio between selects and inserts, the number of applications and users on the database or the required isolation level. The advisor then recommends cache settings, buffer sizes, settings for parallelism, etc.

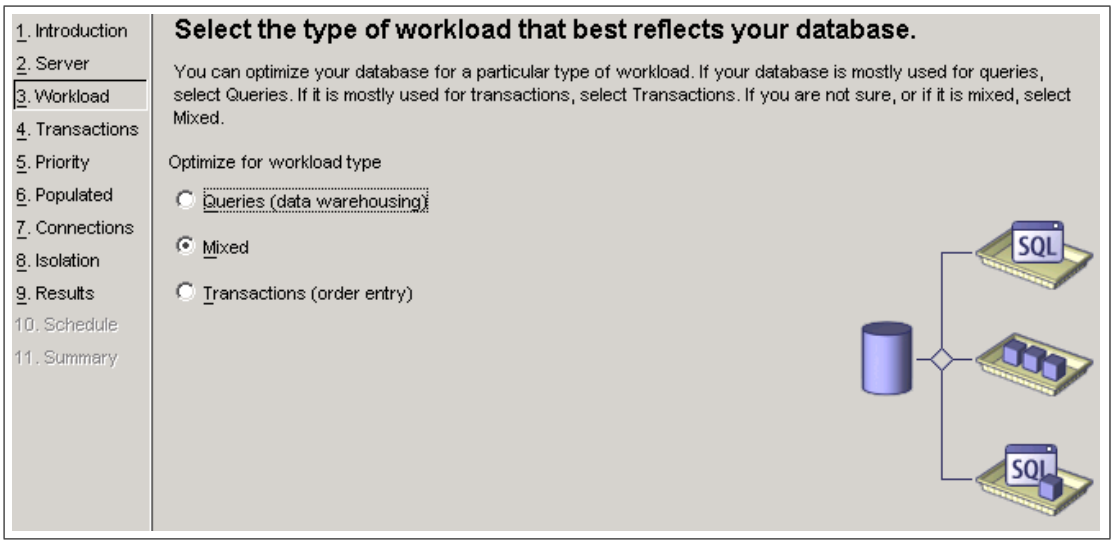


Figure 3.14: IBM DB2 Configuration Advisor

With the DB2 Performance Expert IBM offers another tool to monitor the DBMS. It includes a large number of data points from throughout the system that can be monitored in realtime or recorded for long-term analysis. The Performance Expert also allows the definition of exceptions to alert the DBA in problematic situations.

4. System Concept

This chapter describes the proposed design analyser in detail. The first section will cover the basic decisions that were made to design the concept. The second section will then give an overview of the system architecture followed by descriptions of the three work phases. Chapter 5 will then cover the implementation of the system.

4.1. Design Decisions

The system was designed as a three-layered architecture where all layers are physically independent of each other and only communicate indirectly over SQL. This makes it possible to easily change or even replace one layer without breaking the rest of the system. The three layers correspond to the three steps of monitoring, storing and analysing data as described in section 2.2.4.
For the monitoring the approach of an online workload recording was chosen to allow the analysis of the most current data. Every statement is logged automatically as soon as the DBMS starts operating. The data collection that is persistently storing the workload data over a longer time period was designed to use a delayed writing mechanism keeping the overhead of disk accesses as low as possible. For the analysis a rules-based approach was chosen with a predefined set of rules that map specific data constellations to possible problems and solutions. The *What-If* concept of hypothetical physical structures is used to feed the internal DBMS optimiser with virtual objects to find optimal execution plans without materialising the new structures. This way, the internal cost model of the DBMS can be utilised making sure that recommended changes will actually be used by the optimiser later on. Results of the analysis are shown as textual and graphic reports and the DBA still needs to implement changes by himself. However, an active alerting mechanism already allows automatic reaction on user-defined events.

4.2. Architecture

Following the concept of the tuning control loop, figure 4.1 shows the order of events for monitoring, analysing and tuning the DBMS. (1) While running in normal state the user interacts with the DBMS as always – his work is not being influenced by the monitor as all operations have no or very little overhead. (2) When running a query the optimiser is utilised to collect realtime data about the statement such as the time it takes to find a QEP, which tables and columns are being accessed and if statistics are available or not. This short-term data is put into an internal DBMS structure that resides in main memory for the lifetime of the DBMS process. No additional disk access is required. The structure is attached as a managed object to IMA and is therefore available through the IMADB. (3) The monitor which is running as a daemon

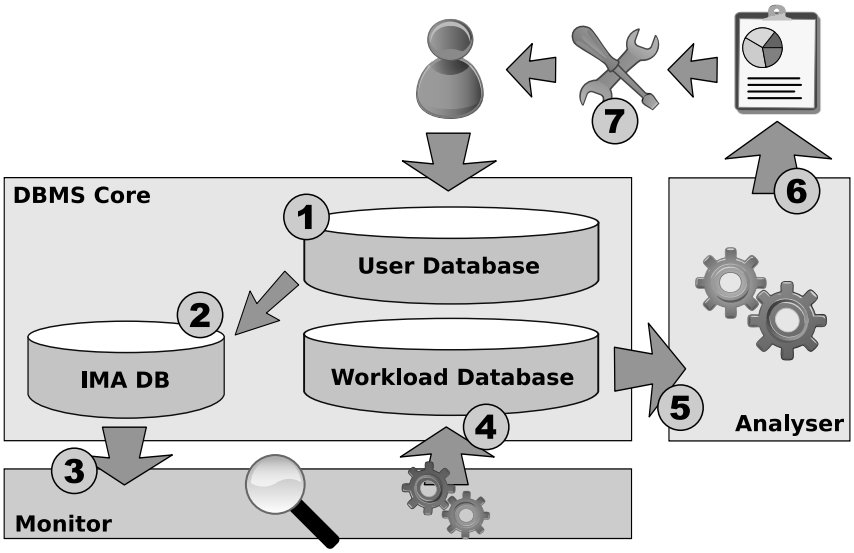


Figure 4.1: System Architecture

process is periodically waking up and querying IMADB to read the collected data. After a given number of polls on IMADB the monitor performs a first aggregation and condensation of the raw data (4) and the result is appended to a workload database which acts as a persistent storage for statistical data. As this only happens once every given timespan, this delayed way of storing the data on disk is believed to have only a very small impact on the DBMS load with no noticeable influence on the user database. During normal operation of the DBMS the workload database is filled continuously up to a predefined maximum and then wraps around to form a moving window of statistical data. The workload database is a standard Ingres database and can therefore be expanded with triggers and procedures which allow the implementation of alerting mechanisms that can react when an attribute drops below or exceeds a certain threshold. (5) The DBA then decides, maybe after receiving an alert, to run the design analyser. This step is recommended when the database that will be analysed is not in use as the tool runs tests to confirm the benefit of new database objects. The analyser starts with scanning the workload to find problems and to identify the need of indexes, table statistics and more. (6) The resulting recommendations are then presented to the DBA. (7) For all recommended changes to database objects such as indexes the analyser provides SQL statements to implement them and the DBA can decide to use them all or only a subset of them. After that, the DBMS can continue with normal operation and monitoring.

As both monitor and analyser use SQL to communicate with the DBMS, it is possible to run them on remote machines – even the workload database can reside on a separate Ingres instance. However, for the saved resource capacities on the DBMS machine there will be additional network traffic added which may be more expensive than the system load.

In the next three sections the three work phases of the system are being described in greater detail.

4.3. Monitor

The goal of the monitoring phase is to collect as much as possible data during the day-to-day usage of the DBMS with very little or no impact on the system's load, response time and performance. Because of that, the DBMS core will only be used to collect data that is known anyway throughout processing a statement without adding too many new features that can cause a decrease in performance. As mentioned in section 2.1.5, IMA already contains a huge amount of attributes from all of Ingres' server facilities that can serve as indicators for performance tuning. This section describes the order of events for the monitoring phase as shown in figure 4.2.

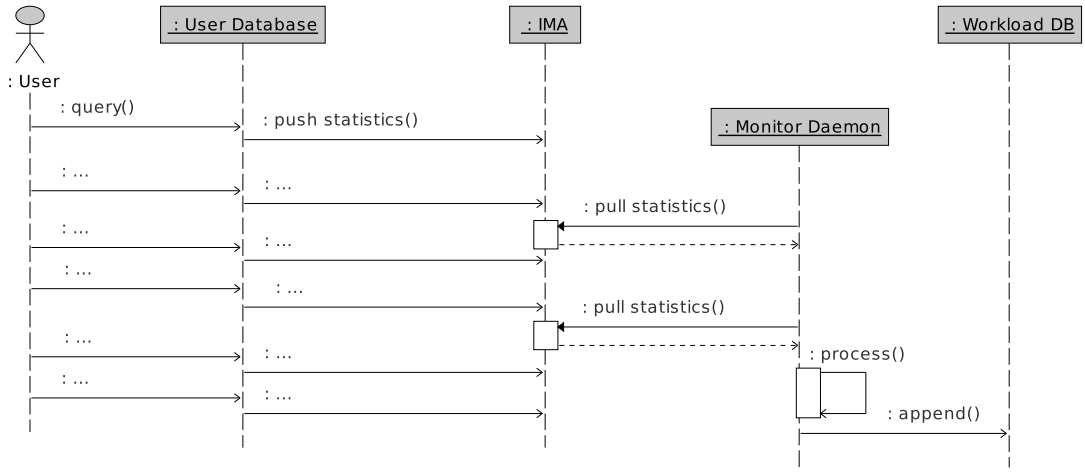


Figure 4.2: Sequence Diagram of the Monitor Phase

4.3.1. Preconditions and Data Collection

The operation of the system begins when the DBMS is started – from that point every query on a user database will cause data to be logged in IMA. The collected data can be separated into three main categories: I) Catalogue information, II) workload information and III) system statistics. The first includes information that can be found in the system catalogues about the database schema such as the tables and their attributes. The workload information contains the actual workload on the database with the statements that were executed. The system statistics are a collection of DBMS-wide data points that can serve as indicators for resource problems or bad configuration of the system.
The different data points of all categories are as follows:

Catalogue Information

This information is read from the system catalogues. These values change slowly over time when database objects are altered or the content of tables significantly changes. The information about database objects is needed to give accurate recommendations for physical design changes.

- Tables
  - Storage structure, main data pages and overflow pages
    As described in section 2.2.2, the two storage structure types hash and ISAM use overflow pages when the main data pages are full. More pages need to be touched to find the requested data and the performance drops. The number of main and overflow pages per table can be used as an indicator for tables that should be considered for restructuring.
  - Actual and estimated cost and cardinality
    As described in section 2.2.3, Ingres provides the estimated and actual values for CPU and disk I/O cost and the number of tuples per node in a QEP. The values for table nodes are stored and can be used as an indicator for outdated or missing table statistics.
- Attributes
  - Statistics
    The Ingres optimiser explicitly requests statistics from the system catalogues. When no statistics are available a uniform data distribution is assumed. This event is being logged to create a list of attributes that have no statistics.
- Indexes
  - Storage structure, main data pages and overflow pages
    Because indexes in Ingres are also stored as tables the same information can be collected here.

Workload Information

This information is read from the parser, the optimiser and the execution facility while queries are processed. Values here are only valid for the time the statement is being processed and may change each time the same statement is executed. Information about the workload on the database can be logged over time to record the usage pattern of the system.

- Statements
  - Statement text
    Each statement is logged as plain text to allow later identification when presenting the analyser results.
  - Cost values
    CPU and disk I/O cost values are logged for the time the statement is being optimised and the time it is executed. For complex selects with joins over many tables the enumeration phase where the optimiser searches for the cheapest QEP can take as long as the actual execution of the query. Logging the time in OPF helps to find these cases. OPF and QEF cost together can be used to identify the most expensive statements.

  - Cost estimates
    The CPU and disk I/O cost estimates as returned by the optmiser help to find statements where actual and estimated costs don't match which may be caused by missing or outdated statistics.

- References

  - Tables, Attributes, Indexes
    For each statement that is being executed the list of database objects used in the statement is logged.
  - Frequency
    For each database object the number of uses in the workload is logged. This allows prioritisation of objects during the analysis. In addition, with a list of used indexes the system can find out which indexes are not needed and therefore should be considered for removal.

System Statistics

A large amount of system statistics from throughout the DBMS is offered over IMA. Most of these values constantly change and can be used to identify and – to a certain degree – also to predict problems with system resources. The observation of values over time can reveal changing demands or anomalies in the system usage.

- Variable values

  - Runtime statistics
    Values such as the number of users that are currently on the system or the current number of locks in use can be collected to perform trend analyses and to identify changing demands on the system.

- Static values

  - System settings
    Configuration settings such as the maximum number of connections on the DBMS or the maximum number of locks per transaction allow identification of misconfiguration and anomalies in the runtime statistics.

All those data points are made available in IMA so that the monitor can pull them easily from the DBMS to process them and store them in the workload DB. Figure 4.3 shows the table schemes in IMADB that contain the collected data.
The *Workload* table contains the actual workload history with information of when was which query executed and what cost values are associated. With that, a complete history of the database usage can be created. The *References* table is the connection between a statement and the database objects the statement uses. With this table the system can see what tables are in the FROM clause of the statement, which indexes OPF has chosen and what table attributes were used.

