

# University of Wollongong - Research Online

## Thesis Collection

Title: Adaptive reorganization of database structures through dynamic vertical partitioning of relational tables

Author: Liu Zhenjie

Year: 2007

Repository DOI:

### Copyright Warning

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following: This work is copyright. Apart from any use permitted under the Copyright Act 1968, no part of this work may be reproduced by any process, nor may any other exclusive right be exercised, without the permission of the author. Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material.

Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

**Unless otherwise indicated, the views expressed in this thesis are those of the author and do not necessarily represent the views of the University of Wollongong.**

Research Online is the open access repository for the University of Wollongong. For further information contact the UOW Library: [research-pubs@uow.edu.au](mailto:research-pubs@uow.edu.au)

*University of Wollongong Theses Collection*

*University of Wollongong Theses Collection*

---

*University of Wollongong*

*Year 2007*

---

Adaptive reorganization of database  
structures through dynamic vertical  
partitioning of relational tables

Liu Zhenjie  
University of Wollongong

Liu, Zhenjie, Adaptive reorganization of database structures through dynamic vertical partitioning of relational tables, MCompSc thesis, School of Information Technology and Computer Science, University of Wollongong, 2007. <http://ro.uow.edu.au/theses/33/>

This paper is posted at Research Online.  
<http://ro.uow.edu.au/theses/33>

## **NOTE**

This online version of the thesis may have different page formatting and pagination from the paper copy held in the University of Wollongong Library.

## **UNIVERSITY OF WOLLONGONG**

### **COPYRIGHT WARNING**

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.



# Adaptive Reorganization of Database Structures Through Dynamic Vertical Partitioning of Relational Tables

A thesis submitted in fulfillment of the  
requirements for the award of the degree

**Master of Computer Science by Research**

from

UNIVERSITY OF WOLLONGONG

by

**Zhenjie Liu**

School of Information Technology and Computer Science  
October 2007

© Copyright 2007

by

Zhenjie Liu

All Rights Reserved

*Dedicated to  
My parents and wife Yanting*

# Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

---

Zhenjie Liu  
October 23, 2007

# Publications

---

Zhenjie Liu and Janusz R. Getta. Optimization of query processing through constrained vertical partitioning of relational tables. In *DBA'06: Proceedings of the 24th IASTED international conference on Database and applications*, pages 221-227, Anaheim, CA, USA, 2006. ACTA Press.



# Abstract

---

Performance tuning of relational database systems is always a challenging task for database administrator. Automated performance tuning has been proposed recently as a new approach to detect and to eliminate performance problems and to support the decisions of database administrators.

This work considers one of the techniques used in automated performance tuning, dynamic vertical partitioning. Dynamic vertical partitioning of relational tables is one of the ways in which the physical structures of a relational database can be reorganised automatically in order to improve the performance of future database applications. The thesis presents how dynamic vertical partitioning can be used for the comprehensive analysis and optimisation of an adaptive reorganisation of database structures. In particular, we propose the algorithms to use to predict the future workload of the system, to analyse the characteristics of the workload, and to find a near optimal vertical partitioning of relational tables. Then, we discuss the implementation aspects of vertical partitioning with the materialized view and index-based techniques.

Our contributions to automated performance tuning of relational database systems can be summarised as follows:

1. Propose a cost model to perform a detailed analysis of the costs of query and data manipulation processing over a given configuration of a relational database;
2. Propose a new algorithm for vertical partitioning of relational schemas in a database system with a given level of redundancies for a given workload;
3. Discuss the limitations of static vertical partitioning and propose dynamical vertical partitioning;
4. Discuss the characteristics of workload in order to predict the future workloads of the system;

5. Implementation aspects of vertical partition discussion: materialized view based and index based;
6. Discussion of the implementation of a vertical partition as a virtual view;
7. Conduct experiments to confirm the correctness of the cost model used by the vertical partitioning algorithm and demonstrate the expected performance gains from the partitioning

# Acknowledgements

---

The research work for this thesis was undertaken at the University of Wollongong.

Firstly, I would like to thank my supervisor, Dr. Janusz R. Getta for his guidance and help in my research. Without him, this thesis would not have been possible.

Also, I wish to acknowledge the support I have received from all the staff in the School of IT & CS, University of Wollongong.

Finally, I would like to thank my family and friends for their enduring love and support.

# Contents

---

<b>Publications</b>	<b>v</b>
<b>Abstract</b>	<b>vi</b>
<b>Acknowledgements</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Database Performance Tuning . . . . .	1
1.1.1 Overview of Automatic Database Administration . . . . .	1
1.1.2 Automatic Query Performance Tuning . . . . .	2
1.2 Partitioning . . . . .	3
1.2.1 Horizontal Partitioning . . . . .	3
1.2.2 Vertical Partitioning . . . . .	5
1.2.3 Comparison of Horizontal and Vertical partitioning . . . . .	5
1.3 The Challenge . . . . .	6
1.4 Strategy of Solution . . . . .	7
1.5 Thesis Organisation . . . . .	7
<b>2 Technical Backgrounds</b>	<b>9</b>
2.1 Autonomic Database Management Systems . . . . .	9
2.2 Automatic Performance Optimisation . . . . .	11
2.2.1 Performance Measurement Methodologies . . . . .	11
2.2.2 Automatic Performance Monitoring and Diagnosis . . . . .	11
2.2.3 Automatic Performance Tuning . . . . .	12
2.3 Dynamic Query Optimisation . . . . .	13
2.3.1 Query Execution Optimisation . . . . .	13
2.3.2 Physical database structure Reorganization . . . . .	13
2.4 Static Vertical Partitioning . . . . .	14

2.4.1	Non-Overlapping Partitioning . . . . .	14
2.4.2	Overlapping Partitioning . . . . .	14
2.4.3	Finding interesting partitions . . . . .	15
2.5	Dynamical Vertical Partitioning . . . . .	16
2.5.1	Workload Monitoring and Analyzing . . . . .	16
2.5.2	Implementing Vertical Partition . . . . .	17
2.6	Summary . . . . .	17
<b>3</b>	<b>Static partitioning</b>	<b>18</b>
3.1	Preliminaries . . . . .	18
3.2	Cost Model . . . . .	21
3.3	Partition Configuration . . . . .	23
3.3.1	Algorithm overview . . . . .	23
3.3.2	Choosing the partitions . . . . .	24
3.3.3	Recycling of the original tables . . . . .	25
3.3.4	Merging the partitions . . . . .	26
3.4	Experiments . . . . .	27
3.4.1	Evaluation of partitioning . . . . .	27
3.4.2	Trend of Storage and Benefit . . . . .	28
3.5	Summary and open problem . . . . .	29
<b>4</b>	<b>Dynamic Partitioning</b>	<b>31</b>
4.1	Workload Patterns . . . . .	31
4.2	Preliminaries . . . . .	32
4.3	Cost Model . . . . .	35
4.4	Overview of Dynamic Partitioning Algorithm . . . . .	35
4.5	Potential Low Workload Time Detecting . . . . .	37
4.6	Finding Most Similar Workload . . . . .	38
4.6.1	Function to calculate similarity of two signatures . . . . .	38
4.6.2	Most Similar Workload Searching . . . . .	39
4.7	Finding configuration . . . . .	39
4.7.1	Choosing partitions . . . . .	39
4.8	Merging the partitions . . . . .	41
4.9	Partition Implementation . . . . .	41
4.10	Comparison between Static and Dynamic partitioning . . . . .	42
4.11	Summary . . . . .	42

<b>5</b>	<b>Implementation of Dynamic Vertical Partitioning</b>	<b>44</b>
5.1	Issues of implementing partitions . . . . .	44
5.1.1	Materialized View-based Implementation . . . . .	45
5.1.2	Index based Implementation . . . . .	46
5.2	Comparison of Index-based and Materialized view-based . . . . .	47
5.2.1	Consistency Control Comparison . . . . .	47
5.2.2	Storage Cost Comparison . . . . .	48
5.2.3	Comparison by Time Cost . . . . .	49
5.3	Open problems . . . . .	50
5.4	Experiments . . . . .	51
5.4.1	Query Rewrite of Materialized View . . . . .	51
5.4.2	Fast Full Index Scan . . . . .	51
5.4.3	Comparison of Materialized View and Index . . . . .	52
5.4.4	View Update . . . . .	52
5.5	Summary . . . . .	53
<b>6</b>	<b>Conclusions and Future work</b>	<b>56</b>
6.1	Conclusions . . . . .	56
6.2	Future work . . . . .	57
	<b>Bibliography</b>	<b>59</b>

# List of Tables

---

# List of Figures

---

1.1	Principle of horizontal partitioning . . . . .	4
1.2	Principle of vertical partitioning . . . . .	6
3.1	Outline of algorithms . . . . .	24
3.2	Storage and performance . . . . .	28
3.3	Performance comparison . . . . .	28
4.1	Total workload . . . . .	33
4.2	Example of session and signature . . . . .	34
4.3	Workload before dynamic partitioning . . . . .	36
4.4	Workload after dynamic partitioning . . . . .	36
4.5	Overview of dynamic partitioning . . . . .	37
5.1	Example of Materialized View . . . . .	46
5.2	Horizontal traversal of index . . . . .	47
5.3	Query Rewrite of Materialized View . . . . .	52
5.4	Fast Full Index Scan . . . . .	52
5.5	Comparison of MV and Index . . . . .	53
5.6	View Update . . . . .	54



# Chapter 1

---

## Introduction

### 1.1 Database Performance Tuning

Performance tuning of relational database systems has an important impact on the successful implementation of modern information systems such as e-business systems, decision support systems, operational business process implementation systems, etc. Poor performance of e-business applications and system crashes have a disastrous effect on customers' attitudes about the system used. High performance of the database servers supporting e-business applications requires system configuration parameters well tuned to the physical properties of the available hardware, efficient query and data manipulation processing, and efficient processing of customer transactions. Performance tuning is always a challenging task for database administrators. Some researchers agreed that it is extremely hard for DBAs and other IT operators to tune a complex system under pressure [17, 7]. Database administrator are likely to make mistakes when tuning complicated system. The automatic performance tuning of relational database servers is considered as an important solution when reducing the running costs of information services [45].

