TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Towards database performance patterns (extended version)

Citation for published version (APA):
Aerts, A. T. M., Molengraft, van de, W. T., & Snijders, J. (2008). *Towards database performance patterns (extended version).* (Computer science reports; Vol. 0812). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/2008

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 16. Nov. 2023

TOWARDS DATABASE PERFORMANCE PATTERNS

- EXTENDED VERSION -

A.T.M. Aerts[1], W.T van de Molengraft[1], and J. Snijders[2]
[1]Technische Universiteit Eindhoven, Eindhoven, The Netherlands
[2]Quinity, Utrecht, The Netherlands

ABSTRACT

We explore the use of patterns as an effective means to deal with recurrent performance issues in software development. The benefit of this use is that one can make argued design choices that have satisfactory performance properties. We give a definition of a performance pattern, and study the structure of database performance patterns in detail. We give an illustrative example of the latter kind of pattern.

KEYWORDS

Performance engineering.Performance patterns, database performance, application design, software engineering,

1. INTRODUCTION

In order to solve recurrent design problems in software development, design patterns are used. Design patterns originate from the architectural world [AIS1977, Ale1979] and have been made popular for software engineering by Gamma et al. [GH1995]. In the latter case patterns are used during the software implementation phase. For this reason, they can be called technical design patterns. Many technical design patterns are available nowadays (see, e.g., [JBP]) and many developers know how to use them.

[GH1995] states what a pattern does: ``A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is customized and implemented to solve the problem in a particular context." Since Gamma et al. published their book the concept of patterns as mechanisms has spread to other domains such as business processes [Fow1996] and workflows [RH2004, WfP]. It is clear from the definition above, that patterns are abstract enough to apply to many design domains, but specific enough to provide hands-on guidance to designers and architects. Patterns also provide a vocabulary for developers to efficiently describe their solution.

The question now arises whether a pattern based approach will also work for the performance aspect of a software system? A software development project is only successful when both functional and performance requirements have been met without exceeding the amount of money and time available. Therefore, performance problems can cause a total project to fail.

Performance is frequently managed reactively: the fix-it-later approach. Typically, functionality is implemented first. During the testing phase the performance requirements are verified. The discovery of performance problems leads to changes in the design or implementation. The drawback of the fix-it-later approach is that an unpredictable number of performance problems is discovered in the testing phase, that is quite close to the project deadline. Just as the number, the impact of those problems is not predictable either. Some problems may be solvable by small adjustments to the implementation. Others might need a complete redesign. As a result the development cycle starts all over again without the guarantee that all problems will be solved next time. Therefore the fix-it-later approach introduces risks for both budget and time. The way out of this appears to

be to learn from prior experience, and proactively apply these lessons at design time for systems where performance is of great importance. The issue now is how to make the lessons learnt available. The approach we propose in this paper is the usage of performance patterns.

We'll first give a brief review of previous research on performance patterns, and point out some gaps. Then we discuss the structure of a performance pattern. After this we discuss several issues arising in the case of a particular performance pattern: the database performance pattern, and provide an illustrative example of such a pattern. Finally we discuss some issues about the usage of patterns.

2. PREVIOUS RESEARCH ON PERFORMANCE PATTERNS

Smith and Williams [SW2002] introduced performance patterns as a technique to improve the software development process. Performance patterns describe generic solutions for common performance problems and can be applied during the functional and technical design phases and the implementation phase to prevent performance problems during the testing and production phase. By reusing existing solutions, time and money can be saved. In this way they prevent designers and developers from introducing performance problems.

Keller wrote about design patterns for an object / relational layer in a series of articles [KC1997, KEL1997, KEL1998]. An object / relational layer is the layer in between a relational database and an object oriented programming language. His patterns both address functionality and performance, and may be called performance aware patterns. Some are more related to performance and some to functionality. They all provide solutions that can directly be applied in several situations that occur in software development. Although Keller uses a semantic point of view rather than a technical one, the idea of combining performance with functionality into building blocks, which can almost directly be applied during software development, is quite useful.

Keller found a nice way to deal with performance in his patterns. First the problem and solution are explained. Afterwards he identifies a number of forces which describe when the solution can be applied. Amongst those forces are reading speed, writing speed and scalability. Sometimes a trade-off between two aspects is described (for example disk space vs. reading speed). As a result, the biggest advantage of performance aware patterns is their usability. There is a concrete solution and the most important consequences are presented so that developers can choose whether or not to use the pattern. Although the structure of the patterns might be useful, the approach to finding and using the patterns is not described and the focus is on functionality rather than performance.

Smith and Williams [SW2002] present a catalog of patterns which do not depend on a specific domain. Their patterns are far more general than the performance aware design patterns mentioned above and based upon performance principles [SMI1990], such as the centering principle: ``Identify the dominant workload functions and minimize their processing''. Their patterns complement and extend the principles. Each performance pattern is a realization of one or more of the performance principles. Performance patterns present common solutions, which can be applied in a number of situations. In addition to patterns, they also define a number of anti-patterns [SW2002b] that identify common solutions, which are likely to cause performance problems.

The major advantage of the performance patterns of Smith et al. is that they are very general. Implementation details and domain knowledge have been abstracted from. However, the solutions described by the patterns are applicable in a wide range of situations and therefore more like strategies. Due to the lack of technical details, moreover, a designer has take several steps to bridge the gap between the pattern and its application in a practical situation. No quantitative measures can be given for the solutions to support an argued choice between possible alternatives. For comparison, the gap between the pattern and its use is much smaller for the technical design patterns of Gamma et al., which increases their usability.