“Statements”	
Column	**Description**
Database	The name of the user database
Hash	The hash value of the query text
Query text	The statement in plain text
Frequency	A counter of how often the query has been executed
Time	A timestamp when the query was last executed

“Workload”	
Column	**Description**
Database	The name of the user database
Hash	The hash value of the query text
OPF CPU	The cpu time the query spends in OPF
OPF I/O	The disk I/O needed in OPF
QEF CPU	The cpu time the query spends in QEF
QEF I/O	The disk I/O needed in QEF
Estimated CPU	The estimated overall cpu time
Estimated I/O	The estimated overall disk I/O
Pages touched	The estimated number of touched data pages
Time	A timestamp when this query was executed
Wallclock time	The time the query took to execute

“Tables”	
Column	**Description**
Database	The name of the user database
ID	The internal ID of the table
Name	The name of the table
Frequency	A counter of how often the table has been referenced
Structure	The storage structure of the table
Data	The number of data pages
Overflow	The number of overflow pages
Time	A timestamp when the table was last seen

“Attributes”	
Column	**Description**
Database	The name of the user database
ID	The internal ID of the attribute
Table ID	The ID of the table the attribute is in
Name	The name of the attribute
Frequency	A counter of how often the attribute has been referenced
Statistics	A boolean that is true when statistics exist
Time	A timestamp when the attribute was last seen

“Indexes”	
Column	**Description**
Database	The name of the user database
ID	The internal ID of the index
Name	The name of the index
Table ID	The ID of the table the index is created on
Attribute ID	The ID of the attribute the index is created on
Frequency	A counter of how often the index has been used
Structure	The storage structure of the table
Data	The number of data pages
Overflow	The number of overflow pages
Time	A timestamp when the index was last seen

“References”	
Column	**Description**
Database	The name of the user database
Hash	The hash key of the statement
Type	The type of the referenced object
Object ID	The table, attribute or index ID of the referenced object
Table ID	The table ID needed to find the correct attribute
Time	A timestamp when this reference was seen

“Statistics”	
Column	**Description**
Connections	The number of connected users on the system
Sessions	The number of active sessions in the system
Maximum sessions	The maximum number of allowed sessions
Total rows	The total number of rows returned since the DBMS is running
Selects processed	The total number of selects since the DBMS is running
Locks per transaction	The maximum number of locks being hold per transaction
Maximum locks	The maximum number of locks in total
Locks used	The number of locks being currently used
Deadlocks	The number of deadlocks that occurred
Escalated locks	The number of locks that need to be escalated to table locks
Lock waits	The number of pending locks that are blocked by other locks

Figure 4.3: IMADB Table Schemes

Because IMA objects reside in main memory there is no additional disk access required to store those IMA tables, however, to avoid constantly growing memory allocation the tables need to be implemented as ring buffers. It is believed that because of the finite number of database objects the *Tables*, *Attributes* and *Indexes* tables may not have to be limited in size, however, the *Statements*, *Workload*, *References* and *Statistics* tables may become very big over time. Therefore, a maximum number of rows for all tables will be defined after which the content wraps around and new entries overwrite the oldest entries.

4.3.2. Invocation of the Monitor

While the collection of data is started as soon as the DBMS is running and processing queries, the monitor needs to be invoked by the DBA because even when it is designed to be a lightweight application it may still produce unwanted load on a system. The monitor is implemented as a daemon that is started with the target database as its argument. It periodically wakes up and queries IMADB to get the newest workload statistics of the target database. Because of the volatility of the data in IMA the time between wake-ups must be short enough to capture changes in a detailed data resolution – but to reduce overhead on the DBMS and to allow time to process the data from IMA the period must not be too short. An interval of 30 seconds is believed to be enough to get a proper data resolution without stressing IMADB too much.

4.3.3. Processing and Storing the Data

The monitor daemon polls IMADB for a given number of times to get the most current data. The default was chosen to be ten times resulting in a five minute time window of IMA data to process. The data from the *Statistics* table is processed depending on the meaning of the value. Some values are averaged, of some other the maximum will be taken. This will lower the amount of data without losing too much accuracy. The monitor then appends the collected and processed IMA data to the workload database.
The workload database is a native Ingres database that contains the same schema as the one used in IMADB. Rows in the *Statements*, *Tables*, *Attributes* and *Indexes* tables are updated when the timestamp in IMADB is greater than the one in the workload database. Updates on the *Workload* and *Statistics* tables will not overwrite existing entries and will be appended instead – this allows trend analysis over the timespan of the monitored data. As with the IMADB, the amount of data in the workload DB needs to be limited as the tables would infinitely grow. Instead of a defined maximum of rows as for the IMA tables, a time window approach was chosen for the workload DB. All entries are kept for seven days if their timestamps are not updated during that time.
Because the workload DB is in fact a user database it is possible to create triggers and procedures to automatically react on changes such as exceeding a given threshold for a chosen data point. With that, the monitor daemon also 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.

4.3.4. Postconditions

The monitor daemon will run until the DBA sends a kill signal to stop its work. Because of the small overhead, the monitor can keep running without having too much impact on the system and therefore can collect a large amount of data that can be used for the analyser phase. The monitor should be at least run long enough to record a full cycle of the typical workload on the user database so that the workload DB contains a complete list of executed queries together with statistical data from within that timeframe.

4.4. Analyse

In the analyser phase the collected data in the workload DB will be processed. This ranges from simple reporting of aggregated data to the automatic identification of performance problems with recommendations to solve them as described in section 2.2.4. In a first step, the data will be scanned based on a set of rules that are explained further below. In a second step, the information gathered from the workload will be used to create a list of changes that can help to improve performance. As a third step, some of the proposed changes are tested on the user database to see if they have a positive effect. In the last step, the analyser will create a final list of recommendations that will be shown to the DBA.
This section describes the order of events for the analyse phase as shown in figure 4.4.

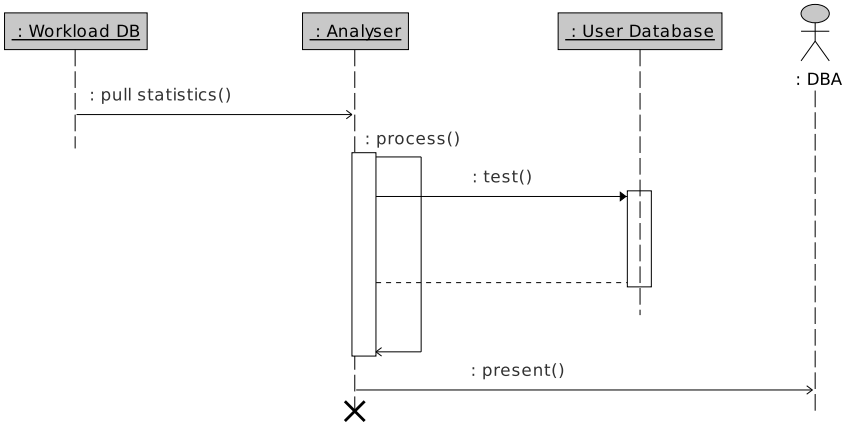


Figure 4.4: Sequence Diagram of the Analyse Phase

4.4.1. Preconditions

The analyser depends on the content of the workload DB. Therefore, as a precondition, the workload DB must have been filled by the monitor with enough data to cover the workload that should be analysed. As part of the tests the analyser will perform, virtual indexes are created to see if they would have a positive effect on the performance. The DBMS will prevent that those virtual objects are used for a user query but they still require an exclusive lock on the table which may interfer with other traffic on the database. Because of this, it is highly recommended that the database is not in use while the analyser runs.

4.4.2. Invocation of the Analyser

The analyser is a client application that is started by the DBA. It takes arguments for the database that should be analysed and an optional time interval when only a part of the collected data should be processed. The analyser runs in the foreground for the time it computes the recommendations.

4.4.3. Processing the Workload

To prioritise the most stressing statements they are are ordered by their frequency times the actual cost value of their QEP that was recorded during query execution. This way, both the frequency and the overall cost of the statement are taken into consideration to find the queries that should get the most attention. The tests and recommendations implemented in this first version of the analyser are as follows:

- A statement is not using any secondary indexes – either no indexes are defined or the query may have to be restructured to use an index.
- Actual and estimated costs of a query differ more than 20% – this may be caused by missing or outdated statistics and the optimiser may not be able to find the cheapest plan.
- The query spends more than 50% of its overall time in OPF – this may be caused by a high number of indexes that increase the number of plans to enumerate.
- An index is never been used in the given workload – the DBA should drop unused indexes as they need to be maintained by the DBMS
- One or more attributes of a table have no statistics – the DBA should run optimizedb to create them.
- A table has more than 10% overflow pages – the DBA should restructure the table or modify it to storage structure B-Tree.
- There is more than one index defined on an attribute – the DBMS is only using one index, the other only adds overhead and should be removed
- There are indexes on more than the half of all attributes in a table – as too many indexes will decrease performance the DBA should try to remove some unnecessary indexes.

This set of rules and tests is very simple and may not fit to complex scenarios. For this to achieve a more comprehensive approach of data interpretation is needed that is not yet part of this work but will be discussed in chapter 7.
In addition to those tests, the analyser collects attribute candidates to recommend the creation of new indexes. This is first done per statement. The attributes referenced in the statement are ordered by their frequency and then new indexes are added up to a defined maximum of 50% of attributes per table. The indexes are created as virtual database objects which means that the system catalogues are updated to contain the new index but the data file is not filled with rows so that the virtual index comes with a constant overhead of only a small catalogue operation, no matter how big the underlying table is. The virtual index contains all the meta data needed by the optimiser to decide whether it is useful for the statement or not. It contains the estimated size of the index data file on disk to allow the computation of cost estimates and the creation of a QEP. The analyser creates all the possibly useful indexes and then uses OPF to decide which index will be used. Because 50% of all attributes in all tables used in the query will have an index at this point the enumeration of execution plans can potentially require a long time to complete – the longer OPF can scan through the possible plans the higher is the chance to find the optimal plan containing those indexes that are most useful. However, to keep the analyser

from running too long an enumeration timeout will be set after which OPF stops searching and uses the so far best plan instead. After the query has been optimised the analyser requests the new cost estimates and the list of virtual indexes used in the QEP over IMA. If the new plan is cheaper than the original one the virtual indexes are turned into index recommendations. After all statements have been tested and index recommendations have been collected for each statement, the indexes are created again as virtual objects and the workload is optimised as a whole to see the effect of all of the new indexes on all statements. This data is then used for the presentation phase. Each recommendation during the analyser phase is being logged even when it is a duplicate. This way, the DBA can prioritise the implementation of recommendations by their frequency.

4.4.4. Postconditions

After the analyser is done with scanning the workload and testing new indexes, it has collected a number of recommendations based on the rules explained above. It has also prepared the DBMS statistics in the *Statistics* table to be displayed to the DBA. Virtual database objects have been removed and the database is back in normal operation. The next section describes the presentation of results.

4.5. Present

In the last phase, the results of the analyser are presented to the DBA to show him where problems exist and to support him in his decisions what needs to be changed. The presentation of results is done by the analyser client described above in form of a static report.

4.5.1. Presentation of the Results

The collected recommendations from the analyse phase are grouped by the affected database object and the frequency how often the analyser recommended it. With this, a list of recommendations for statements, tables, attributes and indexes is being created with a rationale and – where possible – the SQL command that is needed to perform the change. Statistical data is presented in text and with diagrams that allow visual examination of DBMS conditions. Sections 5.3 and 6.1 will show examples of analyser reports and the diagrams shown there.

4.5.2. Postconditions

When the results have been presented, it is up to the DBA to decide what changes he wants to implement as the report only lists plain recommendations. The analyser primarily prioritises by frequency – changes that were recommended more often are believed to have greater influence on the system's performance. An index on an attribute that is used 100 times in the workload may provide better results than an index on an attribute that is only used 10 times. This is still a very naïve approach as it does not take table sizes and other factors into consideration. Possible improvements to the presentation are discussed in chapter 7.

5. Implementation

This chapter describes the implementation of the design analyser for Ingres. Since early 2008 the latest Ingres source is available to the public via Subversion[1] at `http://code.ingres.com`. The implementation in the DBMS core and of the monitor daemon was done using Ingres' compatibility layer (see section 2.1.1) which should ensure the platform independence of the code, however, the coding and testing was done only on 32-Bit Linux.
The next three sections will cover changes performed in the DBMS core, the implementation of the monitor daemon and of the analyser client. References to directory paths are expressed with environment variables where ING_SRC is the directory containing the source code and II_SYSTEM is the directory containing the Ingres installation. For the platform independence path separators are denoted as "!". Code listings are simplified and may be presented as pseudo code for better overview.

5.1. Core Changes

The DBMS core is used to collect data about the current workload on the system. The goal was to get enough data needed for the analysis without adding too much overhead so that the system isn't slowed down while collecting statistics. Where possible, the collection was placed directly at the origin of the data causing only constant overhead while keeping the number of loops over large structures low. However, this first experimental implementation has still a high potential of optimisation with hash tables and better placement in the code.

5.1.1. A New Subfacility

The monitoring code in the core has been placed in the system control facility. To keep the code in one place a new subfacility called SCM (for SCF Monitor) has been created in the folder ING_SRC!back!scf!scm. Here, all files belonging to the monitoring code are located together with the monitoring daemon described further below.
The scm folder contains the following files: *scmmain.c*, *scmima.c* and *scmonitor.sc*. The corresponding header files *scm.h* and *scmonitor.h* are located in ING_SRC!back!scf!hdr. In scm.h the structures to hold the workload data are defined as they were described in figure 4.3 while scmmain.c implements the helper functions to log the data. Figure A.1 in the appendix shows the SCM structure that contains arrays of all the other structures. The *_idx members are used for the ring buffer implementation that keeps the memory requirements at a constant level. MAXMONITOR is a defined constant of how many statements and objects – tables, attributes, indexes – to log at a maximum. For this experimental implementation the maximum was chosen

[1] http://subversion.tigris.org

to be 500. The MAXWORKLOAD constant, which is set to 1000, is the maximum number of statements that can be logged in the workload history. Given that the monitor daemon queries IMA every 30 seconds this means that with these settings the DBMS can monitor at most 500 distinct statements and 1000 statements in total within a timeframe of 30 seconds. If there are more then they either don't get logged or the two constants would need to be adapted.

5.1.2. Log Functions

In this section the functions to log statements and the referenced database objects are described in more detail.