#### 1.1.1 Overview of Automatic Database Administration

Automatic performance tuning is one of the automatic database administration features. Automatic database administration includes two main fields: automatic performance administration and automatic security administration. Automatic performance administration includes self-optimization, self-configuration, self-organizing and self-recovery. Self-optimization enables a database system to automatically optimise a query execution such as query translation, or the modification of a query plan. Self-configuration means that a database system is able to dynamically adjust for hardware and software setting parameters. Self-healing allows the database system to recover

from failure by using logs and backups automatically. Self-organizing is a way to dynamically reorganise a physical database structures. Automatic security administration includes self protecting and auto inspecting. Self-protecting concerns the privacy, auditing mechanisms, encryption of data, and access control. Self-inspecting occurs when the system can detect the problems by itself and provide the auditing for itself.

In the thesis, we propose a technique, dynamic vertical partition, which contributes to the self-optimization and self-organization features of the autonomic database systems. Automatic query optimization is the most important task of self-optimization. To achieve automatic query optimization, one of the ways is to adjust the physical database designs in order to reduce the unnecessary read in a query. Vertical partitioning of the relational tables is one of the ways to adjust the physical database designs.

### 1.1.2 Automatic Query Performance Tuning

The rapid development of the internet application and more and more complex and complicated application system brings a major challenge relates to query optimization. First of all, the continuous growth of database size could slow down the query execution dramatically. Also, the higher-level application services requires the database technology to be unbundled and dispersed. The applications provides user friendly query interfaces which allows users input the selection criteria. Such applications are more focus on the business requirements and rules rather than optimization of the queries. This creates a good amount of nature and non-tuned queries which may make the database system run inefficiently. Therefore, it is critical that database systems can tune the query processing automatically.

To achieve automatic query performance tuning, we can improve data manipulation and query execution plans on run time, and reorganise the physical database structures with the change of workload.

1. Dynamic modification of query processing plans:

Currently, commercial database management systems can provide a cost-efficient execution plan for most queries. There are many alternative ways to execute a plan for a query. Query optimizer evaluates the cost of each alternative and chooses the most cost efficient way. However, estimating a cost query execution is difficult and sometimes the query optimizer will provide a sub-optimal execution plan. The accuracy of estimation strongly relies on up-to-date historical statistics and column distributions. Besides, the explain plan environment can be different

from the execution environment. Therefore, the query optimizer is expected to dynamically change the execution plan.

To dynamically re-optimize the query execution plan, a statistics collector model has been proposed to observe the sizes and data distributions of intermediate query result sizes at run-time [29]. If a sub-optimal execution plan is detected, the query optimizer should dynamically change the remainder parts of execution plan.

## 2. Dynamic storage reorganization techniques

The adaptive reorganisation of the physical database design includes the automatic implementation of additional persistent data structures such as various types of indexes, clusters, materialized views, partitions, etc. in a reply to the expected query and data manipulation operations.

On one hand, the significant improvements in the performance of relational database systems can be achieved through well tailored adjustments to the physical database designs matching the requirements of user applications. On the other hand, a large number of different types of user applications still makes it impossible to have a one-fits-all design that satisfies the wide range of contradictory requirements to fit all user applications. This is why the adaptive reorganisation of physical database structures to the anticipated data access requirements of user applications is an attractive option for the implementation of self-tuning relational database servers.

## 1.2 Partitioning

The thesis targets the performance problems in the systems where a significant amount of query processing time is spent on the full scans of the large relational tables. Partitioning of the relational tables either through vertical partitioning or horizontal partitioning or a combination of both is a reliable way for these problems to be solved. In this section, we will introduce the basic principles of horizontal partitioning and vertical partitioning.

### 1.2.1 Horizontal Partitioning

Horizontal partitioning breaks down a relational table into smaller pieces, called *partitions*, according to the range of values of a given attribute or combination of attributes,

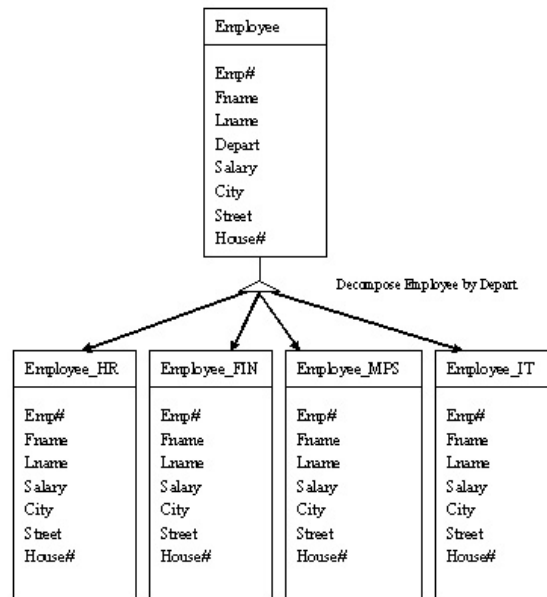


Figure 1.1: Principle of horizontal partitioning

called a *partition key*. Horizontal partitioning reduces the total number of read operations needed to access the rows determined by the values of the partition key provided in a query. Figure 1.1 shows how horizontal partitioning works. Suppose there is a table recording employee information and it contains columns (emp#, f\_name, l\_name, department, salary, city, street, house#). The department attribute records the division of the employee. It has four values: Human Resources(HR), Finance(FIN), Marketing and Product Strategy(MPS) and Information Technology(IT). If the queries access the information by department, we can partition the table employee into four divisions: employee\_HR, employee\_FIN, employee\_MPS, employee\_IT. Therefore, for queries only retrieving information on employees within human resources, we will only access the partition employee\_HR and improve the query processing.

Further improvements can be achieved through vertical partitioning and through replication of data in the appropriate partitions. In this paper, we focus on vertical partitioning with a controlled level of redundancies in the database.

### 1.2.2 Vertical Partitioning

Paradoxically, a database system quite frequently has to access the large amounts of data in order to retrieve or to update a relatively small number of values determined by a user. This is mainly due to the average row length being significantly bigger than the amount of data retrieved or modified in a row. A typical example is a projection on a single attribute of a relational table whose schema consists of many attributes with the values consuming a larger amount of storage. Vertical partitioning splits a relational table into a number of pieces, also called partitions, and replicates a primary key in each partition. This reduces the average row length and in consequence, minimizes the total number of read and write operations.

Figure 1.2 is an example to show the principle of vertical partitioning. Take the employee table discussed in Figure 1.1 as an example. The Human Resources department will require the employees' address information frequently. When scanning the employee table to retrieve address information, the employee's other non-relevant information such as department, salary will also appear. In such a situation, we can partition the table employee into two parts: Emp\_Dept, Emp\_Addr. Then the queries needing only access to address information can be improved by avoiding access to other employee's information.