To illustrate the level of abstraction, we compare performance patterns to the technical design patterns of Gamma et al. Performance patterns describe strategies to avoid performance problems whereas design patterns address common implementation solutions. Therefore a design pattern can be used, and indeed is needed, to implement the strategy presented by a performance pattern. An example of a performance pattern is the Fast Path pattern [SW2002]. It describes how response times can be improved by reducing the amount of processing

required for dominant workloads. The Fast Path performance pattern can be implemented using the proxy design pattern [GH1995].

Although the patterns of Smith are very general and can be applied in many different contexts, it is not always clear when a pattern applies to a situation. Due to this gap between patterns and reality, usability of the patterns decreases. A constant awareness of performance and some experience in applying the patterns is required for developers to efficiently use the performance patterns.

Finally, implementation solutions describe how performance problems can be fixed during implementation. They are needed to fix problems that appear for the first time during implementation, or in a later stage, after the system has been in function for a while and the performance requirements have evolved differently than anticipated at design time. One example of such a solution is tuning. Tuning is part of reactive performance management and is part of the fix-it-later method. As fix-it-later is widely used, books about tuning are widely available [GOL1993, SHA2003]. Therefore tuning is of less interest for this article.

2.1 Current status of performance patterns

The major problem with previous attempts to introduce performance patterns was either a gap between patterns and their application in a practical situation or the focus on functionality rather than performance. However, the idea of using generic solutions for common performance problems works for both performance aware design patterns and performance patterns. Although performance problems are very common in software development and both approaches seem to work, very little attention has, to our knowledge, been paid to performance patterns in literature. This has the consequence that some important aspects are currently missing. Since the number of patterns available is very small, we need a way to identify new performance patterns. Unfortunately so far no method to approach this issue is available. Then, given the availability of patterns, designers and developers need to be motivated and trained to use performance patterns. In this article we analyze the way in which performance patterns can be described to make them useful. We decided to focus on a special type of performance patterns, namely database performance patterns to determine how performance patterns can be used. Therefore we must investigate the differences between normal performance patterns and database performance patterns as well.

3. PERFORMANCE PATTERNS

We adapt the idea of patterns to the area of software performance: “Performance patterns are patterns which document generic solutions for common performance problems.” Although the definition above specifies what a performance pattern does, it does not state how. This leaves open the level of abstraction at which performance is addressed. Consider the approach used by Smith and Williams [SW2002]. They identified seven concepts, which result in a better performance once applied. Those concepts have been translated into performance patterns. Consider for example the Alternative Routes pattern: “Spread the demand for high-usage objects spatially, that is, to different objects or locations.” It describes a performance solution that can be reused without specifying the functional usage of the demands. Therefore these performance patterns are very general and they can be seen as best practices.

Another way to describe performance patterns is based on the idea that an application consists of several building blocks. This is very similar to the usage of components in software development [MIL2002]. Components implement some functionality, but they are general enough to be reused. A component is often configurable by means of parameters. If we apply the idea of components to earlier stages of the development, we get a new approach to performance patterns. Although Keller [KC1997, KEL1997, KEL1998] uses a very similar approach, his patterns are more like technical design patterns that happen to discuss performance as well. They do not focus on performance, however. Since they lack quantitative measures of performance, they cannot be used to compare solutions to each other. We have not seen any other performance patterns using the building blocks approach, so we must find a way to deal with the additional complexity of the performance parameters

ourselves. Building blocks are much closer to reality than best practices. They can even be instances of best practices. By adding extra detail to the patterns, we can be more specific about performance and the gap between reality and patterns becomes smaller. Furthermore these building blocks are very similar to technical design patterns, so developers, who are familiar with those, can use the building blocks easily. As building blocks fit perfectly in our development process, we will focus on them rather than best practice patterns.

3.1 Ingredients of a performance pattern

In order to describe our performance solutions in a structured manner, we define a pattern template. Several pattern templates exist for functional [SNI2004] and technical design patterns. We start creating a suitable template using one proposed by Smith and Williams [SW2002] that contains all essential parts of a performance pattern. Since our patterns are more specific than the ones defined by them, we adapted the way of describing consequences as presented by Keller [KC1997]. Furthermore the use of the building block approach results in a functionality-based problem description that may have several solutions depending on the performance requirements. We describe two pattern templates based on the number of solutions described by the pattern.

3.1.1 Single solution structure

The single solution structure describes a template for performance patterns that consist of one solution which solves the entire problem. Each performance pattern consists at least of a situation and a solution. For usability reasons, the pattern is given a name, an example is added to the problem, the consequences of applying a solution are described and relations to similar problems and solutions are discussed. An outline of the basic pattern structure is presented in Figure 1.

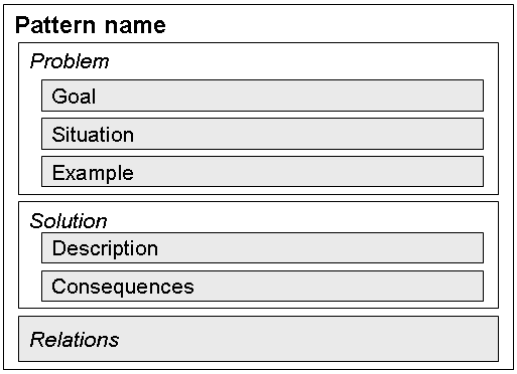


Figure 1 Basic performance pattern structure

The exact meaning and contents of all elements is discussed below:

Name: In order to talk about a pattern, it must have a name. The pattern name covers the contents of the entire pattern and can therefore be used to select patterns as well. If necessary a short explanation of the name can be added as well.

Problem: The problem consists of a goal, a situation and an example. The goal describes the purpose of a pattern. It introduces the design problems that are addressed and consists of a few sentences at most. The situation describes some stage in the development of software. It must be written such that developers can verify whether or not they can apply the pattern to their own problem. Finally, the example provides a situation in which the problem occurs.