Statements

Figure A.2 shows the structure to hold a statement. SCM_STATEMENT entries are unique over the database name and the query key. The workload history, which is the time when the statement was executed together with the cost values, is stored in SCM_WORKLOAD as shown in figure A.3.
The function to log a statement – scm_log_stmt() – is called in scs_sequencer() which is the main function in SCF to pass a query through the various server facilities. The statement is logged right after the call to PSF after which the query has been parsed. In figure 5.1 line seven the session control block is returned. This structure contains information that is global for the current session and which is used to get the statement text and the name of the database. The SCM structure in line eight is placed in Sc_main_cb which is a server-wide global structure that is persistent over session boundaries. This way, every session and every facility is able to access the SCM structure to log data. The hash key in line 11 is computed by calling HSH_char() – an Ingres implementation of a simple hash function which is used to compute a unique key of a statement string. Although the hash function trims leading and trailing whitespaces from the string, the same query with different case or order of attributes and tables will produce a different hash. Therefore, those statements will be treated as new ones which would need to be changed in a later version of the implementation. The call to scm_stmt_lookup() returns a pointer to an SCM_STATEMENT structure containing the current query when it was already executed and logged before or NULL if it wasn't.

```
1     VOID
2     scm_log_stmt( VOID )
3     {
4         /* Variable declarations */
5         ...
6
7         scb = scm_get_scb();
8         scm = &Sc_main_cb->scm;
9         query_text = scb->cs_scb.cs_diag_qry;
10        db_name = scb->scb_sscb.sscb_ics.ics_dbname.db_db_name;
11        hashkey = scm_hash(query_text);
12
13        statement = (SCM_STATEMENT*)scm_stmt_lookup(db_name, hashkey);
```

Figure 5.1: scm_log_stmt (scmmain.c) (1)

As mentioned in section 4.3.1, the number of entries to log must be limited to avoid infinitely growing memory requirements. The structures were implemented as ring buffers with a cycling index pointing to the current position in the buffer. In line 18 in figure 5.2, the position is checked against the maximum number of statements to log – if the number is exceeded the index is set back to zero and old entries are overwritten. For the first MAXMONITOR statements there is an additional overhead of attaching the SCM_STATEMENT to IMA with the call to MOattach() in line 28. Details about the handling of IMA structures are described in section 5.1.3 below. As soon as the full number of SCM_STATEMENT structures is created and attached there is no other overhead of memory allocation and the memory requirements become constant over the complete runtime of the DBMS.

```
16          if (!statement)
17          {
18                  if (scm->cur_stm_idx >= MAXMONITOR)
19                  {
20                      scm->cur_stm_idx = 0;
21                  }
22                  old_stm = scm->statements[scm->cur_stm_idx];
23                  if (old_stm != NULL)
24                  {
25                      statement = old_stm;
26                  } else {
27                      statement = (SCM_STATEMENT*)MEreqmem(...);
28                      MOattach(...);
29                      scm->statements[scm->cur_stm_idx] = statement;
30                  }
31          }
32          /* Fill struct with data */
33          ...
34      }
```

Figure 5.2: scm_log_stmt (scmmain.c) (2)

After the SCM_STATEMENT struct has been either created (when the statement hasn't been logged before) or updated (when it has been logged) a new SCM_WORKLOAD entry is created, appending the statement to the workload history.
In line three of figure 5.3 the SCM_WORKLOAD is attached to IMA and then initialised with data. The wallclock time in line eight is set to -1 so that the monitor daemon later can see which query is still being executed and therefore doesn't get put into the workload DB, yet. In line ten, a pointer to the current SCM_WORKLOAD structure is placed in the session control block so that the following logging functions can easily store their data.

```
1       ...
2           workload = (SCM_WORKLOAD*)MEreqmem(...);
3           MOattach(...);
4           scm->workload[scm->cur_wkl_idx] = workload;
5       }
6       workload->query_key = hashkey;
7       workload->time = now.TM_secs;
8       workload->wctime = -1;
9
10      scb->scm_current_stmt = workload;
```

Figure 5.3: scm_log_stmt (scmmain.c) (3)

Functions such as scm_log_opf_time(), shown in figure 5.4, are placed throughout the server code to capture information. scm_log_opf_time() is called in scs_sequencer() after OPF is finished with building the QEP for the query. CPU and disk I/O costs are also logged for the call to QEF – also in scs_sequencer() – to see how long it took to execute the query. All those functions contain a call to scm_get_cur_stmt(), as seen in line seven, which then returns the current SCM_WORKLOAD.

```
1         VOID
2         scm_log_opf_time(
3                 TIMERSTAT   *start,
4                 TIMERSTAT   *end )
5         {
6             SCM_WORKLOAD    *stm;
7             stm = scm_get_cur_stmt();
8             stm->opf_cpu =  end->stat_cpu - start->stat_cpu;
9             stm->opf_dio =  end->stat_dio - start->stat_dio;
10        }
```

Figure 5.4: scm_log_opf_time (scmmain.c)

Tables

Figure A.4 shows the SCM_TABLE structure containing table information. Entries are unique over the database name and the ID of the table which is an internal sequence number used in the system catalogues. The function to log tables – scm_log_table() – is placed in OPF in opv_parser() right after call to RDF which is responsible to get catalogue information for all tables used in the query. The call is placed within the existing loop over tables so that no overhead other than the actual logging is caused.
The table log function is very similar to scm_log_stmt() – the session control block and the SCM structure are retrieved to perform a lookup for the table. If the table has already been logged before then only its statistics are being updated. If it is a new table then either a new SCM_TABLE structure is being allocated and attached to IMA or an old one is overwritten in the same way as it is done for statements. As it can be seen in figure 5.5, the code to look up a table in the SCM structure is still quite expensive because it may need to loop over the whole array of tables until a decision can be made. The costs have been tried to minimise by moving the expensive string comparisons with STncmp() to the end of the test, however, it is believed that this lookup should be implemented much more efficiently with for example a hash map. But even with this approach the overhead remains within constant boundaries of $[1, MAXMONITOR]$ tests per function call.
In line 35 of figure 5.5, the call to scm_log_reference() creates a reference between the current statement and the table that was being logged. The SCM_REFERENCES structure, shown in figure A.7, acts as a relation between the statement and the tables, attributes and indexes used in the statement. With this, the analyser knows for example which attributes are used in a statement and can then recommend the creation of new indexes.

```
1         VOID
2         scm_log_table(
3                 i4          table_id,
4                 char        *name,
5                 i4          datapages,
6                 i4          overflow,
7                 i4          structure )
8         {
9             /* Variable declarations */
10            ...
11
12            scb = scm_get_scb();
13            db_name = scb->scb_sscb.sscb_ics.ics_dbname.db_db_name;
14            scm = &Sc_main_cb->scm;
15            i4 found = -1;
16
17            for (i = 0; i < MAXMONITOR; i++)
18            {
19                if (scm->tables[i] == NULL)
20                {
21                    break;
22                } else if (
23                    scm->tables[i]->table_id == table_id
24                    &&
25                    STncmp(scm->tables[i]->database, db_name, ↘
                          DB_MAXNAME-1) == 0 )
26                {
27                    found = i;
28                    break;
29                }
30            }
31            /* Code to allocate and attach */
32            ...
33            /* Fill the structure */
34            ...
35            scm_log_reference(SCM_TYPE_TABLE, table_id, 0);
36        }
```

Figure 5.5: scm_log_table (scmmain.c)

Attributes

The SCM_ATTRIBUTE structure, as shown in figure A.5, contains information about attributes used in the statement. Attributes are unique over the database name, the table ID and the attribute ID. Both IDs are required because the attribute ID reflects only the position of the attribute within the table. Attributes are logged in opz_addatts() with a call to scm_log_attributes(). This is where OPF fills the list of attributes needed for the query execution. As with the table logging, scm_log_attributes() is called right at the origin of the information to avoid additional overhead while cycling over a list of attributes.

Indexes

Index data is stored in the SCM_INDEX structure shown in figure A.6. Entries in this structure are unique over the database name and the index ID. Together with the members table ID and attribute ID the index can be referenced back to the attribute it was created on. The call to

scm_log_index() is placed in OPF in opv_index() which is responsible for getting all possibly useful indexes for the given query. If the index has been actually used in the QEP is not known until the enumeration phase is completed. In opj_joinop(), which passes the query through the enumeration, an additional call to scm_log_used_index() is placed to be able to tell which indexes were considered and which ones were used. Figure 5.6 shows that in this case an additional loop is required to get the needed information. After the call to opj_enum in line one the QEP is available with a list of all tables that will be used for execution – including indexes. The upper bound for this list is the maximum number of tables that can be joined in Ingres but in most cases this list should only contain a small number of entries so that this loop is not believed to cause much overhead.

```
1     subqpp = opj_enum(subqpp);
2
3     i4  i;
4     for (i = 0; i < global->ops_rangetab.opv_gv; i++)
5     {
6         RDR_INFO    *rel;
7
8         rel = global->ops_rangetab.opv_base->opv_grv[i]->opv_relation;
9
10        /* Check if this is an index */
11        if (rel->rdr_rel->tbl_id.db_tab_index > 0)
12        {
13            scm_log_used_index(
14                rel->rdr_rel->tbl_id.db_tab_index,
15                rel->rdr_rel->tbl_storage_type,
16                rel->rdr_rel->tbl_dpage_count,
17                rel->rdr_rel->tbl_opage_count );
18        }
19
20    }
```

Figure 5.6: Call to scm_log_used_index (opjjonop.c)

5.1.3. IMA Handling

The concept of IMA has already been described in section 2.1.5. The MO module handles data over class definitions – a class definition is a description of a variable or a structure member whose instances can later be accessed as IMA objects. A definition includes a unique name of the class, the space requirements, access rights and information on how to retrieve and store the data. Figure 5.7 exemplarily shows the definition of the query hash key member in SCM_STATEMENT. The name of the class, *exp.scf.scm.stm.query_key*, in fact only requires to be a unique string within all IMA classes but following the naming convention of a tree-like structure, this class is part of the experimental branch (as currently all other classes, too) and part of SCF and its subfacility SCM. The permissions are set to MO_READ which means that objects of this class will be readonly. IMA allows a more granular rights management to enable and disable read and write access for all or only certain users. MOuintget and MOnoset are part of the MO standard functions for retrieving and storing data in IMA objects. Objects of this class will provide unsigned integer values and cannot be set to a different value over IMA. There are other functions such as MOstrget or MOptrget for retrieving strings and pointers.

```
1     {
2     MO_CDATA_INDEX, "exp.scf.scm.stm.query_key",
3     MO_SIZEOF_MEMBER(SCM_STATEMENT, query_key), MO_READ, ↘
          scm_stm_index_class,
4     CL_OFFSETOF(SCM_STATEMENT, query_key), MOuintget, MOnoset,
5     0, MOidata_index
6     },
```

Figure 5.7: IMA Class Definition (scmima.c)

A list of class definitions like this one can be put into an array which is passed to MOclassdef() – with this call the classes are registered and structures or variables can be attached to IMA as it can be seen in figure 5.8.

```
1    char buf[80];
2    MOptrout(0, (PTR)statement, sizeof(buf), buf);
3    MOattach(MO_INSTANCE_VAR, scm_stm_index_class, buf, (PTR)statement);
```

Figure 5.8: MOattach (scmmain.c)

This will create an IMA object pointing to this single instance of the statement variable. This means that whenever the data in the statement variable respectively its member variables changes, the new value is visible in IMA immediately without any additional effort. Of course, for this to work the variable must be globally available so that it is not implicitly deallocated at the end of a function. As the SCM structure resides in the global server-wide control block in Sc_main_cb it never gets deallocated during the lifetime of a DBMS process.
To be able to access IMA objects from the outside they need to be registered as tables in a database. This can be every database but as a convention all IMA tables are registered in IMADB. Figure 5.9 shows the registration of a table called ima_scm_statements, containing all the members of the SCM_STATEMENT structure. Every column in the table points to one of the IMA classes that were defined. After the registration, a select on this table will return one row of data per every existing instance of the IMA class – in this case every SCM_STATEMENT structure that has been allocated and attached to IMA will produce a row.

```
1     register table ima_scm_statements (
2         server      varchar(64)     not null not default
3             is 'SERVER',
4         database    varchar(32)     not null not default
5             is 'exp.scf.scm.stm.database',
6         query_key   integer4        not null not default
7             is 'exp.scf.scm.stm.query_key',
8         query_text  varchar(1000)   not null not default
9             is 'exp.scf.scm.stm.query_text',
10        frequency   integer4        not null not default
11            is 'exp.scf.scm.stm.frequency',
12        time        integer4        not null not default
13            is 'exp.scf.scm.stm.time'
14    )
15    as import from 'tables'
16    with dbms = IMA,
17    structure = unique sortkeyed,
18    key = (server, query_key);
```

Figure 5.9: IMADB Table

In scmima.c IMA class definitions for all of the structures used by SCM were created and the corresponding tables in IMADB were registered plus an ima_scm_statistics table that contains several IMA objects from throughout the server. How those tables are then read by the monitor daemon is described in section 5.2.

5.1.4. Adding Virtual Indexes

To allow testing the performance improvement of new indexes, the DBMS core was expanded with the ability to create virtual indexes similar to the ones described in [Thi08]. The index is virtual because its data file is not populated with any rows. It is only an entry in the system catalogues that contains the description of this data file with the estimated number of tuples that a real index would have. This is enough for OPF to add the index to the enumeration and to create a QEP containing this index. The resulting cost estimates can be used for a comparison with the old QEP to see if creating this index would result in a cheaper plan.
A new keyword was added to the parser and virtual indexes can be created with:

```
create virtual index idx_name
  on table_name(attribute_name)
    with structure = BTREE
```

The current implementation only contains tuple estimates for a B-Tree structure, therefore, all virtual indexes must be created as B-Tree.

5.1.5. Adding a New Trace Point

Of course, virtual indexes cannot be used to execute a query and the DBMS would return an error when trying to access the non-existing index table. For this reason, the use of virtual indexes has been disabled by default and a new trace point was added that tells the DBMS to optimise using those indexes but not to execute the resulting QEP.
Trace points in Ingres allow the conditional execution of code based on decisions during runtime. Every server facility defines a set of trace points which are mostly used for debugging purposes. How to use a trace point in Ingres is shown in appendix B.
The trace point SC7 was added in SCF so that it is easily available in all server facilities. It is getting checked right before OPF loads the virtual index for a given attribute in opv_parser(). A second check is performed in ops_sequencer() to decide whether QEF will execute the query or not. Without SC7 being set, the DBMS will operate as normal and as long as the user doesn't query the virtual index table explicitly it won't interfer with user traffic. SC7 is a session-level trace point which means that is has only effect within the session that set it.
The trace point is also used in SCM to decide whether a statement or database object is being logged or not so that I) the workload statistics don't get changed when running the analyser and II) no virtual indexes are recorded in the workload DB. Instead of logging a statement to the IMA structures described above, SC7 will cause the cost estimates being written to another structure called SCM_ANALYZE, shown in figure A.8. This structure has only a single instance connected to IMA that is being filled with the estimates for the query that was last executed together with the list of virtual indexes used in its QEP. This way, the analyser can set trace point SC7 to execute a query with virtual indexes and to get the new cost estimates back over IMA to see if the new plan is using any of the new indexes and if it is cheaper than the old plan.

5.2. The Monitor Daemon

In IMA only a small time window of data is presented. Server statistics are only snapshots of the current situation and the logging of database objects is only done for a limited amount of statements resulting in a short-term history. To be able to analyse more than just a small set of data IMADB is being read on a regular base and the data is persistently stored in a dedicated database. This section describes the tool that performs this action. It was implemented as an Ingres database tool similar to createdb or optimizedb which means that the code of the tool is included in the source code of the DBMS and that it can use the same infrastructure as the DBMS core itself speaking of things like the memory management or the compatibility layer functions that make the tool platform independent. The tool was called and is now referenced as *monitordb*.