### 1.2.3 Comparison of Horizontal and Vertical partitioning

Both horizontal partitioning and vertical partitioning can avoid the unnecessary reading of data blocks. However, vertical partitioning is easier to implement than horizontal partitioning. For example, suppose there is table  $T(A, B, C, D, E, F, G)$  and the value of  $G$  ranges from 0 to 50. And there is a query (Select  $A, B, C, D$  from  $T$  where  $G \geq 10$  and  $G < 20$ ). If we create a vertical partition  $VP_1(A, B, C, D, G)$  for  $T$  and horizontal partition  $HP_1$  which  $G$  ranges from 10 to less than 20,  $HP_2$  which  $F$  ranges from 20 to 80.  $VP_1$  can avoid reading unnecessary attribute  $E, F$  while  $HP_1$  can avoid reading rows other than  $(10 \leq G < 20)$ . But because both the predicates of the query and the predicates of horizontal partition can be much more complex, matching both predicates of query and horizontal partition may not be achievable. Consider the condition on attribute  $E, F, G$  is  $[(10 \leq G < 20) \text{ or } \text{Not}(30 \leq G < 50)) \text{ And } (20 \leq F < 80)] \text{ Or } (E = F + G)$ . Since the selection criteria of  $E$  is variable, the query optimizer cannot locate the horizontal partitions for such a condition. By comparison, finding the optimal vertical partition is much simpler than horizontal partitioning because we

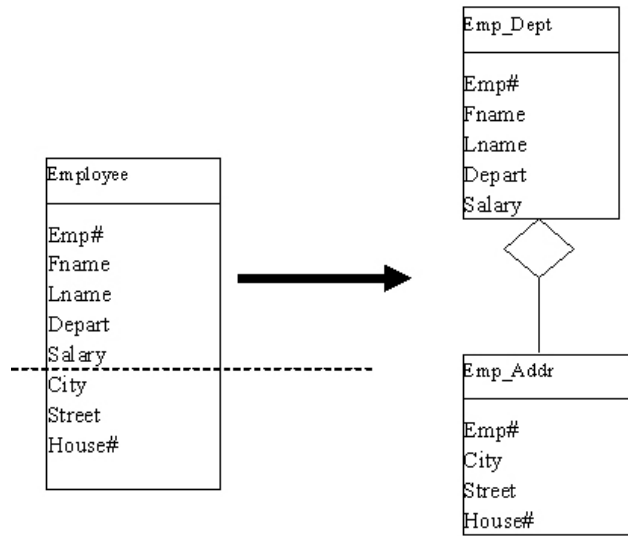


Figure 1.2: Principle of vertical partitioning

only match the attributes used in the query and the attributes in the vertical partition.

## 1.3 The Challenge

The aim of this work is to find how the physical database structures can be dynamically reorganised in order to improve the performance of a relational database server. It is desirable that the solution should have the following characteristics.

1. One of important feature is that the optimisation of the system should be automatically adaptive with the change of workload. The workload of a database system changes with time. An optimal configuration for current workload may cause low performance for the workload in the future. The configuration of vertical partitioning for optimising system must be updated with the workload change.
2. Another important feature is the optimization configuration of vertical partitioning should be transparent to the end user and database applications, to avoid a rewriting query for the change of configuration.

## 1.4 Strategy of Solution

To achieve the objectives described above, we propose the following strategy:

1. Propose a new algorithm for static vertical partitioning of relational schemas in a database system under storage constraints.
2. Introduce an algorithm to predict the future workload of the system. Note that, we assume that a workload cannot rapidly change. Rapid changes occur only when certain events happen, otherwise the system is stable. Workload changes happen in a periodic way which makes prediction reliable.
3. Create vertical partitions for a peak time workload during a low workload time.
4. Propose a method to analyse the characteristics of a workload.
5. Index based and materialised view-based implementation of vertical partitioning are proposed and analysed in order assure the transparency of dynamic vertical partitioning.
6. Perform the experiments to confirm the benefit of vertical partitioning and compare the implementation of vertical partitioning for an index and a materialized view.

## 1.5 Thesis Organisation

The rest of the thesis is organised as follows:

1. In chapter 2, we will present a review of some techniques used and the technical background for automatic tuning database systems in the past.
2. Chapter 3 will be devoted to the current static partition for database tuning. This will introduce a new algorithm that vertically partitions the relational tables according to a given workload and accordingly to a given limit of redundancies acceptable in a database.
3. In Chapter 4, we discuss the algorithm used to change the configuration of a vertical partition for a dynamic workload. Since workloads change in a set pattern, we can trace a certain moment in the past with a present similar workload and then predict what will happen next. The implementation of vertical partitions

will be set up in the low load time so that in high load time, the workload can be relieved by the configuration.

4. To achieve an automatic tuning database system, the change of configuration should be completely transparent to the end user and the applications. This requires that the implementation of the vertical partitioning should be recognised by the query processor and that the query processor will automatically change the routine for the queries. Index and Materialized View are two options for implementing vertical partition. We will compare these two options in Chapter 5, and discuss the storage cost and time cost of building an index.
5. Chapter 6 is the conclusion, where we will summarise our work within this thesis, together with a list suggesting future research directions.
6. The last part of this thesis comprises the references.



# Chapter 2

---

## Technical Backgrounds

With the increasing complexity of database management systems, many researchers proposed the autonomic database system to reduce database administrator's work. An autonomic database system is able to manage itself and to dynamically adapt to the changes of workload. This chapter begins with a brief introduction of autonomic database management system features. Then, an overview about automatic database performance tuning is presented. The database tuning approaches are classified according to their characteristics and several approaches will also be described in this chapter. Vertical partitioning of relational tables is one of the ways to improve database applications by avoiding unnecessary read. The related work on vertical partitioning will be discussed as well.

### 2.1 Autonomic Database Management Systems

The increasing complexity of current database applications demands more administrative costs, such as system installation, configuration, upgrade, deployment and tuning, administrator's training and salaries and so on. Therefore, some researchers have proposed an autonomic database management system, which helps to set administrators free from routine database management and tuning tasks.

The self-managed commercial database management systems should possess the features: self-optimized, self-configuring, self-healing, self-protecting, self-organised and self-inspected [16].

1. self-optimized: it allows DBMS to perform any task in the most efficient way under the constraint of a present workload, hardware configurations and available resources. Apparently, query optimization is the most important task of self-optimization. It involves query translation, the generation of an optimal execution plan and dynamic runtime optimizations [18]. Self-optimized will be

discussed in detail in the following sections.

2. self-configuring: configuration of hardware and software should be adaptive with the changes of environment. A DBMS configuration includes: consumption thresholds of resources like CPU, memory, disks and the performance parameters. To enhance system performance, on one hand, we may add memory, disk or CPU. After adding these resources, we may adjust their allocation and tune the degree of parallelism. On the other, the tuning parameters such as size of log file, buffer size, block size and checkpoint interval and etc. could be modified automatically without severely disrupting online operations. Current commercial database management systems can automatically change memory configurations. Self-Tuning Memory Manager (STMM) is proposed to dynamically tune database memory heaps such as sort memory, locking memory, compiled SQL cache, buffer pool and so on [43]. Memory Controller evaluates the cost-benefit of distribution of memory and then inputs the benefit into the integral control model to determine the redistribution frequency. By comparison, Automatic Shared Memory Management (ASMM by Oracle) and memory tuner in SQL Server have limited ability to automatically configure memory distribution.
3. self-healing: DBMS should provide a way to preserve consistency of database systems and to perform backup and recovery with minimal system disruption. Self-healing should maximise the reliability and availability of data while minimising outage of the system.
4. self-protecting: a self-protection should involve database security, privacy, data encryption, and admission control strategies. In [9], the authors discuss the database security issues such as secrecy, integrity and confidentiality of data. In [3], the authors identified the technical challenges in preserving privacy and performance and/or undesirably consuming system resources. The DB2 Query Patroller and the Oracle Resource Manager are examples of admission control tools used today [16].
5. self-organised: ADBMS should reorganise the table and its associated indexes to control the storage fragmentation level. Also, ADBMS should be able to determine the optimal set of indexes, materialized views, partitions for data access. ADBMS should recommend the optimal set of indexes, materialized views and partitions to be used by the query. Some related research work is discussed in the following section.

6. self-inspected: the DBMS should collect the signs of unhealthiness and store performance data automatically. Such information can be used to make recommendations for maintenance utilities.

The thesis contributes to self-optimization and self-organization features of the autonomic database systems.

## 2.2 Automatic Performance Optimisation

Automatic performance tuning includes performance measurement methodologies, automatic performance monitoring and diagnosis, automatic performance tuning.

### 2.2.1 Performance Measurement Methodologies

Different components of the database are measured in different metrics. For example, hit-ratio is used to measure the efficiency of the buffer cache; transaction-per-second is used to measure the database throughput; read and write latencies are used to measure the I/O. However, to determine the performance impact of a particular component over the total database throughput is extremely hard. A common currency called *Database Time* has been proposed to measure a workload [15]. Database time can be a measure to gauge the time spent in various phases of processing user requests and using or waiting for various database resources to process the requests. This work can help to gauge the cost saving of our work in total database throughput.

### 2.2.2 Automatic Performance Monitoring and Diagnosis

It is hard to discover the root cause of a low database performance and to fix the performance symptoms [45, 13, 25, 8].

The performance diagnosis techniques include *rule-based diagnosis* and *model-based diagnosis*. A class of rule-based diagnosis systems [37, 39] has been proposed to collect data from experts, build up a decision tree, and prune the decision tree to provide a collection of rules for the actions at performance troubleshooting stages. A sample model-based diagnosis system dynamically compares the behaviour of the system with a presumed and correct model. The differences between the assumed mode and the actual system are used to detect malfunctioning of the system. Model-diagnosis approach has a wide application in troubleshooting of mechanical devices, circuits, physical and biological systems. To automatically diagnose the performance problems, a system was

designed [26] to store performance related data in a multidimensional database and uses the contents of the database to determine the source of performance problems.