Solution: The solutions consists two elements: a description of the solution itself and an analysis of the consequences. These consequences are trade-offs that the user of the pattern should make when applying

it. At least the consequences for performance aspects must be covered. Consequences that influence other quality attributes of the software (e.g. maintainability or portability) can be described as well.

Relations: The relations section discusses any relations between the pattern itself and other known problems and solutions (e.g., other patterns, algorithms). The relation section can be used by developers in order to find additional information or to find alternative solutions.

3.1.2 Multiple solutions structure

Some properties of a situation change over time. One can think, in the case of a database performance problem, of the number of rows in a table. We cannot choose a solution for a problem unless we make assumptions for the values of these properties. Since these values may change over time, we introduce multiple solutions which specify solutions that solve the problem under certain conditions. By introducing multiple solutions, we can describe strategies to solve a range of problems structurally rather than describing solutions for instances of problems separately.

An updated overview of the pattern structure is displayed in figure 2. In addition to the basic solution structure, each solution consists of a notion of applicability and optionally relations to other solutions. Furthermore we add a summary of all solutions to allow the savvy user to quickly select the right solution.

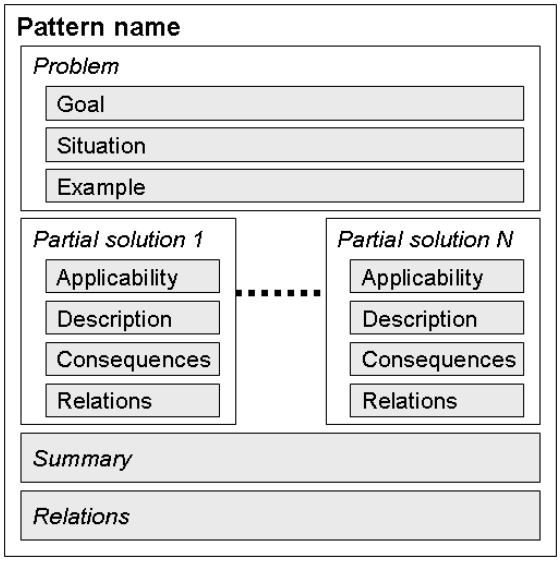


Figure 2 Template for multiple solutions structure

4. DATABASE PERFORMANCE PATTERNS

Database performance patterns are special performance patterns. They only address database actions like search queries. This section extends the previous one by identifying the database specific aspects of database performance patterns.

Designers and developers perform several types of actions on a database. They can search for information or update information in a relational database using an SQL query for instance. The most important aspects to determine database performance are the structure of the data and the structure of the query. If we look at this SQL

query as a database action, we can define database performance as the degree to which actions performed on a database system meet their objectives for timeliness given the structure of the data.

In this context we can measure response times of single actions to express performance. We assume a relational database that takes an SQL query and returns a table of results. The response time of an action is the time that elapses between the moment a query arrives at the database and the table of results is sent back. The biggest problem with the response times described above is that many different measurement methods can be used. One can for instance choose to measure the response times locally using the database itself to clock the time elapsed between arrival of a query and sending of the results. Alternatively, one can use a tool which communicates with the database via a database driver. Although the use of a driver is more realistic, the network between the tool and the database system and the driver may influence the measurement. It is important to choose an appropriate way for each specific situation.

4.1 Performance parameters

Performance patterns are intended to define solutions that perform well. Therefore we must include all characteristics of a solution that influence its performance into a performance pattern. We call these characteristics performance parameters. The settings for these parameters determine a particular solution.

Unlike functional and technical design patterns, performance solutions extend beyond the first three development phases (i.e. functional design, technical design and implementation). Some characteristics of the usage of a solution can influence performance as well. These characteristics make performance dependent on its environment even after the implementation phase. A solution can for example perform well if the number of rows in a table is small, but as the amount of rows increases, performance may get worse. We distinguish between fixed and environment dependent performance parameters.

4.1.1 Fixed performance parameters

Fixed performance parameters are aspects of the functional design, technical design and implementation phase of the software development process that influence the performance of a solution and cannot be changed during the production and testing phase. By describing how to build a solution during these three phases, we can fix their influence. New performance parameters can be found in literature about database tuning [COR1993, SHA2003] and by analyzing the relation between the different aspects of the three development phases and the way queries are processed. A list of these parameters is presented below.

RDBMS (Relational Database Management System): Although the steps in query processing are roughly the same for each RDBMS, the efficiency of a component in one RDBMS may differ from others. Performance is influenced by optimizer decisions and the way data is stored on disk for instance. Furthermore some databases offer specific solutions to improve performance or add extra features. This is how these software products distinguish themselves. For instance, hierarchical queries are supported in Oracle SQL [GEN2001] but not in PostgreSQL [POS2006].

OS (Operating System): An RDBMS does not work without an OS. The OS provides network protocols, memory management, and much more. The database process may be interrupted by another process if the OS decides that this other process has a higher priority for instance. Choosing another OS can therefore influence performance.

Database hardware: This is the hardware on which the database system is located. As can be seen from the specifications of the Transaction Processing Councel benchmarks [TPC], the choice of the combination of hardware, OS, and RDBMS is a delicate one with mutual dependencies.

SQL Query formulation No perfect query optimizer exists and the way an SQL query is formulated influences the time it takes the DBMS to answer it. Some database management systems offer a better optimizer than others. Query optimization is still a thriving research area [SE1979, CH1998, CB2007]. The proper query formulation therefore is database dependent, since different database systems will transform the

same SQL query into different access plans (which specify the procedures by which the result table is constructed).