5.2.1. Writing a Database Tool

Figure 5.10 shows a most simple database tool. The EXsetclient() call tells the Ingres exception handling that this is a user tool running on a command line. With MEadvise() set to ME_INGRES_ALLOC the internal Ingres memory management is used when memory functions are called. The SIprintf then simply prints the string to stdout and PCexit exits the tool with zero as the return value.

```
1     /* Include headers */
2     ...
3
4     i4
5     main(
6         i4      argc,
7         char    *argv[] )
8     {
9         (void)EXsetclient(EX_INGRES_TOOL);
10        MEadvise(ME_INGRES_ALLOC);
11
12        SIprintf("Hello world!\n");
13        PCexit(0);
14    }
```

Figure 5.10: Hello World DB Tool

For monitordb an implementation with ESQL/C was chosen. ESQL/C allows the use of SQL embedded in C and C++ as it can be seen in figure 5.11. All source code files with extension .sc are passed through an ESQL/C preprocessor by the Ingres compile system and every *exec sql* command gets translated into a number of C function calls against the Ingres library *libq*. The resulting .c file is then compiled by the standard C compiler.
The code in figure 5.11 contains a declaration part in lines three to six. Only variables between those special ESQL commands can be used in ESQL statements as seen in line 12 where :db_name will be replaced by the value of the string variable. A cursor is getting declared for the given statement and the resulting rows are then fetched into a structure that is here called ESQL_SCM_STATEMENT. This and the other corresponding structures for tables, attributes, etc, are more or less a copy of the the SCM_* structures described in the previous section. However, because the structures need to be defined within an ESQL declare section and because of

the fact that the scm.h header cannot be included twice in the code those new structures were defined with an ESQL_ prefix.

```
1     exec sql include SQLCA;
2
3     exec sql begin declare section;
4         char                db_name[DB_MAXNAME];
5         ESQL_SCM_STATEMENT  statement;
6     exec sql end declare section;
7
8     exec sql connect imadb;
9
10    exec sql declare stm cursor for
11        select * from ima_scm_statements
12            where database = :db_name;
13
14    exec sql open stm for readonly;
15
16    for (;;)
17    {
18        exec sql fetch stm into :statement;
19        if (sqlca.sqlcode == 100)   /* No rows left */
20        {
21            break;
22        }
23        /* Save content of statement */
24        ...
25
26    }
```

Figure 5.11: ESQL Example

5.2.2. monitordb

Figure 5.12 shows a simplified version of monitordb's main() function. monitordb requires one command line argument being the name of the database to monitor. A connection to this database is being opened to validate if the database exists. monitordb then forks into the background continuing as a daemonised process until it is getting killed from the outside.
The tool enters the main loop and starts to query IMADB for the current content of the ima_scm_* tables. This data is stored in the ESQL_* structures and monitordb sleeps for a given period of time which in the current implementation is set to 30 seconds. After ROUNDS number of wakeups monitordb switches over to the workload DB session and stores the collected IMA data. This is currently done every ten rounds resulting in an interval of five minutes. This way, monitordb delays disk access on the workload DB to avoid constant interference with the user traffic on the system.
The workload DB contains the same table schema as the one for SCM in IMADB except that the statistics table is now growing over time storing the history of values. As described in section 4.3.3, the workload DB needs to be limited in space and therefore only contains data from within a specific timeframe. To implement this, monitordb in line 37 of figure 5.12 calls a function to clean up old data from the workload DB. All rows with a timestamp of more than seven days in the past are going to be deleted here – keeping the data from the last seven days is believed to be enough to capture the significant workload and statistical data of a database.

```
1     i4
2     main(
3         i4      argc,
4         char    *argv[] )
5     {
6         STncpy(db_name, argv[1], DB_MAXNAME);
7         exec sql connect :db_name;
8         if (sqlca.sqlcode != 0)
9         {
10            PCexit(FAIL);
11        }
12        exec sql disconnect;
13
14        switch (pid = PCfork(&status))
15        {
16            case 0:
17                break;
18            default:
19                PCexit(OK);
20        }
21        exec sql connect workloaddb session 2;
22        exec sql connect imadb session 1;
23
24        while(1)
25        {
26            get_ima_content();
27
28            if (++round > ROUNDS)
29            {
30                round = 0;
31                exec sql set_sql(session = 2);
32
33                put_content_to_workloaddb();
34
35                if (++delold > DELOLD)
36                {
37                    delete_old_content_in_workloaddb();
38                    delold = 0;
39                }
40                exec sql set_sql(session = 1);
41            }
42        PCsleep(SLEEP);
43        }
44    }
```

Figure 5.12: monitordb main() (scmonitor.sc)

Figure 5.13 exemplarily shows the put_statements() function which stores the collected IMA data in the workload DB. The select statement in line seven is used to first see if the statement has already been inserted in the workload DB and second to see if the IMA entry is newer and if the workload DB needs to be updated with more current statistics. The function to process the data in ESQL_SCM_STATISTICS looks similar but instead of updating existing rows the data is always appended to allow trend analyses.
monitordb can be started several times on different databases to monitor more than one database at a time. The tool uses Ingres' autocommit setting to avoid long lasting locks on the workload DB. As IMADB tables are only virtual tables there are no locks and autocommit isn't needed for IMADB.

```
1     VOID
2     put_statements( VOID )
3     {
4         for (i = 0; i < scm->cur_stm_idx; i++)
5         {
6             statement = scm->statements[i];
7             exec sql select time
8                 into :time
9                     from statements
10                    where database = :db_name
11                    and query_key = :statement.query_key;
12
13            if (time > 0)
14            {
15                if (statement.time > time)
16                {
17                    exec sql update statements
18                        set
19                            frequency = :statement.frequency,
20                        where database = :db_name
21                        and query_key = :statement.query_key;
22                }
23            } else
24            {
25                exec sql insert into statements
26                    values (:statement);
27            }
28        }
29    }
```

Figure 5.13: put_statements (scmonitor.sc)

How the data that is collected in the workload DB is being processed and analysed is described in section 5.3.

5.2.3. Database Alerts

To alert the DBA in problematic situations, monitordb listens to database events. A database event can be raised with an SQL command in, for example, a procedure as shown in figure 5.14.

```
1     create procedure alert
2         (text varchar(1000) not null) as
3     begin
4             raise dbevent monitor_alert :text;
5     end
```

Figure 5.14: Database Procedure alert()

This procedure can be called by a trigger (or a rule in Ingres) such as the one in figure 5.15 which is fired whenever the number of current sessions in the statistics table reaches the maximum number of sessions on the DBMS.
monitordb can be invoked with an additional command line argument that is an external executable. The tool polls for raised events in its main loop and whenever an alert is detected, this

```
1     create rule max_sessions after insert
2         of statistics
3         where new.current_sessions = new.max_sessions
4         execute procedure alert('Maximum number of sessions reached!')
```

Figure 5.15: Database Rule max_sessions

external command is called with the text description that was passed on to the database procedure as its argument. The external command could be the system's mail program, so that any alert is being sent as a mail to the DBA. With this alerting mechanism, monitordb allows active observation of the system without having the DBA to manually watch critical values.

5.3. The Analyser Client

The analyser client is a stand alone application that is invoked by the DBA. For the implementation of the analyser the scripting language Python[2] has been chosen for various reasons. It is an easy-to-learn high-level language that offers object-orientation and a comprehensive set of libraries, called modules. The script code itself is platform independent and the Python interpreter is available for a wide range of platforms. Ingres offers a database driver for Python and there is a Python wrapper for gnuplot[3] which was chosen for drawing diagrams.
The analyser client is considered as being highly experimental and as stated in section 4.4 most of the implemented tests are still very naïve. This first implementation of the client is meant to exemplarily show potential ways to interpret the workload data and to present recommendations to the DBA.
The Python code was separated into modules placed in ING_SRC!front!misc!analyzedb: *Analyze.py* contains all the classes used to perform the tests and create the recommendations, *Present.py* contains the code to create the result report, in *Draw.py* the gnuplot wrapper class is defined, in *Db.py* the Ingres connection code is placed, *Containers.py* defines classes to hold the workload data, *functions.py* contains some global help functions and *Config.py* defines the default settings for the analyser. The entry point of the tool is in *analyzedb*.

5.3.1. analyzedb

analyzedb is being called by the DBA on the command line. It takes a number of arguments to configure the analysis. The only required argument is the name of the database that should be analysed. Optional arguments include the vnode, query limit and timeout. A vnode – or virtual node – is used in Ingres to connect to a remote database. Per default the analyser connects to vnode *(local)* but it can also connect to any other remote database that was compiled with monitoring code. The query limit is per default set to -1 which means that the analyser processes all statements in the workload. If that is too much it can be set to an absolute number of statements. The timeout parameter sets the joinop timeout for the enumeration. This is the maximum time OPF will spend to find the optimal QEP for a statement. The default is set to

[2] http://www.python.prg
[3] http://www.gnuplot.info

300 seconds.
As the tool is written in Python and uses some external libraries is has some prerequisites:

- Python 2.5 or higher
- gnuplot 4.2 or higher with png support
- gnuplot-py 1.8 or higher
- Ingres Python DBI driver

In addition to that, Ingres must be set up with ODBC support as the current version of the Python driver uses ODBC to communicate with the server.
All the prerequisites are available for most of Ingres' supported platforms. On platforms that don't provide the necessary dependencies the tool can be used remotely on a different machine.

5.3.2. Preparing the Analysis

Figure 5.16 shows the sequence of method calls to prepare and perform the analysis. First, the connection to all of the involved databases is established. The connection to the workload DB is set up with autocommit to avoid locking problems with the monitor daemon. As IMADB is no real database it doesn't need to be set up with autocommit. The connection to the user DB is created without autocommit so that virtual indexes are being rolled back not only when the analyser is completed but also in case of an error. However, this means that there may be locks on the database that block other user traffic. Because of that, it is recommended that the user database is not in use for the time analyzedb runs.

```
1     def start(self):
2
3         self.__userdb = DB(self.__config, self.__config.userdb)
4         self.__imadb = DB(self.__config, "imadb")
5         self.__workloaddb = DB(self.__config, "workloaddb", True)
6
7         """ Tell the DBMS we're going to test now """
8         self.__userdb.execute("set trace point sc7")
9         self.__userdb.execute("set joinop timeout 300")
10
11        self.__load_indexes()
12        self.__load_attributes()
13        self.__load_tables()
14        self.__load_workload()
15        self.__load_statements()
16
17        for statement in self.__statements:
18            self.__process_statement(statement)
19
20        """ Get the cost of the new configuration """
21        self.__test_configuration()
22
23        self.__process_statistics()
```

Figure 5.16: start() (Analyze.py)

In line eight, the new SCF trace point described in section 5.1.5 is set to enable the use of virtual indexes for this session. In line nine, the joinop timeout for the session is set to an absolute value so that OPF doesn't stop searching for a cheaper plan when the time spent so far exceeds the time of the currently cheapest plan. This way, OPF has more time to find the optimal plan that may contain one of the new indexes.
For each table in the workload DB a class has been defined as a container. First, the table, attribute and index containers are filled with all existing database objects in the user database. Then, the list of entries in the workload history and the list of statements is filled – the analyser filters all the selects from the workload as only they get processed. To every statement the list of referenced tables is attached. For each table the list of referenced attributes and for each attribute the list of defined indexes is attached. This way the analyser knows what tables, attributes and indexes are used in a certain statement. As every variable in Python is internally handled as a pointer this can be done efficiently without having many copies of the same object.

5.3.3. Performing the Analysis

Each statement is sent to __process_statement() where the actual analysis is being performed. All the tests described in section 4.4.3 are implemented here and figure 5.17 shows some examples. In line five, the percentage of the time the statement spent in OPF is calculated. If this is more than 50% then a recommendation is logged in line ten. Recommendation templates are stored in a dictionary and accessed by keys such as the high_opf_cost used in this case.

```
1     def __process_statement(self, statement):
2
3         """ Statement tests """
4
5         pct = 100 * (statement.opf_cpu + statement.opf_dio + 1) /  \
6             (statement.opf_cpu + statement.opf_dio +  \
7             statement.qef_cpu + statement.qef_dio + 1)
8
9         if pct > 50:
10            self.__recommend(statement, "high_opf_cost", (pct))
11
12        for table in statement.tables:
13
14            """ Table tests """
15
16            if table.overflow_pages > table.data_pages * 10 / 100:
17                self.__recommend(table, "overflow_pages", (table.name))
18
19            for attribute in table.attributes:
20
21                """ Attribute tests """
22
23                if len(attribute.indexes) > 1:
24                    self.__recommend(attribute, "duplicate_index", \
25                        (table.name, attribute.name))
```

Figure 5.17: __process_statement() (Analyze.py)

For each table used in the statement additional tests are performed. In line 16, the analyser calculates the percentage of overflow pages over the number of data pages. If there are more than 10% overflow pages then a recommendation is logged that will tell the DBA to modify the

table to B-Tree. Tests for attributes include the search for duplicate indexes that will cause a recommendation to drop all but one index on this attribute.