Automatic database diagnostic monitor(ADDM) [15] uses *DBTime-graph* to detect the performance bottlenecks and to provide the recommendations to alleviate them. The *DBTime-graph* represents two dimensional identification process of the causes of performance problems. Those two independent dimensions are: the database time spent in various phases of processing user requests and the database time spent using or waiting for various database resources used in processing user requests. ADDM explores the *DBTime-graph* from the root-node and exploring the node and its children which cause significant performance issue. In the end, ADDM reports the causes of performance issues by ranking the respective impacts. Recently, a prototype called *Resource Advisor* was proposed [33] to continuously collect trace information and maintain a usage summary of buffer pool, storage, CPU. The collected statistics could answer "what-if" questions and give hints to a resource upgrade.

### 2.2.3 Automatic Performance Tuning

Implementation of automatic performance tuning requires not only the investigations of automatic performance diagnosis, but also the approaches to improve performance dynamically.

Optimisation of Database system can be achieved by building a model to select the indices according to workload change; it also can be achieved by configuring proper concurrency control such as number of locks held by each transaction, types of locks, duration a transaction holds lock; besides, it can be achieved by allowing dynamic memory allocation according to workload change so that the buffer ratio can be kept in a recommended level; it can be achieved by executing the optimal execution plan provided by query optimizer; last but not least, it can be achieved by denormalisation of schema to reduce the joins among tables.

In the next section, we will discuss optimization of data manipulation and query execution plans, and reorganization of the physical database structures.

## 2.3 Dynamic Query Optimisation

### 2.3.1 Query Execution Optimisation

Producing an accurate execution plan relies on histograms. There are a few papers that focus on producing accurate histograms. Chaudhuri and Narasayya defined a technique called *Magic Number Sensitivity Analysis* to avoid creating not syntactically relevant statistics [12]. Some researchers explored how to efficiently construct and maintain histograms using sampling [11, 19, 38].

Since a query optimizer may not always find the best way to process a complex query involving complicated predicates, a prototype of a LEarning Optimizer (LEO) has been proposed to dynamically modify the query processing plans [31]. LEO compares the optimizer's estimates with actual cardinalities at each step in a query execution plan. If the estimates are different from actual cardinalities, optimizer will update its estimates for future optimisation of a similar query. At the same time, the detection of estimation errors can also trigger re-optimisation of a query in mid-execution.

### 2.3.2 Physical database structure Reorganization

The automated performance tuning of relational database servers is considered as an important solution when reducing the running costs of information services [45]. The significant improvements in the performance of relational database systems can be achieved through the well tailored adjustments to the physical database designs matching the requirements of user applications. On the other hand, a large number of different types of user applications still makes it impossible to have a one-fits-all design that satisfies the different requirements of all applications. This is why the adaptive reorganisation of physical database structures to the anticipated data access requirements of user applications is an attractive option for the implementation of self-tuning relational database servers. The adaptive structures reorganisation includes reorganisation of indexes, partitions, materialized view.

1. Index organizing: Currently, all commercial DBMSs provide an index advisor to generate a set of indexes for a given workload [11, 2]. In general, an index can retrieve data which satisfies a given selection conditions efficiently. However, if the retrieval of data is a large number rows of relational tables, full scanning tables may be more efficient than using an index. Moreover, if there are frequent updates on indexed data, the cost of index maintenance could be very high.

2. Materialized View organizing: Recently, some researchers proposed algorithms to pick up the materialized views to improve database performance [41, 22, 23]. Agrawal, Chaudhuri and Narasayya suggested to integrate selection of index selection and materialized view selection not in isolation. The drawback of using materialized view is the maintenance of consistency with an underlying table [4].
3. Partition organizing: this will be discussed in next section.

## 2.4 Static Vertical Partitioning

Finding the optimal partitioning for a given workload and a given database schema is NP-complete problem. The previous work on vertical partitioning can be grouped into two categories: overlapping and non-overlapping. First, in this section, we compare these two categories. Then we introduce two main steps to find an optimal configuration of vertical partitions. One is searching for the interesting column-group for partitioning. The other is evaluating the cost of the interesting column-group.

### 2.4.1 Non-Overlapping Partitioning

The configuration of partitions is overlapping if the intersection(except the primary key of the table) of any two vertical partitions is empty. An algorithm [34] has been addressed to partition a table into non-overlapping fragments based on attribute affinity. Some researchers proposed an optimal binary non-overlapping partitioning algorithm to optimise the number of disks accesses [14]. Recently, Agrawal, Narasayya and Yang presented a novel techniques to integrate horizontal partitioning and non-overlapping vertical partitioning into an automated physical database design [6].

- Advantage: duplicating attributes(except primary key) is not necessary. Therefore, not much storage for partitions is required. Furthermore, the cost for consistency control is less than for an overlapping partitioning.
- Disadvantage: in reality, it is extremely hard to find a "clean" solution for queries access common attributes.

### 2.4.2 Overlapping Partitioning

The configuration of partitions is overlapping if the intersection(except the primary key of the table) of any two vertical partitions is not empty. Navathe, Ceri and etc. not only

proposed a non-overlapping partitioning algorithm but also proposed an overlapping partitioning considering the cost spent on preserving the consistency of the replicated data [34]. AutoPart algorithm [1] was proposed to improve the performance of SDSS database [20, 44] by overlapped partitioning.

- Advantage: By duplicating attributes in different partitions we can satisfy more queries.
- Disadvantage: It requires additional cost to preserve the consistency of replicated data.

Since the cost to pay for disk is quite cheap, an acceptable redundancy may enhance the database performance dramatically.

### 2.4.3 Finding interesting partitions

Finding interesting partitions can reduce the complexity of finding the configuration of partitions. A bottom-up approach to partitioning is proposed in [42, 36]. Partitioning starts with single-attribute partitions, which are gradually increased by merging the smaller partitions. Currently, there are two main ways to find interesting partition: affinity based and transaction based.

1. Affinity based: One of the first solutions [32, 27, 24] proposed clustering algorithms based on the affinity between the attributes. Affinity is used for the identification of the usage patterns and groupings of the attributes, which later on, is applied to determine a representative workload. In [34], authors proposed an algorithm that begins with setting up an attribute pairwise affinity matrix. Then it employed an empirical cost function called BEA to separate the matrix into two sets of attributes: one is frequently used together and the other is less relevant. The complexity of this solution has been reduced in [35] by an application of the graphical techniques to optimise the partitioning. The above research did not provide a detailed cost model to evaluate the cost of the attribute. More recently, some researchers presented a method based on the interpretation of a workload as a powerset of items and on the analysis of the frequent itemsets to find out which column-set is interesting for partitioning [6, 21]. However, it is difficult to determine a precise support as it is an empirical value and it needs repeated tries.

2. Transaction based: A solution proposed in [14] assumes that the database transactions provide more semantic meaning than the attributes and because of that it applies a transaction binary partitioning algorithm to decompose the relational schemas. In [30], authors proposed a hypothetical solution that always finds the optimal partitioning would start from the generation of all combination of attributes from the workload.

The previous works focus on the search for an optimal solution for a given workload. However, the workload changes from time to time, the optimal solutions from static algorithms may optimise the database system at present but aggregate the burden in future. Therefore, in our thesis, we proposed a dynamic algorithm to adapt the vertical partitions to the change of workload.

## 2.5 Dynamical Vertical Partitioning

To our best knowledge there have been no attempts to investigate the applications of dynamic vertical partitioning, i.e. partitioning that changes its schema together with the changing amount and characteristics of database workload to the automated performance tuning of relational database systems. However, the knowledge of database workload monitoring, analysis and the way to implement database tuning automatically assist us to improve query performance dynamically by vertical partitioning.

### 2.5.1 Workload Monitoring and Analyzing

The basic idea behind automated performance tuning is a feedback control loop that consists of *observation*, *prediction*, and *reaction*. Self tuning of a database server should be based on the feedback control loop where a database system continuously observes the performance indicators and dynamically adjusts the values of critical system parameters to the current workload characteristics [45]. The solution proposed in this thesis is based on a feedback loop. We observe the change of workload to detect a relatively low workload time. Then we predict the coming workload based on the characteristics of current workload and implement the new vertical partitions.

Capturing a moment when a significant change of workload happens is an important aspect of self tuning systems. Much work has been done in the area of monitoring, analysing and detecting changes in a database workload. A framework has been designed to collect information about the behaviour of a database system and how to



facilitate self-tuning and adaptive actions [10]. A system QUIET [40] has been proposed to gather workload information and to dynamically choose the best indexing schema for the currently running applications. The system considers the continuous workload as an isolated unit and dynamically updates benefit information for each one of the candidate indices.

An ordering of SQL statements is an important hint for performance tuning [5]. This approach treats the workload as a sequence of steps and identifies two sorts of workload: query workload and update workload. By exploiting an order between the query and update statements, a tuning system automatically creates and removes the physical structures that improve performance of the system.

### 2.5.2 Implementing Vertical Partition

Implementation of a vertical partition as a composite key index is possible when a query processor is able to detect and to apply the index-only processing of a query. In such a case, a query is processed by the horizontal traversal of a leaf level of B\*-Tree index without access to a relational table. At a logical level it is equivalent to a sequential scan of a vertical partition. As long as a partition is implemented as an index over a relational table, the query processor is able to invoke a horizontal traversal of index automatically.

The advanced query processors in the commercial database management systems are capable of rewriting a query such that a materialized view is used instead of a relational table when it improves the performance of query processing. As a consequence, the mechanism is completely transparent to query processing and it allows for the dynamic implementation at vertical partitions.