Data model: The data model is part of the functional design phase. It describes how data is organized in tables and how those tables relate to each other. Performance depends on the structure of the data, which is represented by the data model. We can for instance decide to denormalize to improve read performance of a specific action. The consequence of denormalization is a loss of write performance, so we can make performance decisions while designing the data model.

Indexes: Indexes are ancillary data structures that improve read performance, but are not functionally necessary. Therefore they are part of the technical design. Without indexes, we would (worst case) have to scan an entire table to find one single record. Several types of indexes exist. Each type of index has its own advantages with respect to read performance.

Next to the improvement of the read performance, indexes may decrease performance as well. Indexes store redundant information and therefore they take additional storage space on disk and in the cache. Because of this, data may no longer be cached and therefore the read performance of an arbitrary solution can deteriorate. Furthermore the introduction of indexes leads to additional updates and therefore results in a deterioration of the write performance.

4.1.2 Environment dependent performance parameters

Environment dependent performance parameters are aspects of the usage of the solution that influence its performance. Although we know a range of possible parameter values during the design and implementation and we can take that into account when choosing a solution, we cannot determine one exact value for performance due to these external parameters. The existence of environment dependent parameters makes it hard to describe database performance patterns since we cannot evaluate any possible combination of parameter values. We identified the environment dependent performance parameters listed below.

Data statistics: A cost based optimizer estimates the costs for a query using statistics about the data in a database. These statistics are based on the amount of data and the availability of auxiliary structures in the database itself. As the amount and the composition of the data changes, the performance of a solution changes.

Database and OS cache: Both OS and RDBMS keep results in cache so that frequently used results can be accessed directly instead of retrieving them from disk or computing them anew. This improves performance. An RDBMS caches frequently used results from queries and from the optimizer. When a query is submitted sufficiently frequently, the RDMBS may recognize this fact and reuse a previously computed access plan or the access plan of a similarly structured query.

Also query results are stored in cache by the database. If many queries are applied to the same table, the same data must be accessed from disk each time. By storing the most frequently used data in memory database performance can be improved significantly. Yet another type of cache is the OS cache. The OS cache is very similar to the database cache. It stores frequently accessed blocks from disk in memory. The OS makes its own choices about caching and the database cannot directly manipulate the OS cache.

Disk fragmentation: Database systems use the hard disk to store and access information. Besides the hardware aspects like the number of cycles per minute and cache size of the hard disk, the way data is stored influences performance as well [SKS2005]. If the data requested is not clustered in an easily accessible part of the disk, it takes more time to read all information.

System load: An increase of the system load can influence performance in many ways. If the system load is high for example, a query may have to wait some time before resources are available to execute it. Furthermore the number of context switches (from activating and deactivating processes) increases and therefore performance decreases even more.

Infrastructure: Depending on the infrastructure, external aspects can influence the performance as well. Suppose that the database server process is located on the same (hardware) server as the application

server process. Then the communication speed increases (no network delays), but the amount of resources must be shared amongst the database server and application server.

The values of each of these parameters may change over time. In order to allow the pattern user to choose the right solution, a range of parameter values has to be tested for each solution for each applicable environment dependent performance parameter. As a result we must describe these values and the values for which the solution was tested in the applicability, consequences and summary sections of our pattern.

4.2 Usability and limitations

The major difference between database performance patterns and functional and technical design patterns is that the performance of a solution is environment dependent. A change of environment dependent performance parameters may result in a change of the performance of a solution after implementation. Although we can specify some values for which we know that the solution is likely to perform well, we cannot take any possible combination of parameter values into account at design time when creating a pattern. This means that unexpected things still can happen to solutions after implementation.
Although this might seem to be a big problem, we can deal with it. Our problems are based on many known situations in real life and therefore we can establish that the solutions will work in most situations. Occasionally we end up in a situation that a solution does not perform according to the description in the pattern. This situation was not explored before and also without a pattern we would have had to deal with it. In both cases we decided on a badly performing solution and we need to find another one. We can use this experience to improve the performance pattern and thereby prevent the problem from happening again.

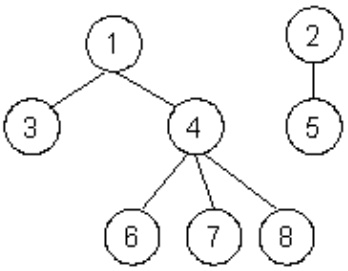
5. EXAMPLE PATTERN: SUBTREE

We illustrate the use of database performance patterns by means of a number of aspects of the database performance pattern, dealing with the subtree problem. Note that this description is illustrative. A lot more detail could be added, tests as well as alternatives. However, the present example should give an acceptable impression of what information is recorded. We use the template from Figure 2.

5.1 Problem

Goal: The goal is to retrieve a list of entities from a hierarchical data structure, which are located in the subtree of a given root entity.
Situation: A collection of entities is ordered in a number of hierarchies. We want to store these entities in a relational database such that we can determine a list of entities located in the subtree of a given root entity. An example of such a collection of entities and the corresponding subtrees can be found in Figure 3.
Example: Users are structured in a hierarchy. A user *U* is allowed to view his own information and information of users who are placed below him in the hierarchy. In order to verify that *U* is allowed to view some information, we generate the list of users in the subtree with root *U* and check if the information belongs to any of the users in that list.

Data structure



Subrees

root	*Subtree*
1	1, 3, 4, 6, 7, 8
2	2, 5
3	3
4	4, 6, 7, 8
5	5
6	6
7	7
8	8

Figure 3 Example data structure and subtrees for the root entities

5.2 Partial solution 1: Recursive Structure

The solution 'recursive structure' describes a solution for the subtree problem. First we discuss the applicability of this solution. Then we describe the solution and discuss the consequences of applying the solution.