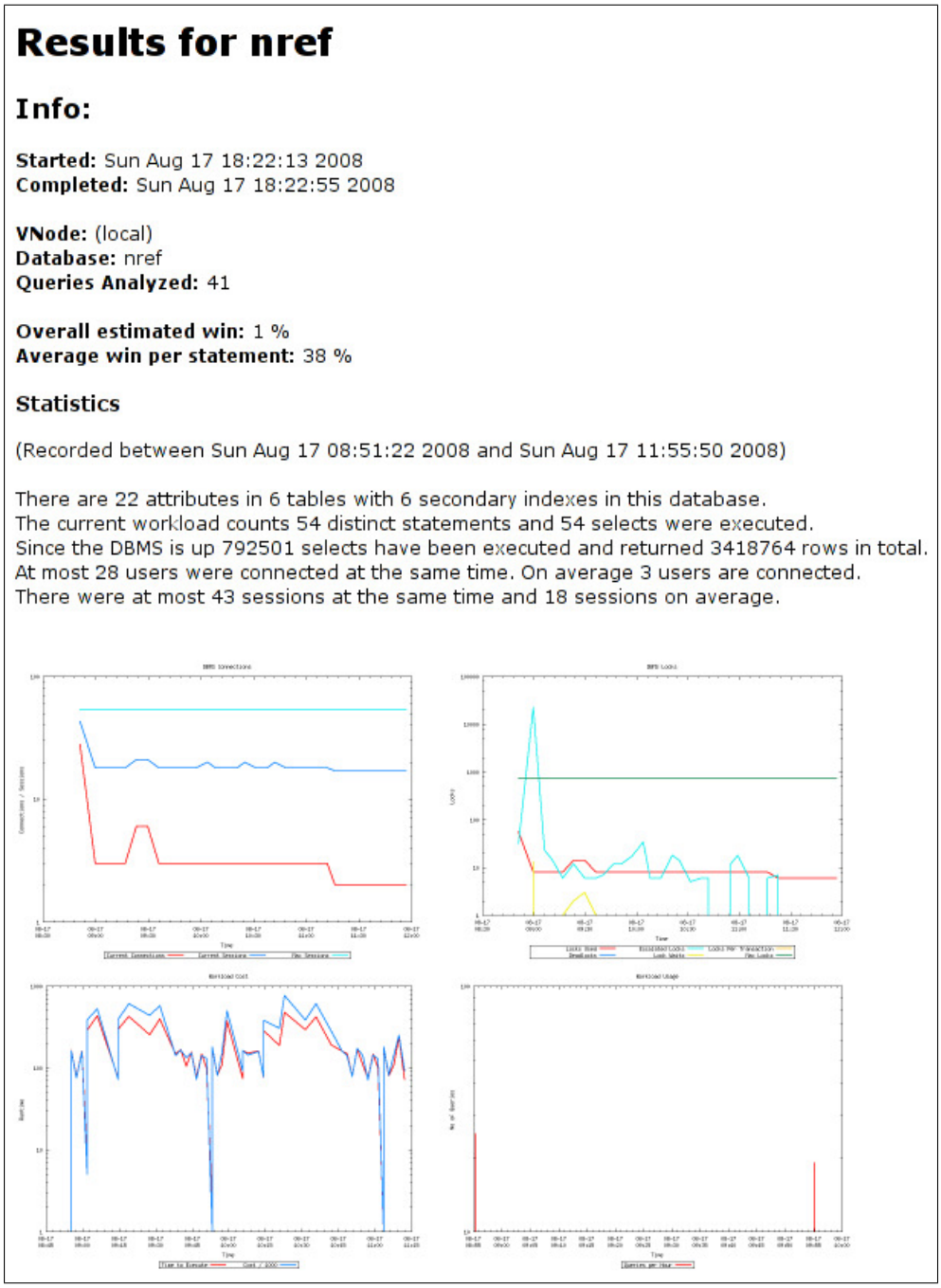
```
1     max_idx_count = len(table.attributes) / 2
2     if max_idx_count < 5:
3         max_idx_count = 5
4     idx_count = number_of_indexes_on_table
5
6     for attribute in table.attributes:
7         if idx_count < max_idx_count:
8             if len(attribute.indexes) == 0:
9                 idx_count += 1
10                test_indexes.append(attribute)
11
12    for attribute in test_indexes:
13        self.__userdb.execute("create virtual index...")
14
15    self.__userdb.execute(statement.query_text)
16
17    result = self.__imadb.execute("select \
18            cpu, dio, vindexes from ima_scm_analyze")
19    cpu = result[0][0]
20    dio = result[0][1]
21    newcost = cpu + dio
22
23    if newcost < oldcost:
24        """ Recommend new indexes """
25
26    self.__userdb.rollback()
```

Figure 5.18: Find New Indexes (Analyze.py)

Per definition the analyser allows indexes on 50% of all attributes per table. For index recommendations the number of existing indexes is counted and the rest is filled up with virtual indexes, created in line 13 of figure 5.18. In lines 15 and 17, the statement is executed and the new cost estimates together with the list of used virtual indexes is read from IMADB. If the new plan is cheaper than the old one recommendations for the used indexes are logged and the transaction is rolled back to remove the virtual indexes. After all statements have been processed the analyser creates all the recommended (virtual) indexes at once and runs the whole workload in one batch to get the final cost estimates. These values can then be used for a comparison between the old overall performance and the estimated new one.

5.3.4. Presenting the Results

For this implementation a presentation using HTML has been chosen because of its ease of use and flexibility. The output of the analyser is an HTML report with textual explanations and diagrams created with gnuplot.
The first part of an example report can be seen in figure 5.19. The DBA is presented with a set of general statistics about the database and diagrams that should help to identify and even to forecast possible problems. The diagrams are shown as thumbnail graphics and can be enlarged. Chapter 6 will show some example diagrams and discuss them. The report shows the overall performance win as well as the average win per statement that is expected when all recommended indexes are implemented. These numbers don't take into account other changes



Results for nref

Info:

Started: Sun Aug 17 18:22:13 2008
Completed: Sun Aug 17 18:22:55 2008

VNode: (local)
Database: nref
Queries Analyzed: 41

Overall estimated win: 1 %
Average win per statement: 38 %

Statistics

(Recorded between Sun Aug 17 08:51:22 2008 and Sun Aug 17 11:55:50 2008)

There are 22 attributes in 6 tables with 6 secondary indexes in this database.
The current workload counts 54 distinct statements and 54 selects were executed.
Since the DBMS is up 792501 selects have been executed and returned 3418764 rows in total.
At most 28 users were connected at the same time. On average 3 users are connected.
There were at most 43 sessions at the same time and 18 sessions on average.

Figure 5.19: Analyser Result Report – Statistics

such as the creation of statistics or the restructuring of tables so that the real performance win will probably differ.
Figure 5.20 shows the recommendation part of the report. The cost diagram, also discussed in the next chapter, compares actual cost to old and new cost estimates of the ten most expensive statements. Here, the DBA can see where estimates don't match the actual execution costs and which of the expensive queries will benefit from the implementation of new indexes. A complete list of queries is attached to the end of the report with query text and cost values.
The recommendations are presented in textual form grouped by statement, table, attribute and index with the most frequent ones first. Statements are referenced with their hash key and the

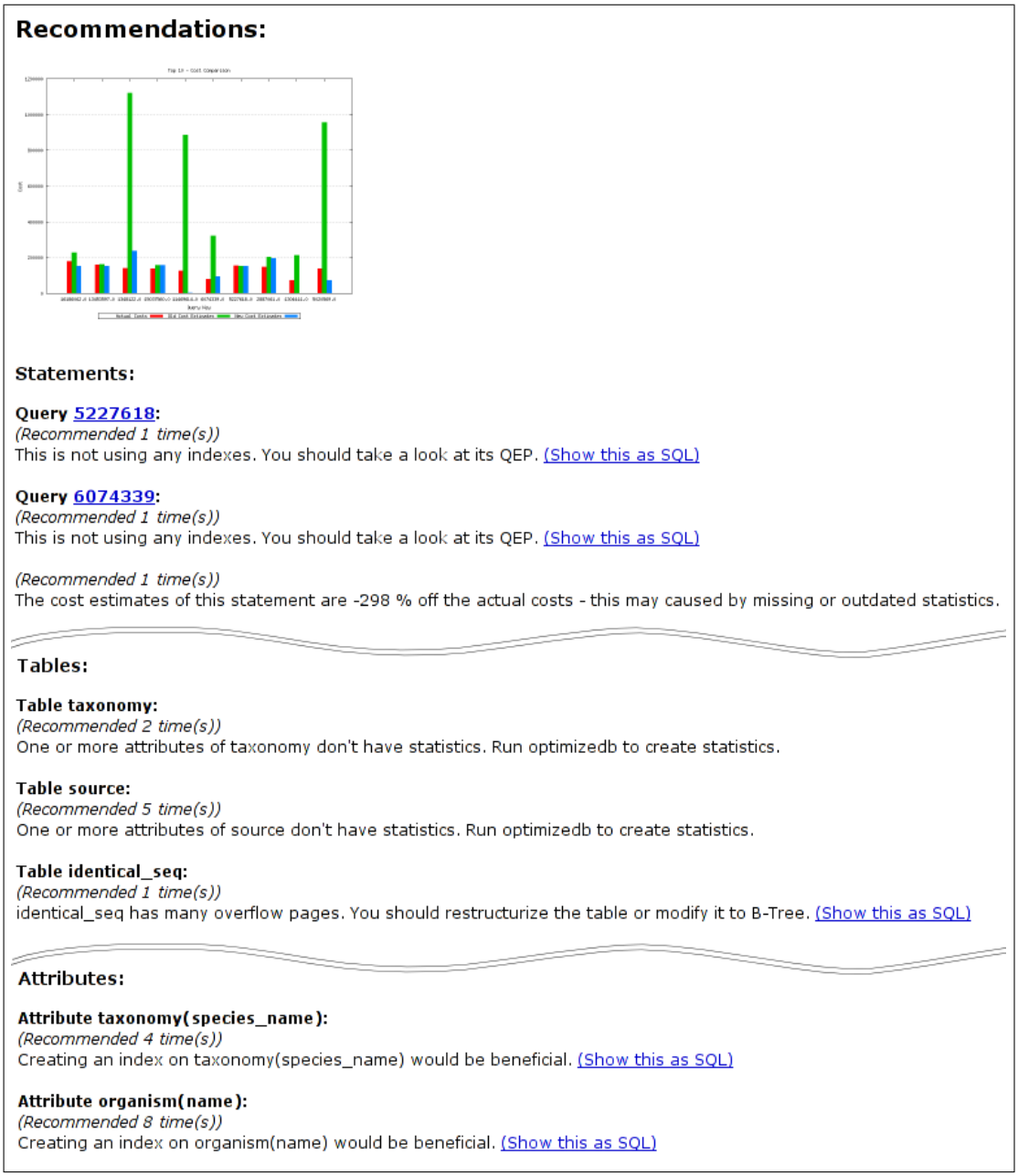


Figure 5.20: Analyser Result Report – Recommendations

DBA can click on the key to jump to the statement text. The DBA sees how often a change has been recommended during the analysis together with a description of the change. Where applicable, the SQL command to implement the change can be displayed.

5.4. Packaging

This section briefly describes how the design analyser was packaged with Ingres. The goal was to integrate the analyser into the existing Ingres structure to avoid any additional manual installation step.

The monitoring code is compiled into the DBMS core and therefore doesn't need any special handling. The IMA tables were added to the makimau.sql and makiman.sql scripts in ING_SRC!front!st!vdba that are executed to create the IMADB when installing the DBMS. The workload database is also automatically created during the installation process through a call in the DBMS installation script iisudbms. The monitor daemon and the analyser client are both installed in II_SYSTEM!ingres!bin which is in the PATH of the Ingres user. With that setup the DBA can immediately start to use the design analyser by starting the monitor daemon on the target database.
The HTML report of the analyser is per default placed in II_SYSTEM!files!analyser but this location can be overwritten with a parameter when calling the analyser. For example, when there is an HTTP server on the system the analyser report could be written to its document root and made available over the net.

6. Evaluation

This chapter will now discuss the performance of the concept of the design analyser described in this work. The next sections will cover use cases and possible interpretations of analyser results, time and space consumption especially of the DBMS core changes and the monitor daemon and a set of experiments showing the performance of the actual implementation shown in the previous chapter.

6.1. Application

The monitoring of workload and statistical data in the DBMS is the tool. The correct usage of this tool – the analysis of the data – is what makes the system useful. This section briefly discusses some use cases and what the results of the design analyser can tell in these situations. In section 6.3.2 the analyser and the actual performance win on a real workload is tested on the example of the NREF database.

6.1.1. Reporting

The analyser creates a report on system statistics and usage patterns of the database. The current implementation only presents a small subset of the statistical data that is available in the DBMS core. The two main examples that were chosen are connections and sessions as well as the locking system.
Figure 6.1 shows a scenario in which the number of open sessions on the DBMS has reached the current maximum that is configured. The DBA can see that the number of users and sessions was constantly rising and then hit the limit several times. This is where a trigger on the statistics table in the workload DB could have alerted the DBA before the limit has been hit.
Statistics for the locking system can also help to identify problems. The locking graph shows not only the maximum number of locks together with the number of currently used locks but also the number of lock waits, escalated locks and deadlocks. Lock waits describe how often a lock couldn't be granted immediately but had to wait. Escalated locks are many page-level locks that were turned into one table-level lock. The number of deadlocks that occurred is a sign of problems with the application using the database. The DBMS has detected a circle in the locking system that could only be resolved by aborting a transaction.
In figure 6.2 several problems can be seen. First, a high number of lock waits indicates that many sessions on the system try to lock the same database objects. The DBMS also detected a reoccurring and high number of deadlocks. The DBA can only see the anomalies but not what actually happened. In addition to the locking problems, the graph also reveals that the number of used locks is never near the configured limit so that the DBA can safely reduce the maximum number of locks to free system resources.