## 2.6 Summary

Nowadays more complicated database systems require automated database management systems. There is still much work to do to reduce the high need for human intervention such as human input and intelligence, need for dynamic adaption, the lack of ability to reset DBMS parameters, the lack of analytical capabilities, no smart maintenance strategies and so on. Dynamic vertical partition is a technique that contributes to self-optimization and self-organization features of the autonomic database systems. In this thesis, we propose a solution to achieve query optimization based on dynamic vertical partition.

# Chapter 3

---

## Static partitioning

This chapter proposes a new algorithm that uses a suboptimal vertical partitioning of relational tables under the constraint that a certain level of redundancies is acceptable in a database. The algorithm is based on a new cost model, which precisely estimates I/O throughput as the total number of physical read/write database operations required to implement a given workload. The solution described in this chapter transforms a schema of relational database into a partitioned one and decides which components of the original schema should be replicated as the separate partitions. The experiments conducted confirm the correctness of the cost model used by the vertical partitioning algorithm and demonstrate the expected performance gains from the partitioning.

The chapter is organised as follows. Section 3.1 defines the problem and describes the fundamental concepts of the solution. Section 3.2 presents a cost model of the relational database, which easily calculates the cost for a representative workload. Section 3.3 presents the algorithm that finds the suboptimal partitions for the given workload with a given level of redundancies. The results of our experiments and an evaluation are included in section 3.4. Finally, section 3.5 concludes the paper and outlines the future work.

### 3.1 Preliminaries

This section reviews the basic concepts of vertical partitioning of relational database schemas.

**Definition 1** *Let  $A = \{a_1, \dots, a_n\}$  be a set of attribute names. Then let database schema be a set of relational schemas  $S = \{s_1, \dots, s_n\}$  such that  $\forall i = 1, \dots, n (s_i \subseteq A$  and  $s_i \neq \emptyset)$ .*

The relational schemas are normalised to the best extent for an adopted model of data dependencies. Typically, BCNF or sometimes 3NF is enough when considering only a class of functional dependencies, and 4NF is sufficient when extending a model with a class of multivalued dependencies.

**Definition 2** *Let  $T = \{t_1, \dots, t_n\}$  be a set of SQL statements. Let a frequency  $f_i$  of a statement  $t_i$  represents the total number of times the statement has been submitted for the execution by the database applications in a given period of time. Then, the total workload imposed on a database with a schema  $S$  is defined as a set  $W_S = \{w_1, \dots, w_n\}$  where each  $w_i = (t_i, f_i)$ .*

A workload on a schema may be alleviated by using index, horizontal partition besides vertical partition. However, in some cases, the majority of the workload on a schema is a full table scan. In such a case, the workload will not benefit from indexing and horizontal partitioning at all. In our thesis, we try to optimise the full table scan workload by vertical partitioning. A typical workload consists of two types of SQL statements: queries and data manipulation statements. The queries include single relation queries and multi-relation queries such as join and antijoin queries, and set algebra expressions. Multi-relation queries contribute to the scans of relational tables in the same way as single relation queries. As a consequence the frequency of a query is calculated as the summation of times a relational table is scanned by the single and multi-relation queries. Furthermore, some of data manipulation statements contribute to the scans of relational tables due to **SELECT** statements included in the multilevel **WHERE** clauses of **UPDATE**, **DELETE**, and **INSERT INTO ... (SELECT ...)** statements.

**Definition 3** *Let  $2^S$  be a set of all database schemas created over a set of attributes  $A$ . Then, a vertical partitioning is defined as a mapping  $P : 2^S \rightarrow 2^S$  that transforms a database schema  $S = \{s_1, \dots, s_n\}$  into a database schema  $S' = \{s'_1, \dots, s'_m\}$  such that  $\forall s'_i \in S' \exists s_j \in S (s'_i \subseteq s_j, k_j \subseteq s'_i \text{ where } k_j \text{ is a primary key of } s_j, \text{ and } \cup_i s_i = \cup_i s'_i)$ .*

An elementary step of partitioning extracts from a relational schema  $s$  a new schema  $s'$  such that  $s' \subset s$  and primary key of  $s$  is included in  $s'$ . A relational schema  $s$  can be dropped during a partitioning process if there exists the schemas  $s_1, \dots, s_k$  such that primary key of  $s$  is included in each one of  $s_1, \dots, s_k$  and  $\cup_i s_i = s$ . In our approach, elimination of the relational schemas which are the supersets of smaller schemas is not compulsory. For example, a database schema that consists of the relational schemas  $s_1 = \{a, b, c\}$ ,  $s_2 = \{a, b\}$ , and  $s_3 = \{a, c\}$  is acceptable if the large number of queries

frequently scan each one of the schemas  $s_1, s_2, s_3$ . Besides, dropping of  $s_1$  is possible, however it may not be beneficial due to the frequent joins of  $s_2$  and  $s_3$  in order to restore  $s_1$ . We assume, that the reduced time of query processing compensates the costs of redundancies and replications of data manipulation statements. It is also important to note, that even though this kind of partitioning introduces the redundancies into a database, it has no negative impact on the normalisation of relational schemas because no two schemas are merged at any stage of the process.

**Definition 4** *Consider a database schema  $S$  and workload  $W_S$  imposed on a database with a schema  $S$ . The storage requirements of implementation of schema  $S$  for the particular contents of a database are denoted by  $\text{storage}(S)$  and are measured as the total number of megabytes of persistent storage needed to keep the data.*

**Definition 5** *The costs paid for the implementation of a workload  $W_S$  i.e. the costs of processing queries and data manipulation statements are denoted by  $\text{cost}(W_S)$  and are measured as the total number I/O operations required to implement the workload. The detailed analysis of the implementation costs is provided in the next section.*

**Definition 6** *Given a database schema  $S$  and workload  $W_S$ . A problem of optimal unconstrained vertical partitioning is to find a vertical partitioning  $S' = P(S)$  such that  $\text{cost}(W_{S'}) < \text{cost}(W_S)$  and  $\nexists S'' (\text{cost}(W_{S''}) < \text{cost}(W_{S'}))$ .*

If an extreme situation workload  $W_S$  includes only queries and no insertions, updates, and deletions then a simple solution to the problem is to add to the original database schema the new schemas that optimise the full scans of the projections of the relational tables. If we additionally consider the data manipulation operations then an optimal solution can be found through merging of the new schemas. Let, workload  $W_S$  includes  $n$  queries  $q_1, \dots, q_n$  each scanning a relational table created over a respective relational schema  $s_1, \dots, s_n$ . If a query  $q_i$  for  $i = 1, \dots, n$ , accesses only the attributes in a subset  $s'_i$  of a full schema  $s_i$  of a relational table  $r_i$  then in the first step we create  $n$  relational tables  $r'_1, \dots, r'_n$  over the schemas  $s'_1, \dots, s'_n$ . If an additional load created by the data manipulation operations replicated in the new relational tables exceeds the benefits from query processing, i.e. the total number of additional I/O operations spent on implementation of data manipulations is larger than the total number of I/O operations saved on query processing, then we reverse partitioning by merging the schemas  $s'_1, \dots, s'_n$ . This process is a part of an algorithm discussed later in the paper. Note, that unconstrained vertical partitioning does not eliminate original relational schemas and because of that it does not increase the costs of join operations.

**Definition 7** *A problem of optimal constrained vertical partitioning is to find a vertical partitioning  $S' = P(S)$  such that  $\text{cost}(W_{S'}) < \text{cost}(W_S)$  and  $\nexists S'' (\text{cost}(W_{S''}) < \text{cost}(W_{S'}) \text{ and } \text{storage}(S'') \leq c_{\max})$  where  $c_{\max}$  is a constraint imposed on the total amount of persistent storage available for the implementation of schema  $S'$ .*

A constraint imposed on the total amount of additional persistent storage available for the vertical partitions makes the problem more realistic and significantly increases its complexity. As finding an optimal solution is NP-hard problem we weaken it to a suboptimal solution.

**Definition 8** *A problem of the suboptimal constrained vertical partitioning is to find a vertical partitioning  $S' = P(S)$  such that  $\text{cost}(W_{S'}) < \text{cost}(W_S)$  and  $\text{storage}(S') \leq c_{\max}$ .*

In this case we are interested in a solution that increases the performance of query processing at a rate higher than the costs from increased processing of data manipulation statements and it does not exceeds a given amount of persistent storage.

## 3.2 Cost Model

A central point in any algorithm used to find the optimal or suboptimal partitioning is the quality of the cost model being used to calculate a value  $\text{cost}(W_S)$  for a given workload  $W_S$  imposed on a database with a schema  $S$ .

The iterations towards the optimal partitioning are conducted by moving from one database schema to another in an attempt to reduce the data processing costs and to stay below a given threshold of persistent storage.

An elementary step of this procedure, i.e. the choice of the next best database schema from a given set of options is determined by the largest reduction of  $\text{cost}(W_S)$  for any new database schema. This is why, the precise estimation of  $\text{cost}(W_S)$  at each stage is so important for the quality of the entire process.