5.2.1 Summary of applicability:

- Recursive queries are database specific and not supported by all database systems.
- Portability decreases if this solution is applied.
- No redundant information is used and therefore the write performance and maintainability are good and storage space is minimal.

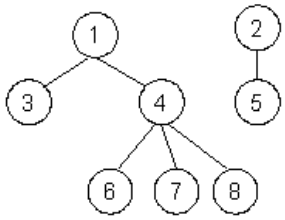
Entity	
PK	entityId
	parentEntityId

Figure 4 Data model solution recursive structure

5.2.2 Solution description

We add one column *parentEntityId*, which refers to the parent of an entity to the table *Entity*. The corresponding data model is displayed in figure 4. The value of this field is *NULL*, if no parent exists. An example of a structure with two hierarchies and their representation in the table *Entity* is displayed in figure 5.

Structure



Representation in Entity

entityId	*parentEntityId*
1	NULL
2	NULL
3	1
4	1
5	2
6	1
7	1
8	1

Figure 5 Example of the representation of a structure

In addition to the data model as described above, this solution requires a recursive query. Although some database systems support this type of queries, they are not part of the SQL:1999 standard. An Oracle version of such a query is (the Id of the entity indicating the root of the subtree is entered as `@rootEntityId`):

```
SELECT entityId
FROM Entity
CONNECT BY PRIOR entityId = parentEntityId
START WITH entityId = @rootEntityId
```

Here in order to support this query the Entity table should be enhanced with an index on (parentEntityId, entityId). The query then will be answered from the index alone. Note that the Entity table in general will have additional columns.

5.2.3 Consequences

Read performance One of the aspects that influence the read performance of this solution is the number of entities in the subtree defined by a root entity. We created a number of tests to investigate the influence of this number on the read performance. We found a linear relation between the number of entities and the execution time. The resulting graph is displayed in figure 6.
Write performance Since there is no redundant information and the amount of indexes is small, the write performance of this solution is good.
Functionality No functional restrictions exist for this solution.
Storage space No redundant information is used in order to store the subtree. Therefore the amount of storage space required is optimal.
Portability Recursive queries are not supported by default in SQL. Some databases offer special constructions [GEN2001] and others do not support recursive queries at all [POS2006]. Therefore the portability of the query in this solution is poor.

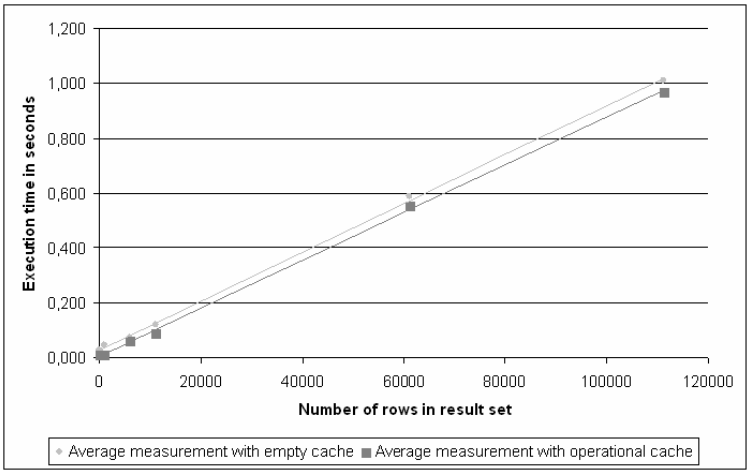


Figure 6 Influence of the number of entities in a subtree on the execution time

5.2.4 Relations

This is an isolated example. The only solutions available are the alternatives. They are related to the present solution by the problem description that they share.

5.3 Partial solution 2: Flat Structure in Rows

The solution 'flat structure in rows' describes an alternative solution for the subtree problem.

5.3.1 Summary of applicability

- The read performance of this solution is best for large subtrees. If subtrees are small, other solutions are more suitable.
- The goal of this solution is to improve read performance. As a result write speed decreases and more storage space is needed.

5.3.2 Solution description

This solution was designed to improve the read performance. The recursive solution as described in the previous section requires the database to access the table *Entity* multiple times. This solution was designed to limit this amount to one. Therefore we introduce redundant references in our data model.

We use two tables to store the hierarchical structure. The structure itself is stored in *EntityStruct* and the data is stored in *Entity*. For each entity we store references to all possible roots of subtrees in which the entity occurs in the field *EntityStruct.rootEntityId*. To be more precise, *rootEntityId* contains the following references for each entity *E*:

- Reference to *E* itself;
- Reference to any entity which is placed above *E* in the hierarchical structure.

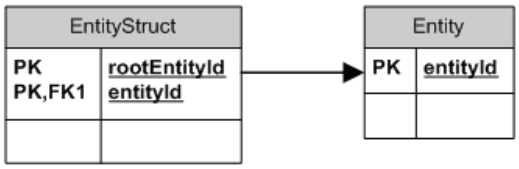


Figure 7 Data model solution flat structure in rows

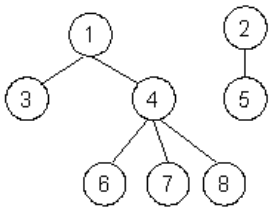
This means that an entity with two entities above occurs three times in the table *EntityStruct*. The corresponding data model is displayed in figure 7. An example of the representation of a structure in the table *EntityStruct* is displayed in figure 8. Note that we assume that each entity occur only once in the structure by defining the fields *rootEntityId* and *entityId* to be the primary key of *EntityStruct*. Only a simple query is needed to retrieve the required subtree:

```
SELECT entityId
FROM EntityStruct
WHERE rootEntityId = @rootEntityId
```

When the query needs to produce more Entity related information, an inner join with the Entity table needs to performed.