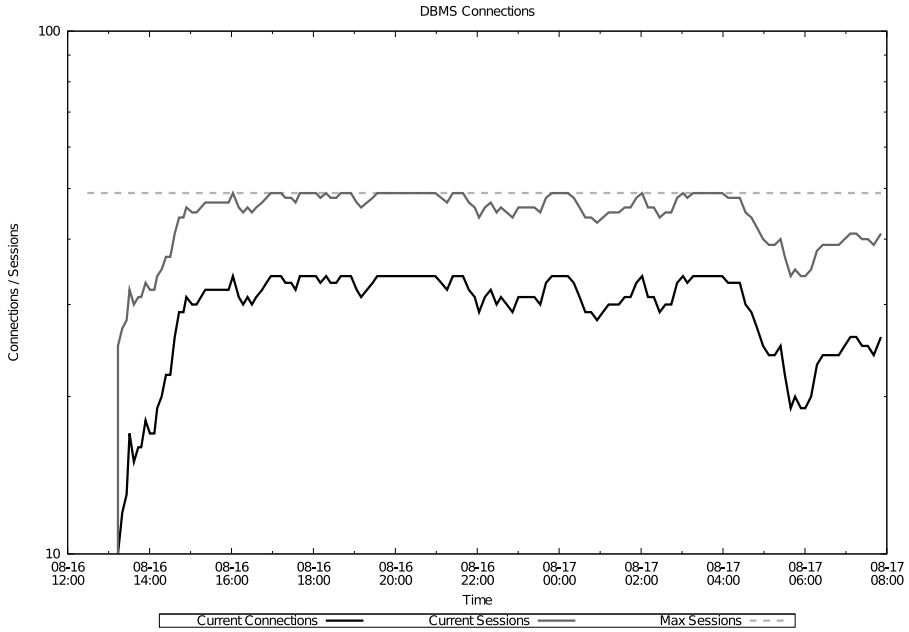


Figure 6.1: DBMS Connections

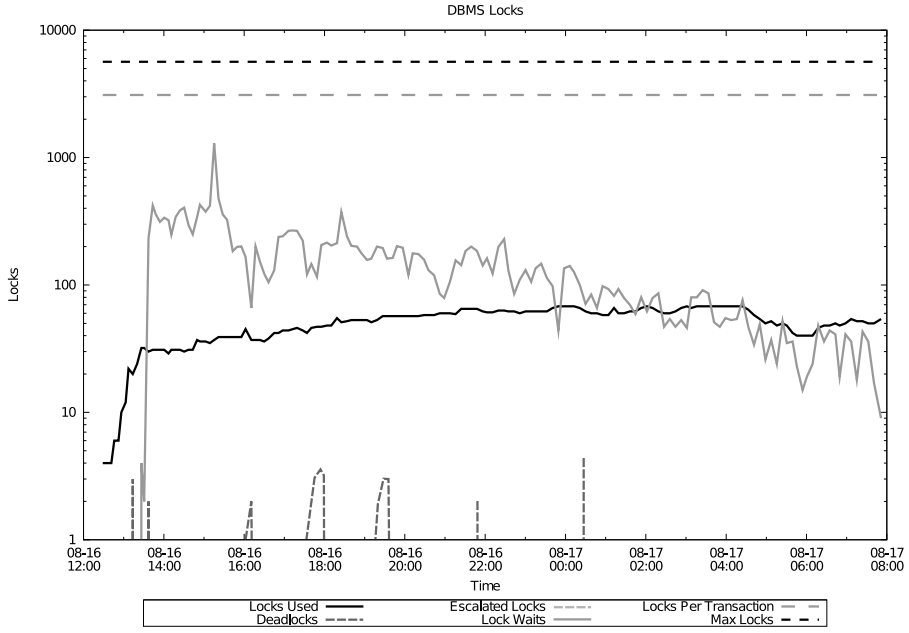
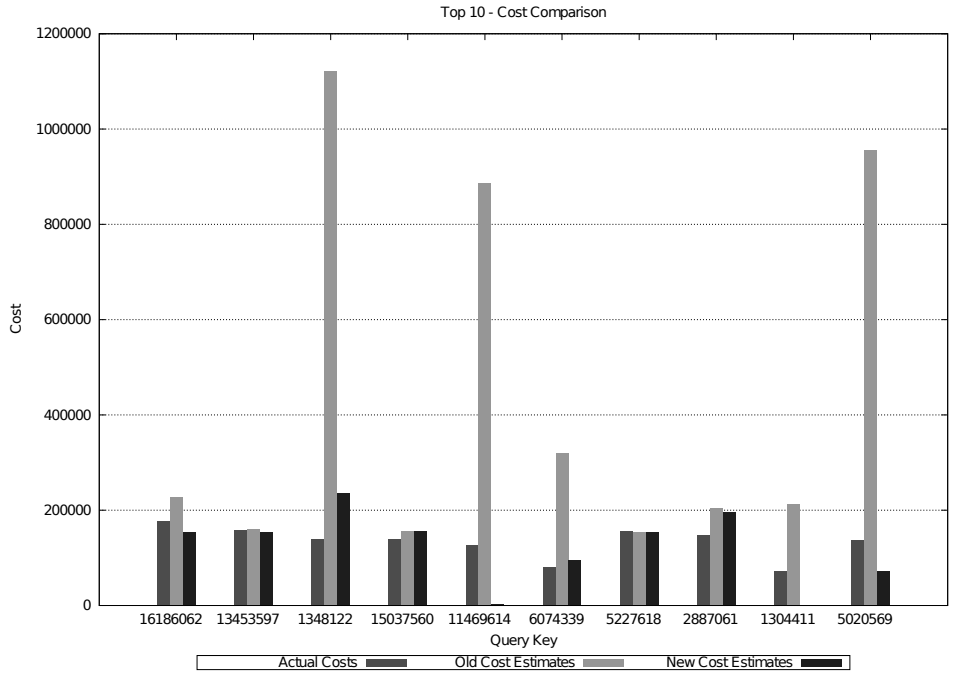


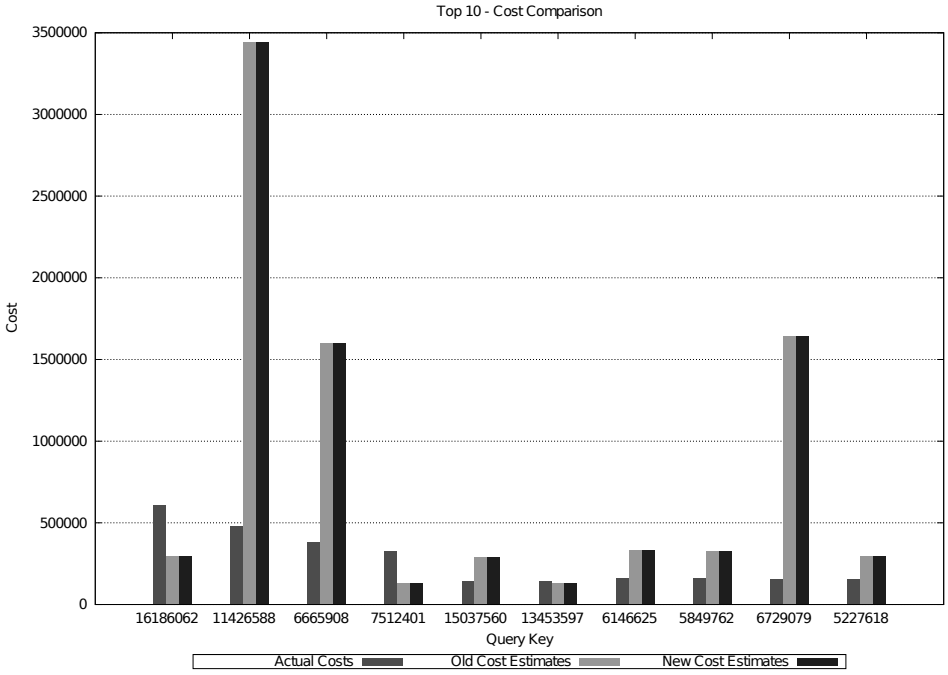
Figure 6.2: DBMS Locks

6.1.2. Recommendations

Recommendations are shown as a listing grouped by query, table, attribute and index together with a diagram of the ten most expensive queries in the workload.
Figure 6.3a shows the results of an analyser run on an unoptimised database. The numbers on the x axis are the query hash keys. Here, estimates of the original plan can be compared to the estimates of the plan that would be used when the new indexes were created. OPF expects all of



(a) Unoptimised



(b) Optimised

Figure 6.3: Top 10 Costs - Diagram

the query plans to become cheaper when the recommended indexes are in place. In addition to that, the DBA can compare the actual costs of the plan to the estimated values. For example, the estimates for the query with key 1348122 are significantly higher than what the DBMS actually needed to execute it. This is an indication for false cardinality estimates where OPF probably assumed that a lot more tuples need to be processed. This can be often encountered with the

creation of statistics.
Figure 6.3b shows an analyser result of an already optimised database that has the necessary indexes and table statistics. It can be seen that all the new estimates are equal to the old estimates which means that OPF already had the optimal plans. But even with an optimised database actual and estimated costs can still differ. This is where Oracle's concept of SQL Profiles (described in section 3.2.2) tries to deliver more accurate estimates by storing statement-specific cardinalities instead of only table statistics.
Together with this cost diagram the analyser provides a textual list of all the recommendations as for example:

"*(Recommended 6 time(s))*
Creating an index on neighboring_seq(nref_id_2) would be beneficial."

Together with the SQL command:

"*CREATE INDEX nref_id_2_idx ON neighboring_seq(nref_id_2)*
WITH STRUCTURE = BTREE \g"

For new indexes, the DBA sees the expected performance win and can easily decide to create it or not. Other recommendations such as

"*This statement spends 100 % of its time in the optimizer.*
You should take a look at its QEP."

are still very simple and generic. Here, the DBA needs to find the cause of the problem all by himself by looking at the query and the QEP. This could be a query with many tables to join so that OPF indeed requires too much time – it could also be a very simple query that is executed in virtually no time. Such queries are not even sent to OPF but still the analyser reports this recommendation.
Section 6.3.2 of this chapter will show the performance win when implementing all the recommended changes with the example of the NREF database.

6.2. Time and Space Consumption

This section discusses the memory consumption and runtime of the implementation. While the analyser client is intended to run only in a maintenance window while no other user is on the database, the monitoring code in the DBMS core and the monitor daemon running in the background are both designed to be lightweighted. How the implementation actually performs can be seen in the experiments in section 6.3.

6.2.1. DBMS Core

The monitoring code in the DBMS core adds new functions that are called during the optimisation of a query. The function scm_log_stmt() is called once for every query and it contains the generation of the query hash key, the lookup of the statement in the list of already known queries and the possible allocation of a new statement structure in memory. The hash key generation and the lookup are both linear to the text length respectively to the number of statements but they are limited by their defined maxima. With the chosen limits of 1000 characters per statement and 500 statements in memory scm_log_stmt() takes $1500 * const$ computational steps in the worst case. The overhead added by the other functions for logging tables, attributes and indexes is also limited. The table and attribute functions are called right when the optimiser

calls RDF to get the object descriptions. This again adds a maximum number of $500 * const$ steps to lookup the object in memory plus the additional constant number of steps to allocate the new structure and to log the data. To log indexes an additional loop over the list of tables was required so that this takes more steps but still has a defined limit as the number of tables in a single statement itself is limited. The experiments in section 6.3 will show the real impact of this overhead.
The same as for time consumption applies to the memory requirements. Every statement structure requires only a constant space in main memory. For the SCM_STATEMENT structure a space of $500 * 1000$ bytes plus a constant amount of additional bytes for the other members is needed. The other structures such as SCM_WORKLOAD or SCM_TABLE each need between 50 and 100 bytes. With the current limits of maximum 500 database objects and 1000 statements in the workload history the SCM structure that holds all the pointers to the other structures takes about 17000 bytes. Adding all this, the whole monitoring takes about 815 kilobytes of main memory which grows linear with higher maxima. With a maximum of 1000 database objects and 10,000 statements in the history the memory consumption rises up to 2.2 megabytes and with 10,000 objects and 100,000 statements 22 megabytes are needed. The number of objects and statements that fit into the structures defines the data resolution of the monitoring code. Given that the monitor daemon runs every 30 seconds, a maximum of 1000 statements in the workload history means that up to 33 statements per second can be recorded. Everything above that is not getting logged unless the monitor limits are increased.

6.2.2. Monitor Daemon

For the first part of the monitor daemon the memory consumption and runtime is similar to the core changes. Again, a limited amount of structures with constant sizes is kept in memory. The memory needs are higher as the daemon stores the data over several rounds. With the current implementation the monitor daemon queries IMA five times before the data is written to the workload DB. This means that the memory needed is five times the amount needed in the DBMS core. As there are no locks, transactions or disk accesses on IMADB, querying IMA data is also a constant computational task.
The second part of the daemon includes inserts and updates in a database which requires disk accesses. While the number of inserts and updates is still limited by the maximum number of statements and database objects that were logged, the disk access itself adds an unknown number of computational steps. This can be the time to allocate new pages on the disk, the time to load new pages to the cache, etc.
To minimise the impact of these varying requirements the number of inserts and updates has been tried to reduce and the actual storage of data in the workload DB is done only every five minutes. Section 6.3 will show experiments with and without the monitor daemon to see the impact of the additional disk accesses.
The workload DB that is filled by the daemon grows over a period of seven days. At a constant load of 33 statements per second the workload history table contains nearly three million rows after one day and about 20 million rows in a week.

6.2.3. Analyser Client

Although the implementation of the analyser client still tries to be efficient, it wasn't designed to be lightweighted as runtime and memory consumption weren't considered to be a problem. The analyser is intended to run only while no other load is on the database. The preparation of the analysis depends on the amount of objects and statements in the workload DB. In the worst case 20 million rows need to be scanned to find the most expensive statements. Depending on the complexity of statements to analyse the most expensive part is the testing of virtual indexes where the optimiser performs the enumeration of query plans with a possibly large number of indexes. To keep a limit on the runtime the analyser sets an enumeration timeout after which the optimiser stops searching.

6.3. Experiments

This section shows a set of experiments that were performed first to see the impact of the monitoring on the DBMS and second to show the effect of the analyser results on the workload performance. The test setup is the same as described in section 3.2.1 but with a Gentoo Linux 32-Bit installation instead of Windows. The Ingres binaries were self-compiled for both versions with and without the monitoring and both versions were compiled with debug symbols. Because of that, the test results shown here cannot be taken as figures of a production system and are again not comparable to the results in chapter 3. They can only be seen as relative changes between different code versions of Ingres.

6.3.1. Monitoring Tests

The experiments in this section should show the impact on the overall system performance caused by the overhead of the monitoring code in the DBMS core and the monitor daemon. It is expected that for a workload of expensive queries that take several seconds or minutes to complete the impact of the monitoring is minimal and negligible because the additional function calls in the core are all of subsecond runtime and are only executed once per query. The impact of the monitoring daemon could already be visible 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 queries. With a higher throughput of queries a growing impact on the performance is expected. For queries that execute in a fraction of a second the overhead of the monitoring code becomes substantial. The overhead of the monitor daemon increases with the number of rows that need to be written to the workload DB.

Test Setup

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 work. All default settings were kept. The NREF database as described in section 3.2.1 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. Same DBMS configuration and NREF database.