In this paper we measure  $\text{cost}(W_S)$  as the total number of read and/or write data block operations needed to compute a workload  $W_S$  over a database schema  $S$ . A workload consists of query processing component  $Q_S$ , data entry component  $I_S$ , update  $U_S$ , and delete  $D_S$  components. The data manipulation components contribute to a query processing component when the respective SQL statements contain **SELECT** statement in their bodies, e.g **INSERT INTO T (SELECT ...)** statement which inserts

into a relational table  $T$  the rows retrieved by **SELECT** statement. Hence,  $cost(W_S) = cost(Q_S) + cost(I_S) + cost(U_S) + cost(D_S)$ .

To calculate  $cost(Q_S)$  we decompose the multi-queries into a collection of single queries and we add up the frequencies which exclude the times of the repeating reads from caching. Currently, The DBMS stores information in memory caches and on disk. Memory access is much faster than disk access. For this reason, many cache algorithms like Least Recently Used (LRU), Most Recently Used (MRU), Least Frequently Used (LFU), etc. have been proposed to store frequently accessed objects in the memory, rather than requiring disk access. Therefore,  $cost(Q_S)$  does not include the cost of memory access for the selection query.

$$cost(Q_S) = \sum_i cost(Q_S^{(i)}) * (f_i - c_i)$$

where  $cost(Q_S^{(i)})$  is the total number of I/O operations needed of query  $Q^{(i)}$  over a database schema  $S$ .

To calculate  $cost(I_S)$ ,  $cost(U_S)$ ,  $cost(D_S)$  we decompose each data manipulation statement into an operation part and query part. As the query parts have been already considered in query  $cost(Q_S)$ , the cost of the operation part is evaluated in the following ways. The costs of insertions are determined by a formula:

$$cost(I_S) = \sum_i (cost(I_S^{(i)}) * f_i)$$

where for each insertion  $cost(I_S^{(i)})$  is equal to  $N * (1 + \log_f K) + 2$  Insertion into a relational table needs an insertion into a B-tree index over an index key and two operations (read and write) on a data block to insert a row into the table. If an insertion affects  $N$  schemas then it has to be repeated  $N$  times.

The costs of updates are determined by a formula:

$$cost(U_S) = \sum_i (cost(U_S^i) * f_i)$$

When the number of updated rows in the first partition is small then  $cost(U_S^{(1)})$  is equal to the total number of I/O operations needed to scan the first partition and for  $i > 1$   $cost(U_S^{(i)})$  is equal to  $((1 + \log_f K) + 2) * n * (M - 1)$  where  $n$  is the total number of rows updated in the first partition and  $M$  is the total number of partitions to be updated and  $f$  denotes a fanout of B-tree implementation of an index,  $k$  denotes the total number of keys in an index. When the total number of updated rows in the first partition is large then  $cost(U_S^{(i)})$  for  $i \geq 1$  is equal to the total number of I/O operations needed to scan all partitions that need to be updated.

The deletions are performed in the same way as the updates. The costs of deletions are determined by a formula:

$$cost(D_S) = \sum_i (cost(D_S^{(i)}) * f_i)$$

When the number of deleted rows from the first partition is small then  $cost(D_S^{(1)})$  is equal to the total number of I/O operations needed to scan the first partition and for  $i > 1$   $cost(D_S^{(i)})$  is equal to  $((1 + \log_f K) + 2) * n * (M - 1)$  where  $n$  is the total number of rows deleted in the first partition and  $M$  is the total number of partitions to be considered and  $f$  denotes a fanout of B-tree implementation of an index,  $k$  denotes the total number of keys in an index. When the total number of deleted rows in the first partition is large then  $cost(D_S^{(i)})$  for  $i \geq 1$  is equal to the total number of I/O operations needed to scan all partitions that are affected by the deletions.

### 3.3 Partition Configuration

Finding the optimal partitioning for a given workload  $W_S$  and a given database schema  $S$  is NP-complete problem.

A hypothetical solution that always finds the optimal partitioning would start with the generation of all combinations of attributes from a relational schema and then repeating this process for all relational schemas in a database schema. Next, we would compute  $cost(W_S)$  for all combinations of the sets of attributes found in the previous step so that each combination covers an entire database schema. The optimal solution is the one that minimises  $cost(W_S)$ . The complexity of such an algorithm would be  $O(n^2)$  where  $m$  is the total number of attributes in a database schema.

The partitioning algorithms presented in this section speed up this process by starting from the original schema  $S$  of a database and using the greedy partition selection algorithm to find a partition configuration that minimises the cost of the workload at each step.

#### 3.3.1 Algorithm overview

Figure 3.1 shows the main steps of the algorithm. At the beginning the algorithm generates the candidate partitions for each query and data manipulation task included in a given workload. Next, it calculates the benefits from the separate implementation of each one of the partitions taken from a candidate set. A partitioning that returns the

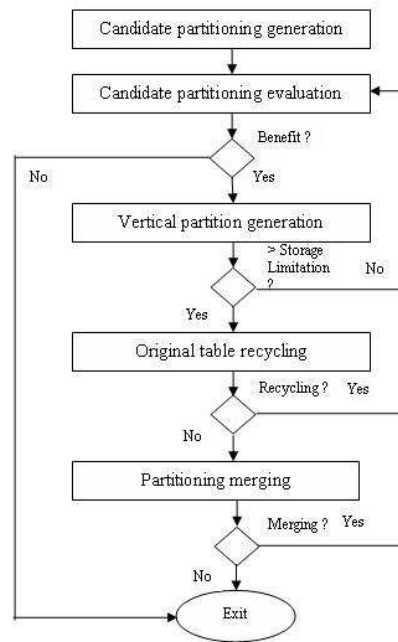


Figure 3.1: Outline of algorithms

highest benefits is accepted for the implementation. As we keep all the original tables in a database, the implementations of the new partitions require the additional amounts of persistent storage. The algorithm repeats the allocations of the new partitions from a collection of partitions computed at the beginning as long as the total amounts of additional persistent storage do not exceed a given threshold. When that happens, the algorithm recycles one of the relational tables from the original database. A recycled space is used to implement the next partition. The algorithm stops when either all partitions are implemented or when we no longer benefit from the implementation of a new partition.

### 3.3.2 Choosing the partitions

A greedy algorithm (shown as Algorithm 1) controls the selection of the partitions from a set of candidate partitions generated at the beginning of the algorithm. A partition is chosen when it provides the highest benefits from a collection of the still remaining partitions and its implementation does not exceed the assumed storage threshold. Once the new configuration is generated, the algorithm recalculates the cost of the candidate partition which is a subset of a new partition or includes the subset of a new partition.



```

Initialization;
Set candidate partition schemas set of Workload CP =  $\{cp_1, cp_2, \dots, cp_n\}$ ;
Calculate the benefit of each candidate partition and Sort them;
repeat
    if max benefit candidate partition  $cp_{max} > 0$  then
        Add  $cp_{max}$  into partition configuration P to generate new partition
        configuration  $P'$ ;
        Remove  $cp_{max}$  from CP;
        Calculate the benefit of every candidate partition;
        Find the maximum benefit of candidate partition  $cp'_{max}$  for new Partition
         $P'$  ;
    else
        | exit;
    end
until ( $Storage(P') + Storage(cp'_{max}) > c_{max}$ ;
Run Algorithm 2;
Algorithm 1: Partition greedy selection algorithm

```

### 3.3.3 Recycling of the original tables

The original tables covered by the new partitions are kept in a database as long as we do not run out of storage for the implementation of the new partitions. Such a strategy allows for the effective utilisation of cheap persistent storage and as long as it is possible, eliminates the expensive computations of join operations when two or more partitions are used in the same query. At this stage, according to the current configuration, we can easily evaluate the join cost for the current workload and decide whether or not to recycle the space of origin table to enhance performance. The complementary attribute set of original table T is denoted as  $\overline{T}$ . After removing an original table from the partition configuration, we will add  $\overline{T}$  to make sure no any attribute of T will be lost. Algorithm 2 shows the procedure of recycling the original tables.

---

```

foreach original Table  $T_k$  in  $D$  do
  foreach candidate partition  $cp_i \subset T_k$  do
    if  $\text{Storage}(\text{new partition } P') \leq c_{max}$  then
      | Calculate the benefit of  $cp_i$ ;
    end
  end
  Find the maximum benefit of candidate partition  $cp_{max}$ ;
  if there exists  $cp_{max} > 0$  then
    | Add candidate partition  $cp_{max}$  into partition configuration  $P$ ;
    | Remove  $cp_{max}$  from  $CP$ ;
    | Remove  $T_k$  from partition configuration; Add  $\overline{T_k}$  into partition
    | configuration  $P$ ;
    | Set  $\text{Flag} = \text{True}$ ;
  end
end
if there is some space left and  $\text{Flag} = \text{True}$  then
  | run Algorithm1;
end

```

**Algorithm 2:** Table recycling algorithm

### 3.3.4 Merging the partitions

At the final stage we merge the pairs of very similar partitions in order to save some disk space for further partitioning. This algorithm enhances the quality of partitioning by merging the similar partitions.

```

foreach original Table  $T_k$  do
  foreach  $p_i, p_j \in T_k$  do
    if benefit of merging  $p_i, p_j > 0$  then Merge( $p_i, p_j$ );
    Run algorithm 1;
    else if benefit of (merging  $p_i, p_j$  and adding  $cp_{max}$  ) and new Partition
       $\leq c_{max} > 0$  then
        Merge( $p_i, p_j$ );
        Run algorithm 1;
      else
        exit;
      end
    end
  end
end

```

**Algorithm 3:** Partition merging algorithm

## 3.4 Experiments

The vertical partitioning algorithm has been implemented and tested on a sample database containing synthetically generated data. The experiments involved a number of various distributions of the database loads. All experiments have been conducted with the off-the-shelf, commercially available database server Oracle 10g, release 1 running on a single processor 2-GHz Intel CPU box with 512 MB of main memory and a 40-GB hard drive. A sample database implemented TPC-R[28] benchmark database with data generated accordingly to the benchmark specifications.