Just as in the case of the recursive structure solution, we want to make use as much as possible of an index to avoid the access of the table itself. Since we select on the rootEntityId column and require the entityId column as output and possibly as join column for retrieving information from the Entity table, and index on (rootEntityId, entityId) is indicated. Since this combination is also designated as the primary key, an index on these columns will already be present. One has to make sure of the proper order of the columns though.

Structure



Representation in EntityStruct

entityId	*rootEntityId*
1	1
2	2
3	1
3	3
4	1
4	4
5	2
5	5
6	4
6	1
6	6
7	4
7	1
7	7
8	4
8	1
8	8

Figure 8 Example of the representation of a structure

5.3.3 Consequences:

Read performance One of the aspects that influences the read performance of this solution, is the number of entities in the subtree defined by a root entity. We created a number of tests to investigate the influence of this number on the read performance. We found a linear relation between the number of entities and the execution time. The resulting graph is displayed in figure 9.

Write performance The write performance of this solution is influenced by the existence of redundant information. Each time the structure of a hierarchy is changed, the redundant information needs to be updated as well. The amount of updates required depends on the position in the structure.

Functionality It is not always possible to retrieve the exact level number. There is no difference for instance between a hierarchy consisting of one entity at level one and a hierarchy consisting of one entity at level 3 unless we know that there is only one root at level 1 of the entire structure. This can be necessary if the different levels have a different meaning (e.g. country, city, street). One additional column to store the level can solve this problem, but this requires some additional maintenance.

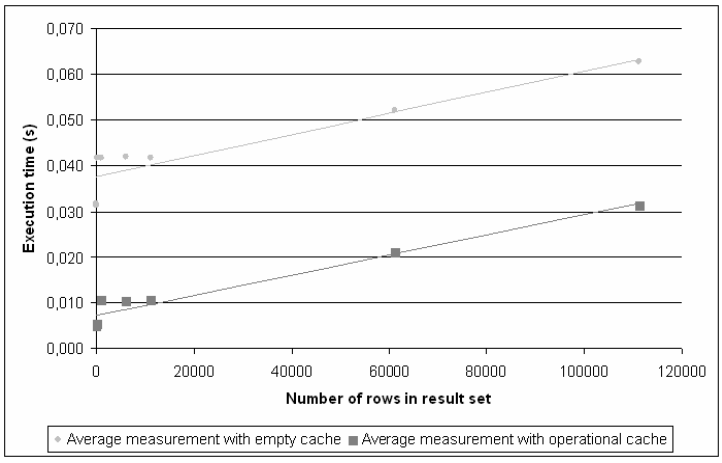


Figure 9 Influence of the number of entities in a subtree on the execution

Maintenance As mentioned earlier, the redundant rows need additional updates and therefore the complexity of maintenance increases.
Storage space The additional space for the redundant information depends on the levels the entities are located on. An entity at level *x* requires *x* rows and therefore, the number of rows in the table EntityStruct becomes *O*(*#entities* × *#levels*).
Portability No database specific properties are used for this solution, so portability is good.

5.3.4 Relations
This is an isolated example. The only solutions available are the alternatives. They are related to the present solution by the problem description that they share.

5.4 Partial solution 3: Flat Structure in Columns

The solution 'flat structure in columns' describes a third solution for the subtree problem.

5.4.1 Summary of applicability

- The maximum number of levels in the structure must be known in the implementation phase.
- Read performance is average for both small and large subtrees.
- The goal of this solution is to improve read performance. Thereby the write performance decreases and the storage space required increases.

5.4.2 Solution description

Similar to the previous solution, the goal of this solution is to enable the database to retrieve the subtree by scanning the table Entity only once and thereby improve the read performance. Therefore we introduce redundant information in our data model as well.

For each entity we store references to all roots of subtrees in which the entity occurs. This is done by introducing one column for each level in the structure. In the column of level *x* a reference to the root at level *x* must be stored. If no root exists at that level, the value is *NULL*. Now if we know at what level an entity *E* is located, we can find all elements of its subtree by scanning the column corresponding to the level of *E*. The corresponding data model is displayed in figure 10. The column *level* is used to store the level on which the entity is located in the structure.

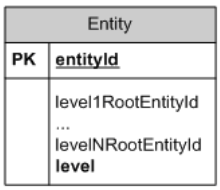


Figure 10 Data model solution flat structure in columns

Consider for example the structure as displayed in figure 11. This structure consists of three levels, so three columns are used to store the structure. The entity 4 is contained in two subtrees: the subtree with root 1 at level 1 and the subtree with root 4 at level 2. Since entity 4 is at level 2, no subtree exists at level 3. A more complicated example can be found in figure 12.

The query that goes with this datamodel assumes that the level of the root entity is known:

```
SELECT entityId
FROM EntityStruct
WHERE levelxRootEntityId = @rootEntityId
```

In this solution single column indices on all levelxRootEntityId columns are required.