III) The same Ingres instance but with the monitor daemon running in the background.
These setups will show the influence of the monitoring code in the core alone and of the monitor daemon running on the same machine.
On each of these setups three different tests were performed – first, the set of 50 NREF queries that was used to test the other DBMS in chapter 3 was executed. Second, a set of 50,000 simple joins of two tables was executed. All queries 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 queries in the form of

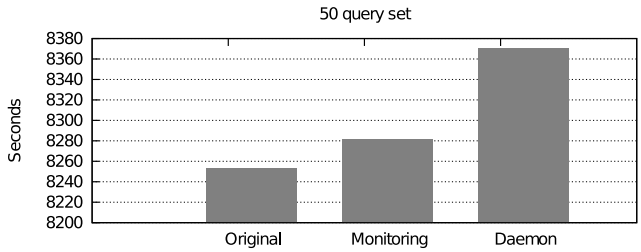
```
select p.nref_id
  from protein p
  where p.nref_id = 'NF00000001'
```

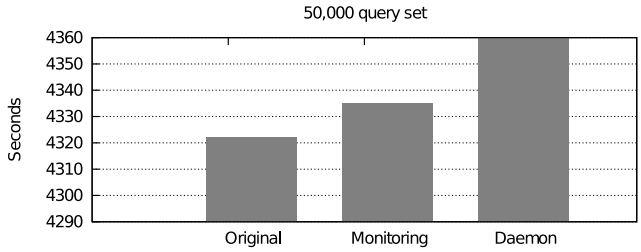
to see how the monitor lowers the pure throughput of query processing.
All tests were repeated three times to minimise local anomalies. The results are presented with the absolute time needed to complete the test and the ratio of queries per second for the last two tests.

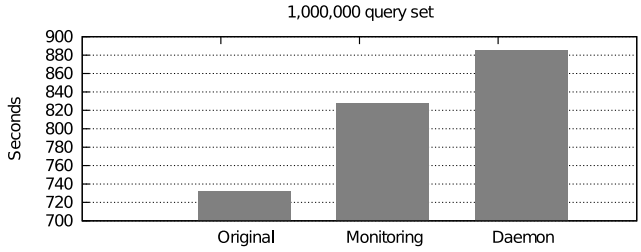
Results

Figure 6.4 shows the result matrix 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 queries executed. For the 50 query test the ratio of queries per second is far below one and therefore was left out.
The tests show that the monitoring code in the DBMS core has very little influence when executing complex and expensive queries. The DBMS needs several 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 greater impact can be seen with the monitor 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 query set. 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 monitor daemon doesn't slow down the query execution as much as before. Still, it can be seen that the DBMS without the daemon has a higher troughput of queries.
The 1m test eventually reveals the impact of the monitoring. The one million very simple queries 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 query while the daemon still only wakes up every 30 seconds and writes to disk every five minutes. The data resolution of 33 queries 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.
While the 50 NREF queries are intended to be a realistic workload that could be used in an

application, the 50k and 1m tests are simple stress tests that should provoke a reaction of the DBMS. Given that a significant performance drop could only be seen with 1,000 queries per second it is believed that the overhead is tolerable for most real-life systems.
The next section will show how the analyser recommendations can help to improve the overall performance of the workload.







		Original		**Monitoring**		**Daemon**	
	Run #	absolute	query/s	absolute	query/s	absolute	query/s
50	1	8263 s	–	8279 s	–	8381 s	–
	2	8256 s	–	8290 s	–	8365 s	–
	3	8240 s	–	8278 s	–	8364 s	–
	Average	**8253 s**	–	**8282 s**	–	**8370 s**	–
50k	1	4327 s	11.55	4339 s	11.52	4362 s	11.46
	2	4320 s	11.57	4335 s	11.53	4360 s	11.46
	3	4321 s	11.57	4331 s	11.54	4360 s	11.46
	Average	**4322 s**	**11.56**	**4335 s**	**11.53**	**4360 s**	**11.46**
1m	1	731 s	1367.98	826 s	1210.65	876 s	1141.55
	2	732 s	1366.12	832 s	1201.92	893 s	1119.82
	3	734 s	1362.39	827 s	1209.18	888 s	1126.12
	Average	**732 s**	**1365.49**	**828 s**	**1207.72**	**885 s**	**1129.94**

Figure 6.4: Monitoring Tests – Results

6.3.2. Analyser Tests

The experiment in this section should show if the recommendations of the analyser are actually useful to improve the performance of a system.

Test Setup

To test the usefulness of the analyser the two Ingres instances from the previous section were used. The original instance with no code changes applied was manually optimised. The 33 indexes that were recommended by [CBTM05] were applied, statistics on all tables were created and all tables were modified to B-Tree. The second instance with the monitoring code was used to record the workload of the 50 NREF queries. The analyser client 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 optimisation. It is expected that the performance of both systems will be similar but due to the fact that the analyser recommends only a limited number of indexes the size of the resulting database should be smaller.

Analyser Report

The analysis took about 40 seconds to scan the workload of 50 NREF queries and to test possible new indexes on the DBMS. For 31 queries the analyser reported that the estimated cost values significantly differ from the actual cost to execute and suggested to run optimizedb on the database. This was affirmed in the table section of the report that stated that none of the NREF tables had statistics. In addition, all of the six tables had a high number of overflow pages and the analyser recommended to modify them to B-Tree. In total, the analyser recommended 12 secondary indexes on the database and estimated an overall performance win of only 1% and an average win per statement of 38% – which is only reflecting the ratio between estimated costs with and without the recommended indexes and not taking into account other changes such as the creation of statistics. Figure 6.5 shows the cost diagram of the ten most expensive queries of this workload.

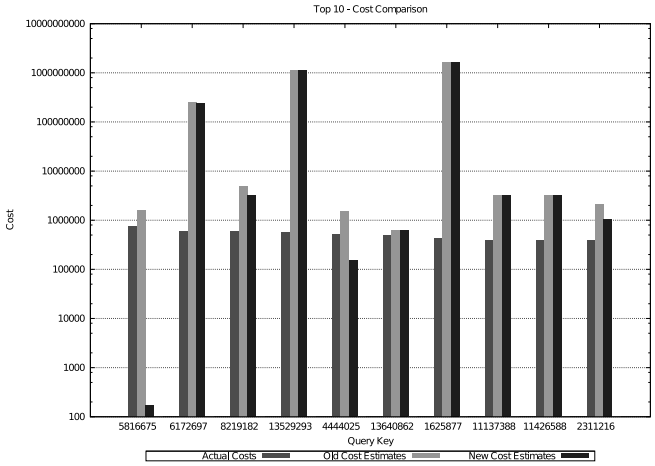


Figure 6.5: Analyser Tests - Cost Diagram

While some of the queries show a high performance win many also are expected to stay the same, i.e. will not benefit from new indexes.
To test the usefulness of the analyser all the recommended changes where implemented – all tables where modified to B-Tree, the 12 recommended indexes were created and optimizedb was executed.

Results

Figure 6.6 shows the results observed in this experiment. The manual optimisation of the NREF database grew the size of the data files from 33 to 65 gigabytes while the time to execute the query set was reduced to 4955 seconds which is a performance win of about 66%. The recommended changes of the analyser grew the size to 53 gigabytes, saving 12 gigabytes over the manual optimisation with 33 indexes. The execution time was cut down to 5130 seconds which is about 3% more than with the manual optimisation (without taking the overhead of the monitoring into account) and corresponds to an overall performance win of 61% for the given workload.

	Original		**Design Analyser**	
Run #	Non-optimised	Optimised	Non-optimised	Optimised
1	8263 s	4954 s	8279 s	5139 s
2	8256 s	4957 s	8290 s	5125 s
3	8240 s	4954 s	8278 s	5126 s
Average	**8253 s**	**4955 s**	**8282 s**	**5130 s**

Figure 6.6: Analyser Tests – Results

The analyser 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 many times is more useful than an index that has less recommendations. This – as some of the other rules – is only true for certain situations. But still, the presented results of the analyser 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 to create too many indexes that could even slow down the system. The experiment could show that the analyser was able to recommend useful changes to the physical database design with a performance win that was comparable to the manual optimisation. The recommended index set was only half as big as the reference index set saving disk space of over ten gigabytes. However, the estimates of the analyser were far off its actual optimisation results.

7. Outlook

The previous chapters have shown a basic concept and an initial implementation of a system that enhances Ingres with features for an autonomous database tuning. Chapter 6 has shown that this experimental implementation is already capable of recommending a set of changes to the physical database design that can significantly improve the overall performance of the system for a given workload. However, the proposed concept is in many ways still very simple as it currently concentrates mostly on the creation of indexes and leaves out partitions as a possible performance factor. The analysis also ignores dependencies between the various factors risking to recommend conflicting changes. This chapter will discuss possible improvements and additions to the four steps of monitoring the data in the core, collection the data in the workload db, analysing the workload and then presenting recommendations to the DBA.

7.1. Monitoring

As seen in chapter 6, the monitoring code performs well for queries that need several seconds to execute but with a high number of queries per second the overhead starts to lower the throughput of the DBMS noticeably. An improvement to the monitoring code could be the implementation of a keyed or indexed storage of the logged database objects and statements to avoid the currently used serial search within the structures (see figure 5.5). Ingres offers methods to create hash maps which could be evaluated for the use in SCM. This way, a lookup of a statement or an object could be done efficiently and could save a significant part of the overhead.
The monitoring itself needs to be expanded to include more information about the DBMS and the workload. For example, the current implementation does not include information about table partitioning. The analyser does not know if a table is partitioned or not and therefore cannot make recommendations about this physical structure. For this to add, more research is needed to understand how Ingres handles partitions and when it benefits from partitioned tables.
Although secondary indexes are already included, the monitor does not yet support multi-column indexes that span more than one attribute. Those indexes are in fact handled (and logged) as one-column indexes at the moment which could result in wrong design decisions later on.
Another valuable improvement for the monitoring would be the logging of query execution plans. Based on the currently available data, the analyser can only tell the DBA that there are expensive queries but not why. With the QEP being logged, the system could scan each tree node for the most expensive actions (sorts, joins, etc) and could give targeted recommendations to specific parts of a statement.

7.2. Data Collection

The overhead of the data collection part of the monitor daemon is not believed to be critical. The tool runs as its own process and doesn't add its overhead to the actual query processing. However, storing the data on disk can slow down the system noticeably – first, when a very large number of statements and objects need to be written to the workload DB but also when the user runs disk intensive queries where the monitor daemon disturbs the disk operation. This effect could be mitigated by moving the workload DB to a physically separate disk but it is also believed that the actual code to store the data could be improved to be more efficient.
Instead of having the montor daemon to be a stand-alone tool, it could also be included in the DBMS core, merged with the monitoring code. It needs to be evaluated what resources can be saved with this option and how it performs over the separated model.

7.3. Analysis

The analyser presented in this work is indented to be a proof-of-concept, showing a possible way to process and present the data in the workload DB. For a valuable advisor tool, it needs a more sophisticated approach of data analysis. Similar to [ZRL+04], the analyser needs to define dependencies not only between the various physical structures but between all its recommendations. Currently, all recommendations are based on a single and local decision. There is no search for reasonable recommendations but only plain execution of rules. With a dependency graph the analyser could actually search for an optimal set of recommendations that would fit better to the given workload. As mentioned before, the analyser also needs to include table partitioning and multi-column indexes to benefit from all available physical structures in Ingres.
Currently, the analyser can only report estimated performance wins based on the creation of virtual indexes. The experiment in section 6.3.2 has shown that these figures can be far off the real win. In this case the win was expected to be only at 1% while it was actually at over 60% – the DBA would probably never implemented the changes with such a low estimate. Even worse, the estimate could be high but the changes could even slow down the system. The analyser doesn't take into account storage structures or table statistics that can greatly influence the system performance. But it also ignores the costs of changes in form of time and disk space requirements. All recommended changes need to be connected to cost and win values so that the DBA can decide whether implementing the change is beneficial or not. It needs to be evaluated what costs and wins can be taken from the internal optimiser and what needs to be provided by an external cost model.
The current implementation of the analyser only processes selects and discards other statements types, however, the Ingres optimiser doesn't differentiate between access paths for selects or updates and deletes. It is believed that the analyser can easily be changed to include other statement types as well.
A design decision for the proposed concept was to monitor a real workload to allow an accurate analysis but the DBA could also wish to analyse a theoretical workload so that expanding the analyser with an import interface would be useful. This way, the DBA could prepare the system for a future workload. However, static statements with no real cost values attached cannot benefit from all of the analyser rules and the recommended changes may not as good as for a real workload.

7.4. Presentation

The report of the analyser in its current form is only a list of information and recommendations put on a plain HTML page. The report needs to become more user-friendly in terms of clarity and navigation. While creating the report as HTML has advantages such as its platform independence and availablility over the network, the report could also be presented by a graphical desktop application that would even allow interaction with the system such as the automatic implementation of recommended changes.
The diagrams that are currently shown in the report need to be interpreted by the DBA. The report could include written explanations and even interpretations of the values seen in the diagrams. These could range from simple observations such as a reached limit or threshold to more predictive statements that warn of possible problems in the future.
A next step could then be the autonomous implementation of changes without interaction of the DBA. For this to achieve, more efficient ways of creating physical structures are needed to allow creating them online while the system is in normal operation without slowing down or even blocking user traffic.

8. Conclusion

Studies such as [Hab03] or [IDC07] state that today the smallest fraction of overall costs to operate an IT system are costs for software and hardware. The by far 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 effort of maintenance as low as possible to minimise the required amount of human work. Following the concept of autonomic computing ([KC03]), software systems become more and more self-sustaining and independent of human intervention. Especially in the area of relational database management systems the topic of self-management and self-optimisation is an area of high interest because of the ever growing complexity of these systems. Not only the number of features, settings and possibilities to tune a DBMS is increasing but also the data volume and logical design of databases is becoming bigger and more complex. Database administrators are faced with automatically created database schemes where the implementation of an optimal configuration is no longer a trivial task. At the same time, even the amount of data in end-user and desktop applications is reaching a point where more and more software developers decide to embed a DBMS into their application. These hidden systems need to maintain themselves as they would become slower over time when the initial configuration no longer suits.

This work has provided background knowledge of factors that influence the performance of a DBMS such as the physical structures of secondary indexes or partitions but also configuration settings of the system. It has also shown performance indicators that can be used by the DBA to identify problems. The principles of the autonomous tuning of databases were then presented as a solution to support or even replace the DBA by monitoring the same indicators and applying changes to the system automatically.

Together with a list of past and ongoing research in the area of autonomous database tuning this work has discussed and tested the implementation of tuning features in a selection of modern commercial DBMS.

The work has then presented a concept for an autonomous tuning in the relational database management system Ingres that had not yet the features of monitoring, collecting and analysing data to improve the performance of the system. For an evaluation, the proposed concept was implemented in Ingres by expanding the DBMS core to monitor the workload on the system. The continuous data collection is done through a monitoring daemon that is reading the data from the DBMS and persistently storing it in a workload database. The third part of the implementation is an analyser that is scanning the collected data to find possible performance improvements but also to present reports on the current and past system state that should allow problem prediction.