The TPC-R database comprises 8 tables. We experimented with the workloads consisting of 20 queries accessing two of the largest relational tables in a sample database. The size of the largest table is about 2 Gbytes and its schema consists of 16 attributes. The size of the second largest table is about 650MB and its schema consists of 9 attributes. Both tables have the non-clustered B-tree indexes automatically constructed on the primary keys.

### 3.4.1 Evaluation of partitioning

In the first experiment, we generated two partition configurations, VPMY and VPQO respectively, with our cost model and a cost model provided by Oracle query optimizer. We compared these two configurations with the original table ORIG. The figure 3.2

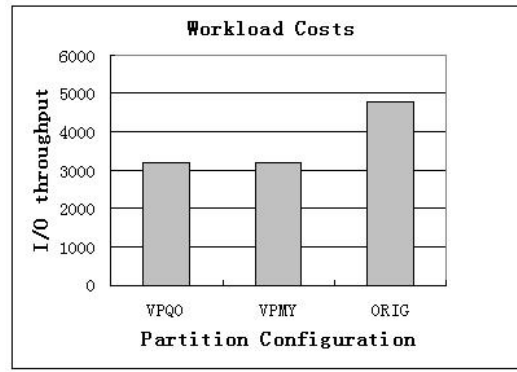


Figure 3.2: Storage and performance

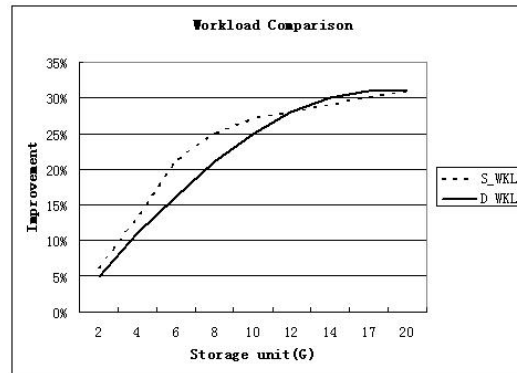


Figure 3.3: Performance comparison

shows that the partition configurations generated by both cost models got the same benefit. Meanwhile, the performance of the new partition configuration is enhanced about 30%.

### 3.4.2 Trend of Storage and Benefit

In the second experiment, we defined two quite typical workloads. The first one has many similar schemas to those in the workload *S\_WKLD* while the other *D\_WKLD* has only a few similar schemas. We allocated the same storage threshold for both workloads. Figure 3.3 shows that:

1. the performance is enhanced with the storage  $c_{max}$  increasing;
2. with the storage boundary increasing, the performance improvement trends for the workloads in a different similarity level.

### 3.5 Summary and open problem

This chapter introduces a new algorithm that vertically partitions the relational tables according to a given workload and accordingly to a given limit of redundancies acceptable in a database. The contributions of the chapter are as follows:

1. we show how to perform a detailed analysis of the costs of query and data manipulation processing over a given configuration of a relational database and we compare the analytical results with the results obtained from a cost-based optimizer of a commercial relational database server,
2. we propose a new algorithm for vertical partitioning of relational schemas in a database system with a given level of redundancies

However, there are some limitations of static partitioning algorithms. Firstly, the configuration conducted from a static partitioning algorithm cannot optimise the database system automatically. The workload fluctuates dramatically from time to time. A configuration which optimises the database system at present may not do so in future. Furthermore, we should not create the configuration during a heavy workload because it will place a burden on the system. One of solutions to optimise a dynamic workload is to create the vertical partitions for the heavy workload while in the low workload time.

Secondly, the query processing subsystems of the commercial database servers cannot detect which vertical partitions are the constituents of which relational tables. Currently query optimizers consider only relational tables, indexes and relational views. Vertically partitioned relational tables are not allowed for generating better query processing plans. Therefore, if we implement the vertical partitions as relational tables, we have to adjust the applications and compile them every time the configuration changes. If we implement the vertical partitions as relational tables and then we use the relational views to compose the original relational tables, the query optimizer is not smart enough to find the best length path to find the records. For example, we have an original table  $T1(A,B,C)$  and we reorganise table  $T1$  as vertical partitions  $P1(A, B)$ ,  $P2(A,C)$  and view  $V1(P1.A, P1.B, P2.C)$ . If we have a query on  $V1.B$ , then the query optimizer still joins two partitions first, and is not smart enough to take  $P1$  only.

To make the query optimizers smart enough to use the vertical partitions, we implement the partition as an index. The optimizer will consider a fast full scan of the index which represents the partition without changing the applications.

---

In the next chapter, we will introduce an algorithm that will dynamically optimise the workload on a schema by indexing a partition based on the static partitioning algorithm.

# Chapter 4

---

## Dynamic Partitioning

We begin this chapter by introducing the periodical behaviors of the database workload. Then, we define the basic concepts for detecting the potential low workload time. When a potential low workload time comes, we start to search for all similar workloads. If there exists a similar workload, we use a greedy algorithm to find a sub-optimal configuration for a high workload. When we get the configuration, we may schedule the implementation to avoid affecting the performance of the low workload time. The chapter finishes with some experiments and a conclusion for dynamic partitioning.

### 4.1 Workload Patterns

Workloads change from time to time, however the changes are not totally random. The order of database access is restricted by business rules or business logic, which results in database workload changes that appear cyclical over time. For example, in the morning, a commercial system will start with reviewing all the transactions that happened yesterday, then it handles the exceptional transactions or approves successful transactions. Therefore, the business rules and business logic of a database system make the workload behave in a strongly periodic way. From the structure of a program view, many programs are written in a modular way and they are composed by a set of procedures contained in a loop. These loops in the application make the workload behave periodically. As well, database administrators may arrange some job lists to run in a specific time which contributes to the system's periodic performance. In this thesis, we assume that a workload cannot rapidly change. Rapid changes occur only when certain events happen, otherwise the system is stable.

In a database system, some unexpected events cause a workload to show an unusual pattern. In some cases, we may not find any similar present or past patterns. For example, when a table crashes it takes some time to rebuild it. The queries based on the tables cannot be executed and will have to wait until the table is available. In such

a case, the workload changes are unpredictable and similar workloads cannot be found. In some other cases, the exceptional event could have happened before and we may find a similar workload based on the characteristics of these accidental events. For this thesis, the algorithm may not be necessary to find a similar workload when unexpected events happen.

## 4.2 Preliminaries

This section introduces the basic concepts of dynamical vertical partitioning of relational database schemas.

**Definition 9** Let  $A = \{a_1, \dots, a_n\}$  be a set of attribute names. We say that a database schema is a set of relational schemas  $R = \{r_1, \dots, r_n\}$  such that  $\forall i = 1, \dots, n (r_i \subseteq A$  and  $r_i \neq \emptyset)$ .

**Definition 10** Let  $2^R$  be a set of all database schemas created over a set of attributes  $A$ . Then, a vertical partitioning is defined as a mapping  $P : 2^R \rightarrow 2^R$  that transforms a database schema  $R = \{r_1, \dots, r_n\}$  into a database schema  $R' = \{r'_1, \dots, r'_m\}$  such that  $\forall r'_i \in R' \exists r_j \in R (r'_i \subseteq r_j, k_j \subseteq r'_i$  where  $k_j$  is a primary key of  $r_j$ , and  $\cup_i r_i = \cup_i r'_i)$ .

**Definition 11** An execution  $e$  is a quadruple  $(s, t_s, t_e, u)$  where  $s$  is SQL statement,  $t_s$  and  $t_e$  represent respectively the times the statement  $s$  starts and ends,  $t_s < t_e$ . A statement  $s$  is executed on behalf of a user  $u$ .

**Definition 12** A tracefile  $T$  is a set of executions  $\{e_1, e_2, \dots, e_n\}$ .

**Definition 13** Let  $s_i, s_j$  be SQL statements. We say that  $s_i$  is equivalent to  $s_j$  when a query execution plan of  $s_i$  is the same as that of  $s_j$ . A query execution plan is an extended relational algebra expression parsed by the query optimizer. An extended relational algebra includes the operation of selections, projections, joins, divisions and groupings.