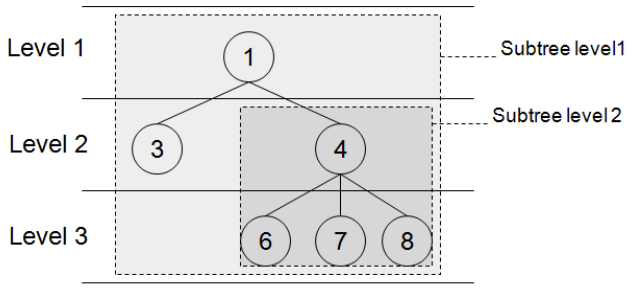
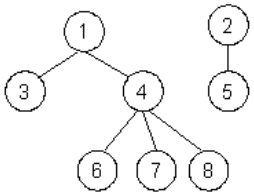


Figure 11 Subtrees in which the entity 4 occurs

Structure



Representation in Entity

entityId	*level1Root EntityId*	*level2Root EntityId*	*level3Root EntityId*
1	1	NULL	NULL
2	2	NULL	NULL
3	1	3	NULL
4	1	4	NULL
5	2	5	NULL
6	1	4	6
7	1	4	7
8	1	4	8

Figure 12 Example of the representation of a structure

5.4.3 Consequences

Read performance One of the aspects that influence the read performance of this solution, is the number of entities in the subtree defined by a root entity. We created a number of tests to investigate the influence of this number on the read performance. We found a linear relation between the number of entities and the execution time. The resulting graph is displayed in figure 13. As can be seen, the execution time is almost constant for this solution.

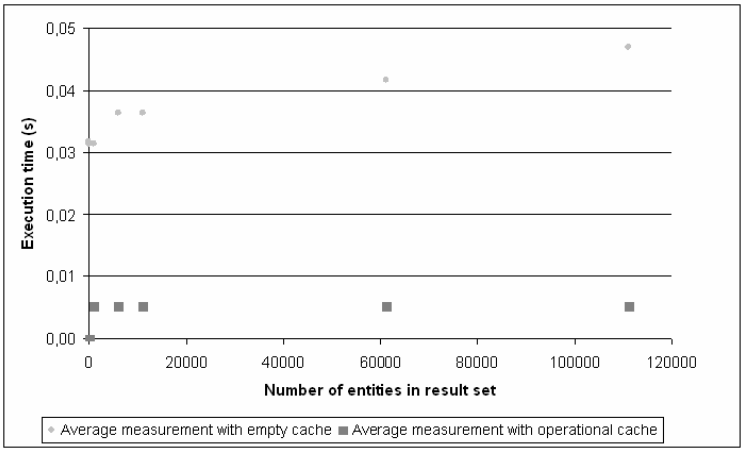


Figure 13 The influence of the number of entries in a subtree on the execution time

Write performance Each time the position of an entity in the structure is changed all redundant columns must be updated as well.
Functionality Since each level in the structure is represented by a column in the data model, the maximum number of levels must be known at design time. Later on, additional levels can only be created by adapting the data model.

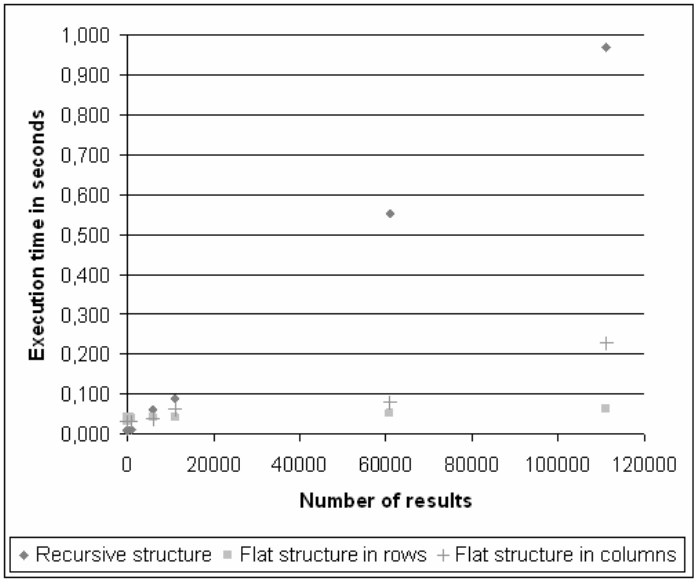


Figure 14 Influence of the number of entities in a subtree on the execution time for different solutions

Maintenance Each time the structure of a hierarchy is changed all redundant references must be updated as well. The data model does not guarantee that all references are functionally correct, so this has to be done by a query and as a result, maintenance becomes more complicated.
Storage space Since each entity contains *#levels* references whereas one would suffice, the amount of storage space reserved for redundant references is *O(#levels × #entities*).
Portability No database specific properties are used for this solution, so portability is good.

5.4.4 Relations

This is an isolated example. The only solutions available are the alternatives. They are related to the present solution by the problem description that they share.

5.5 Summary of the solutions

In this section, we summarize the results from all three solutions. The read performance of the solutions is displayed graphically in figure 14. A summary of all consequences is displayed in table 1.

Aspect	Parameter	Recursive	Flat in rows	Flat in columns
Read performance	Number of entities in subtree	Suitable for numbers < N	Not suitable for numbers < N, best for numbers > N' (> N)	Average for all numbers
	…	…	…	…

Write performance	Redundant information	None, optimal write performance	Multiple row updates for changes in structure	Multiple column updates for changes in structure
	Indices	…	…	…
Functionality	Maximum number of levels	None	May need an additional level number to be stored	Must be known in implementation phase
Maintenance	Redundant information	None	Correctness must be guaranteed in code	Correctness must be guaranteed in code
Storage space	Redundant information	None	Number of rows in *EntityStruct* $O(\#entities \times \#levels)$	Number of additional references in columns $O(\#entities \times \#levels)$
Portability	Database dependent queries	Recursive query	None	None

Table 1 Decision table for the choice of solutions for the subtree pattern

6. DISCUSSION AND OUTLOOK

In the previous sections we discussed the concept of a database performance pattern. In an actual application, many more aspects come into play. So far we have only considered patterns for databases on a problem by problem basis. Of course, the typical workload for an application will involve many different types of queries and updates the performance of which will have to be guaranteed. These different types will entail conflicting requirements. One will therefore have to choose which queries to favor over which others. In such a situation one typically will apply the centering principle [SW2002], which states that one should allocate most resources to those tasks for which they have the most effect. In extreme cases, workloads comprising thousands of different database actions need to be optimized. In such cases, another approach using automated tools, such as discussed in [CGL2003] will be needed.