The experiments in this work could show that the overhead added by the monitoring is negligible for complex queries that take several seconds or more to execute. The impact becomes visible for subsecond queries where – at over 1000 queries per second – the throughput was lowered by about 17%.

To test the usefulness of the concept the analyser was executed on a workload to see if the

recommended changes could actually improve the performance of the system. The experiment could show that the analyser results in a performance win that is comparable to a manual optimisation while using less disk space.
Throughout the work it was noted that the presented implementation and the concept of the analyser are still experimental and should be considered as a proof-of-concept rather than ready for production use. Possible ways to improve the existing concept were presented as well as auto-tuning features that can be built upon the concept.
It is believed that implementing this concept to Ingres is of great value for the DBMS not only to keep up with other modern systems but also to pave the way for ongoing research that eventually leads to an autonomic relational system.

A. Listings

```
1   typedef struct _SCM
2   {
3       SCM_STATEMENT            *statements[MAXMONITOR];
4       i4                        cur_stm_idx;
5       SCM_WORKLOAD             *workload[MAXWORKLOAD];
6       i4                        cur_wkl_idx;
7       SCM_TABLE                *tables[MAXMONITOR];
8       i4                        cur_tab_idx;
9       SCM_ATTRIBUTE            *attributes[MAXMONITOR];
10      i4                        cur_atr_idx;
11      SCM_INDEX                *indexes[MAXMONITOR];
12      i4                        cur_idx_idx;
13      SCM_REFERENCE            *references[MAXMONITOR*2];
14      i4                        cur_ref_idx;
15      SCM_STATISTICS           *statistics[MAXMONITOR/2];
16      i4                        cur_sts_idx;
17  }   SCM;
```

Figure A.1: SCM (scm.h)

```
1       typedef struct _SCM_STATEMENT
2       {
3           char        database[DB_MAXNAME];
4           u_i4        query_key;
5           char        query_text[MAXQUERYLEN];
6           u_i4        frequency;
7           i4          time;
8       }   SCM_STATEMENT;
```

Figure A.2: SCM_STATEMENT (scm.h)

```
typedef struct _SCM_WORKLOAD
{
    char        database[DB_MAXNAME];
    i4          idx;
    u_i4        query_key;
    u_i4        opf_cpu;
    u_i4        opf_dio;
    u_i4        qef_cpu;
    u_i4        qef_dio;
    u_i4        est_cpu;
    u_i4        est_dio;
    u_i4        pages_touched;
    i4          time;
    i4          wctime;
}   SCM_WORKLOAD;
```

Figure A.3: SCM_WORKLOAD (scm.h)

```
typedef struct _SCM_TABLE
{
    char        database[DB_MAXNAME];
    i4          table_id;
    char        name[DB_MAXNAME];
    u_i4        frequency;
    u_i4        est_cpu;
    u_i4        act_cpu;
    u_i4        est_dio;
    u_i4        act_dio;
    u_i4        est_tup;
    u_i4        act_tup;
    i4          structure;
    u_i4        data_pages;
    u_i4        overflow_pages;
    i4          time;
}   SCM_TABLE;
```

Figure A.4: SCM_TABLE (scm.h)

```
typedef struct _SCM_ATTRIBUTE
{
    char        database[DB_MAXNAME];
    i4          attribute_id;
    char        name[DB_MAXNAME];
    i4          table_id;
    u_i4        frequency;
    i4          statistics;
    i4          time;
}   SCM_ATTRIBUTE;
```

Figure A.5: SCM_ATTRIBUTE (scm.h)

```
1   typedef struct _SCM_INDEX
2   {
3       char        database[DB_MAXNAME];
4       i4          index_id;
5       char        name[DB_MAXNAME];
6       i4          table_id;
7       i4          attribute_id;
8       u_i4        frequency;
9       i4          structure;
10      u_i4        data_pages;
11      u_i4        overflow_pages;
12      i4          time;
13  }   SCM_INDEX;
```

Figure A.6: SCM_INDEX (scm.h)

```
1   typedef struct _SCM_REFERENCE
2   {
3       char        database[DB_MAXNAME];
4       u_i4        query_key;
5
6   #define         SCM_TYPE_TABLE  0
7   #define         SCM_TYPE_ATTR   1
8   #define         SCM_TYPE_INDEX  2
9
10      i4          object_type;
11      i4          object_id;
12      i4          table_id;
13      i4          time;
14  }   SCM_REFERENCE;
```

Figure A.7: SCM_REFERENCE (scm.h)

```
1   typedef struct _SCM_ANALYZE
2   {
3       u_i4        cpu;
4       u_i4        dio;
5       u_i4        pages_touched;
6       char        vindexes[MAXQUERYLEN];
7   }   SCM_ANALYZE;
```

Figure A.8: SCM_ANALYZE (scm.h)

B. Debugging Ingres

Query Execution Plans

QEPs can be visualised in Ingres via the command line terminal monitor or the Windows tool Visual DBA. On the command line a QEP is printed when executing the command

```
set qep
```

The QEP will show how Ingres decided to join tables together, what indexes will be used and what the estimated costs and cardinalities are. A QEP can help to identify problems with long-running queries. An explanation of how to interpret a QEP can be found in [Cora].

Stop Before Execution

For the examination of the QEP the result of the query may not be of interest. In such a case the actual execution of the query can be left out by executing

```
set optimizeonly
```

Trace Points

Trace points are used in Ingres for debugging purposes. Setting a trace point during runtime of the DBMS triggers the execution of code that wouldn't be executed otherwise. Trace points are primarily used for debug output but some of the existing trace points can also change the behaviour of the system. A trace point is activated by the command

```
set trace point IDENTIFIER
```

where *IDENTIFIER* is the name of the trace point. Trace points are not supported by Ingres and therefore not documented at all – neither their names nor the meaning of their output. What trace points exist may only be found out by searching the internet or looking at the code directly. Trace points used in this work are explained where needed.

Trace File

Most of the debug output of the DBMS is sent to a local file when the trace file directive has been set. With the command

```
set trace output '/path/to/file'
```

Ingres will create the file and fill it with debug information coming e.g. from trace points.

Bibliography

[BC06] Nicolas Bruno and Surajit 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*, pages 499–510. VLDB Endowment, 2006.

[BPS90] Elena Barcucci, R. Pinzani, and Renzo Sprugnoli. Optimal Selection of Secondary Indexes. *IEEE Trans. Softw. Eng.*, 16(1):32–38, 1990.

[Bro93] David Brower. The INGRES Management Architecture. ASK Group Inc. (Unpublished) – Presented at INGRES World, 1993.

[CBTM05] Mariano P. Consens, Denilson Barbosa, Adrian Teisanu, and Laurent Mignet. Goals and benchmarks for autonomic configuration recommenders. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 239–250, New York, NY, USA, 2005. ACM. `http://www.cs.toronto.edu/~consens/tab/`. Link visited 30 Apr 2008.

[CH76] Arvola Y. Chan and Michael Hammer. Index Selection in a Self-Adaptive Relational Data Base Management System. Technical report, Cambridge, MA, USA, 1976.

[CN98] Surajit Chaudhuri and Vivek Narasayya. AutoAdmin "what-if" index analysis utility. *SIGMOD Rec.*, 27(2):367–378, 1998.

[CN07] Surajit Chaudhuri and Vivek R. Narasayya. Self-Tuning Database Systems: A Decade of Progress. In *VLDB '07: Proceedings of the 33nd international conference on Very large data bases*, pages 3–14. VLDB Endowment, 2007.

[Cod70] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, 1970.

[Cora] Ingres Corportation. *Ingres 2006 Release 2 Documentation*. Documentation online at `http://docs.ingres.com`. Link visited 16 June 2008.

[Corb] Oracle Corportation. *Orcale Documentation Library 11g Release 1*. `http://www.oracle.com/pls/db111/homepage`. Link visited 2 May 2008.

[DDD+04] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. Automatic SQL Tuning in Oracle 10g. In *VLDB '04: Proceedings of the 30nd international conference on Very large data bases*, pages 1098–1109. VLDB Endowment, 2004.

[FON92] Martin R. Frank, Edward Omiecinski, and Shamkant B. Navathe. Adaptive and Automated Index Selection in RDBMS. In *EDBT '92: Proceedings of the 3rd International Conference on Extending Database Technology*, pages 277–292, London, UK, 1992. Springer-Verlag.

[GA08] Kareem El Gebaly and Ashraf Aboulnaga. Robustness in automatic physical database design. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 145–156, New York, NY, USA, 2008. ACM.

[Gra00] Goetz Graefe. Dynamic Query Evaluation Plans: Some Course Corrections? *Bulletin of the Technical Committee on Data Enigneering*, 23(2):3–6, 2000.

[Hab03] Gottfried 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.

[IDC07] IDC. Selling to Your C-Level Executives, 2007. `http://www.microsoft.com/windowsserver/compare/ReportsDetails.mspx?recid=3`, page 5. Link visited 16 Aug 2008.

[Ink03] Douglas N. Inkster. OPF Enumeration Enhancement. Ingres Corporation (Unpublished), 2003.

[Ink04] Douglas N. Inkster. Ingres Optimizer Facility. Ingres Corporation (Unpublished), 2004.

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[Koo80] Robert Philip Kooi. *The Optimization of Queries in Relational Databases*. PhD thesis, 1980.

[LL02] Guy M. Lohman and Sam S. Lightstone. SMART: making DB2 (more) autonomic. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 877–879. VLDB Endowment, 2002.

[Lue06] Martin Luehring. Anfragengetriebene Auswahl von Soft-Indexen. Master's thesis, Technische Universität Ilmenau, 2006.

[ML05] Anolan Yamilé Milanés and Sérgio Lifschitz. Design and Implementation of a Global Self-tuning Architecture. In *SBBD*, pages 70–84, 2005.

[Pee07] Jeremy Peel. Ingres: Re-Opened, Innovating and in Business. *Datenbank-Spektrum*, 7(22):13–19, 2007.

[RS91] Steve Rozen and Dennis Shasha. A Framework for Automating Physical Database Design. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 401–411, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[SB03] Dennis Shasha and Philippe Bonnet. *Database tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[Sch06] Karsten Schmidt. Automatische Erstellung von Soft-Indexen in PostgreSQL. Master's thesis, Technische Universität Ilmenau, 2006.

[SGS03] 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*, pages 1129–1132, 2003.

[SGS04] K. Sattler, I. Geist, and E. Schallehn. Autonomous Query-driven Index Tuning. In *IDEAS'04, Coimbra, Portugal*, pages 439–448, 2004.

[Sti04] Mario Stiffel. Anfragengetriebenes Index-Tuning in PostgreSQL. Master's thesis, Technische Universität Ilmenau, 2004.

[Sto86] Michael Stonebraker, editor. *The INGRES Papers: Anatomy of a Relational Database System*. Addison-Wesley, 1986.

[Thi08] Alexander Thiem. Implementing Virtual Indexes in Ingres 2006. Technical report, Technische Universität Ilmenau, 2008.

[WIG+06] Edward Whalen, Victor Isakov, Marcilina Garcia, Burzin Patel, and Stacia Misner. *Microsoft SQL Server(TM) 2005 Administrator's Companion (Pro - Administrator's Companion)*. Microsoft Press, Redmond, WA, USA, 2006.

[ZRL+04] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. Db2 design advisor: integrated automatic physical database design. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1087–1097. VLDB Endowment, 2004.

List of Abbreviations

CMIP . . . . . Common Management Interface Protocol.

DBA . . . . . DataBase Administrator.

DBMS . . . . . DataBase Management System.

DMF . . . . . Data Management Facility.

ESQL/C . . . . . Embedded SQL for C.

GCF . . . . . General Communications Facility.

IMA . . . . . Ingres Management Architecture.

IMADB . . . . . IMA DataBase.

ISAM . . . . . Indexed Sequential Access Method.

MIB . . . . . Management Information Base.

MO . . . . . Management Object.

OPF . . . . . Optimiser Facility.

PSF . . . . . ParSer Facility.

QEF . . . . . Query Execution Facility.

QEP . . . . . Query Execution Plan.

QSF . . . . . Query Storage Facility.

RDF . . . . . Relation Description Facility.

SCF . . . . . System Control Facility.

SCM . . . . . SCF Monitor.

SNMP . . . . . Simple Network Management Protocol.

TID . . . . . Tuple IDentifier.

List of Figures

Thesen

1. Die stetig ansteigende Komplexität von Datenbank Management Systemen führt dazu, dass eine manuelle Wartung dieser Systeme zu einer wachsenden Herausforderung für Administratoren wird.
2. Ein Großteil der aufzubringenden Zeit wird für die Überwachung des Systems benötigt.
3. Der DBA hat selten die Möglichkeit die Anfragen an die Datenbank zu ändern und ist stattdessen auf das physische Datenbankdesign begrenzt.
4. Durch das Konzept des Autonomous Tuning wird das DBMS selbstständiger und übernimmt Aufgaben, die der DBA zuvor manuell durchführen musste.
5. Auto-Tuning basiert auf den Schritten der Überwachung, dem Sammeln und der Analyse von Daten. Analyseergebnisse werden entweder automatisch angewandt oder dem DBA präsentiert.
6. Die Erweiterung des Datenbank Management Systems Ingres um Fähigkeiten des Auto-Tunings ermöglicht DBAs eine effizientere Administration ihres DBMS.
7. Mit dem Ingres Design Analyser wird das Anfrageverhalten an eine Datenbank aufgezeichnet und zusammen mit Systemstatistiken über einen längeren Zeitraum gespeichert.
8. Eine aktive Überwachung der Daten ermöglicht die Alarmierung des DBAs im Falle einer akuten Problemsituation.
9. Die manuell gestartete Analyse der aufgezeichneten Daten erkennt automatisch Probleme und findet Lösungsmöglichkeiten.
10. Mit hypthetischen physischen Strukturen kann der interne Optimierer des DBMS genutzt werden, um potentielle Verbesserungen auf ihre Wirksamkeit zu testen.
11. Änderungsvorschläge werden dem DBA visuell und textuell präsentiert. Er entscheidet, welche Änderungen vorgenommen werden.
12. Mit effizienten Analyseverfahren kann die Problemerkennung und Lösungsfindung auch automatisch während des Betriebs geschehen. Das System kann Änderungen dann selbstständig anwenden und dem DBA diese Arbeit abnehmen.

Ilmenau, den 04. September 2008

Alexander Thiem