In the discussion of performance patterns we so far have considered only the functional and technical design and implementation phases of a project. To allow discrimination between alternative solutions we have made use of tests, e.g. to find out the values of N and N' in table 1. Such tests can be used again in the testing phase to validate the implementation of the database against its requirements. A useful extension of the pattern template therefore appears to be the inclusion of the test descriptions and datasets.

In addition to the performance of the database subsystem, the performance of an application also will depend on the performance of other the components in the system. For instance, in a web application, performance patterns can cover elements such as client, application server, database and communication between client and application server or between application server and database. Since the performance of a solution is influenced by parameters from all elements covered by the solution, it is easier to describe solutions for one element only than for a combination of elements. Therefore we also envisage application, client, and communication performance patterns. Probably performance patterns that cover multiple elements at once exist as well. We have not seen the use of those in practice however. Furthermore it will be much harder to describe them in terms of building blocks since the problem space is much larger than for the single element patterns.

In this paper we have proposed a mechanism to record knowledge about (database) performance patterns and demonstrated how to use it. To realize the full benefit from performance patterns in the

software development life cycle, steps need to be integrated to structurally identify new performance patterns, and update available ones. This topic is currently under study. Of course, such steps will involve the creation of awareness and motivation of designers and developers to use performance patterns.

7. REFERENCES

[Ale1979] C. Alexander, 1979. The Timeless Way of Building. *Oxford University Press*, New York.

[AIS1977] C. Alexander, S. Ishikawa, and M. Silverstein, 1977. A Pattern Language: Towns, Buildings, Construction. *Oxford University Press*, New York.

[CB2007] B. Cao and A. Badia, 2007. SQL Query optimization through nested relational algebra. *ACM transactions on database systems, Vol. 32, No. 3, Article 18*, pp. 1-46.

[CGN2003] S. Chaudhuri, P. Ganesan, V. Narasayya, 2003. Primitives for Workload Summarization and Implications for SQL, *Proceedings of the 29th VLDB Conference*, Berlin, Germany.

[CH1998] S. Chaudhuri, 1998. An overview of query optimization in relational systems. *Proceedings of the ACM PODS conference*, Seattle, pp. 34-43.

[COR1993] P. Corrigan and M. Gurry, 1993. ORACLE performance tuning, *O'Reilly & Associates, Inc*.

[Fow1996] M. Fowler, 1996. Analysis Patterns : Reusable Object Models. *Addison-Wesley*, Boston.

[Fow2003] M. Fowler, 2003. Patterns of Enterprise Application Architecture. *Addison-Wesley*, Boston, 2003.

[GEN2001] J. Gennick, 2001. New CONNECT BY features in Oracle database 10g, *Oracle Magazine*.

[GH1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, 1995. Design Patterns: Elements of Reusable Object-Oriented Software. *Addison-Wesley*.

[JBP] http://java.sun.com/reference/blueprints/ (visited 15 december, 2007)

[KC1997] Wolfgang Keller and Jens Coldewey, 1997. Relational database access layers -- A pattern language, *in Robert C. Martin, Dirk Riehle, Frank Buschmann (Eds.): Pattern Language of Program Design 3, Addison-Wesley*.

[KEL1997] Wolfgang Keller, 1997. Mapping objects to tables - A pattern language, *Proc. of European conference on pattern languages of programming conference (EuroPLOP)97, Bushman, F. and Riehle, D. (eds)*, Irsee, Germany.

[KEL1998] Wolfgang Keller, 1998. Object / relational access layers: A roadmap, missing links and more patterns", *In 3rd European conference on pattern languages of programming and computing (EuroPLoP'98)*, Bad Irsee, Germany.

[MIL2002] Hafedh Mili, Ali Mili, Sherif Yacoub, and Edward Addy, 2002. Reuse based software engineering: Techniques, orgtanization, and measurement, *John Wiley & Sons*.

[POS2006] The PostgreSQL Global Development Group, 2006. PostgreSQL 8.2.0 Documentation.

[RH2004] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst, 2004.Workflow Data Patterns. *QUT Technical report, FIT-TR-2004-01, Queensland University of Technology*, Brisbane.

[SE1979] G. P. Selinger et al., 1979. Access path selection in a relational database management system. *Proceedings of the ACM Sigmod international conference on management of data*, pp. 23-34.

[SHA2003] D. Shasha and P. Bonnet, 2003. Database tuning: principles, experiments, and troubleshooting techniques, *Morgan Kaufmann Publishers Inc*.

[SKS2005] A. Silberschatz, H. F. Korth, and S. Sudarshan, 2005. Database system concepts, 5th Edition, *McGraw-Hill Book Company*.

[SM1990] Connie U. Smith, 1990. Performance engineering of software systems, *Addison-Wesley*.
[SW2002] Connie U. Smith and Lloyd G. Williams, 2002. Performance solutions: a practical guide to creating responsive, scalable software, *Addison Wesley Longman Publishing Co., Inc.*
[SW2002b] Connie U. Smith and Lloyd G. Williams, 2002. New software performance antipatterns: More ways to shoot yourself in the foot., *in: Proceedings of the 28th International Computer Measurement Group Conference*, pp. 667-674.
[TPC] http://www.tpc.org/default.asp (visited 15 December 2007)
[WfP] http://www.workflowpatterns.com/ W. van der Aalst en A. ter Hofstede (visited 15 december, 2007)