For example, given queries Q1 and Q2 on table T(a, b, c, d) as follows:

Q1: SELECT a, b FROM T WHERE c = 100;

Q2: SELECT a, b FROM T WHERE c > 72 and c < 100;



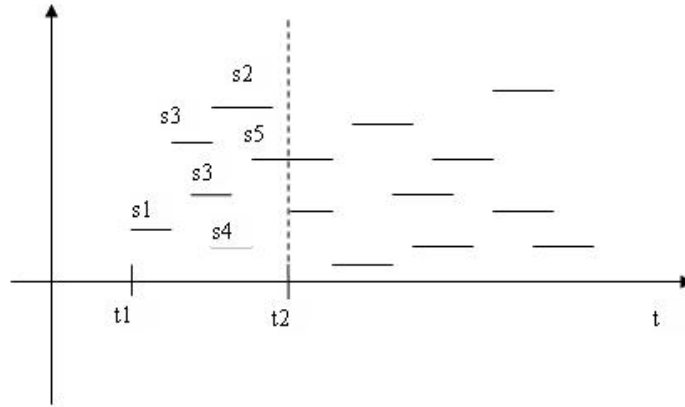


Figure 4.1: Total workload

The relational algebra of Q1 and Q2 is a projection of T on (a,b,c). Therefore, Q1 is equivalent to Q2.

**Definition 14** Let  $load(s)$  denotes an amount of workload imposed on the system by the execution of a statements. For any statements if  $s$  has been entirely executed within a period  $(t_{from}, t_{to})$  then it contributes with  $load(s)$  to the total workload, otherwise it contributes with  $(g/c) * load(s)$  which  $g$  is the length execution time in the period and  $c$  is the total execution time of  $s$ ;

**Definition 15** Let  $S = s_1, s_2, \dots, s_k$  be a set of SQL statements. The total workload imposed on a database with schema R within a given time period  $(t_m, t_n)$  is defined as a set  $W_R^{(t_m, t_n)} = \cup_{i=1..k} (load(s_i) * g_i / c_i)$ .

For example, Figure 4.1 shows that in duration  $(t_1, t_2)$ , the system completely executed the statements  $s_1, s_2, s_3, s_4$  and half of  $s_5$  and such that  $s_2$  is equivalent to  $s_3$  then the total workload is  $load(s_1) + load(s_2) + 2 * load(s_3) + load(s_4) + 0.5 * load(s_5) = load(s_1) + 3 * load(s_3) + load(s_4) + 0.5 * load(s_5)$ .

**Definition 16** Let  $\delta$  be a unit of time used to measure a workload. Then we denote an amount workload over  $\delta$  as unit workload  $W_\delta$ .

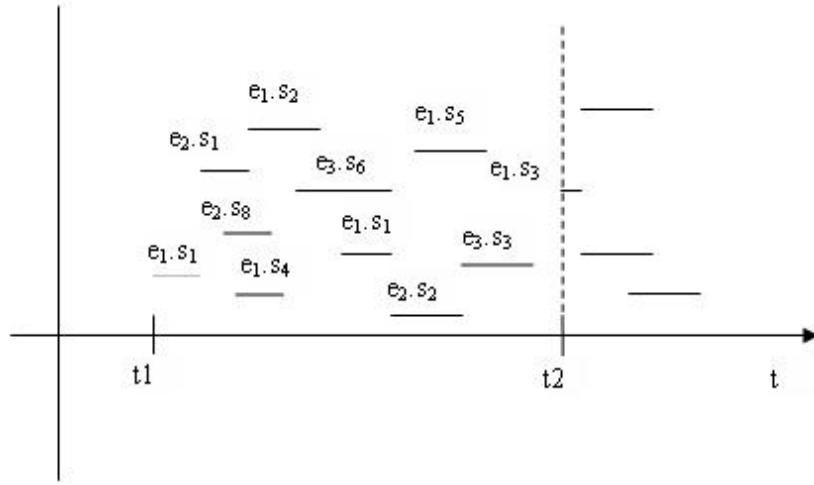


Figure 4.2: Example of session and signature

**Definition 17** We call a given period  $(t_m, t_n)$  as low workload time, when  $\forall i = 0, \dots, n (W_{\delta_i} \leq v_{low})$  and  $\delta_i$  included in  $(t_m, t_n)$  and  $v_{low}$  is a minimum support value of low workload.

**Definition 18** We call a given period  $(t_m, t_n)$  as high workload time, when  $\forall i = 0, \dots, n (W_{\delta_i} \geq v_{high})$  and  $\delta_i$  included in  $(t_m, t_n)$  and  $v_{high}$  is a minimum support value of low workload.

**Definition 19** We define a session  $c$  as a sequence of executions  $\langle e_1, \dots, e_n \rangle$  in a duration from  $t_i$  to  $t_j$ .  $\forall i \leq n, e_i = (s_i, t', t'', u)$ .

**Definition 20** We define a signature of a workload  $W_{(t_i, t_j)}$  as a set of sessions  $\{c_1, \dots, c_x\}$  that occurred in duration  $(t_i, t_j)$ .

As shown in Figure 4.2, in  $(t_1, t_2)$ , a session  $c_1$  of user  $e_1$  is  $\langle s_1, s_4, s_2, s_1, s_5, s_3 \rangle$ ;  $c_2$  of user  $e_2$  is  $\langle s_1, s_8, s_2 \rangle$ ;  $c_3$  of user  $e_3$  is  $\langle s_6, s_3 \rangle$ ;

A signature of a workload is a set of sessions  $(c_1, c_2, c_3)$ .

## 4.3 Cost Model

As discussed in chapter 3, the quality of any algorithm relies on the quality of the cost model being used to calculate a value  $cost(W_R)$  for a given workload  $W_R$  imposed on a database with a schema  $R$ .

In this chapter, the measurement of  $cost(W_R)$  is the same as cost model in static partition, which is the total number of read and/or write data block operations needed to compute a workload  $W_R$  over a database schema  $R$ . The workload consists of the query processing component  $Q_R$ , the data entry component  $I_R$ , update  $U_R$ , and delete  $D_R$  components. The data manipulation components contribute to a query processing component when the respective SQL statements contain a **SELECT** statement in their bodies, e.g. **INSERT INTO T (SELECT ...)** statement which inserts into a relational table **T** the rows retrieved by **SELECT** statement. Hence,  $cost(W_R) = cost(Q_R) + cost(I_R) + cost(U_R) + cost(D_R)$ . Detailed calculation of cost model can be referred to chapter 3.

## 4.4 Overview of Dynamic Partitioning Algorithm

The objective of the dynamic partitioning algorithm is to balance system workloads. As shown in Figure 4.3, at some durations, there are excessive applications access to the database which causes the database system to work extremely slow. At other durations, there are only a few applications access to the database system while many resources of the system are wasted. To improve the performance in busy time and to fully use the resources in idle time, we may generate the optimal configuration for high workload time during low workload time. Figure 4.4 shows the dynamic partitioning algorithm have this result.

Figure 4.5 shows the main steps of the dynamic partitioning algorithm. A starting point to the dynamic partitioning algorithm is the continuously repeated evaluation of a workload level imposed on the system by the database applications. Next, we search for the best match for current and previous workload by comparing the signatures of current workload and the previous low workloads. If the most similar workload can be found, we analyse its historical executions to find what query and data manipulations will be performed. The analysis of anticipated an database operation is used to generate a set of vertical partitions. Finally, the implementation of partitions will carry on until all partitions are implemented or the workload during implementation increases above  $v_{high}$ .

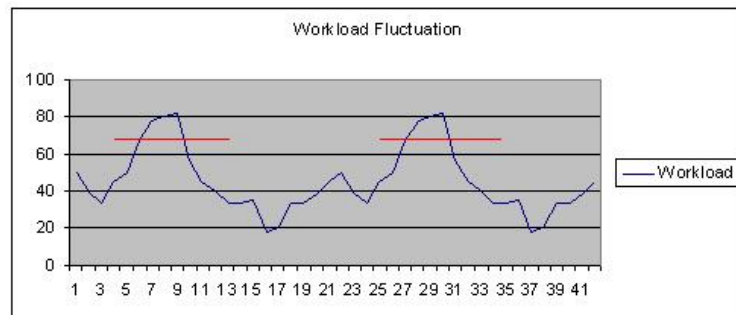


Figure 4.3: Workload before dynamic partitioning

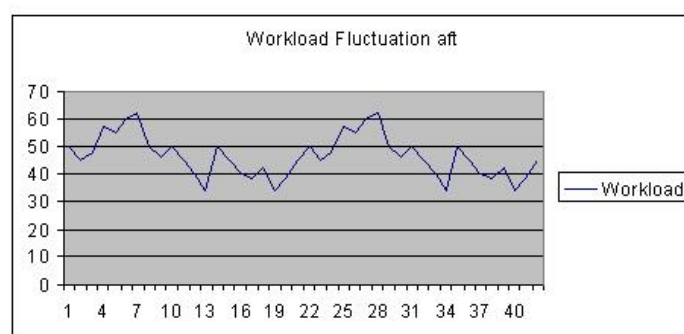


Figure 4.4: Workload after dynamic partitioning

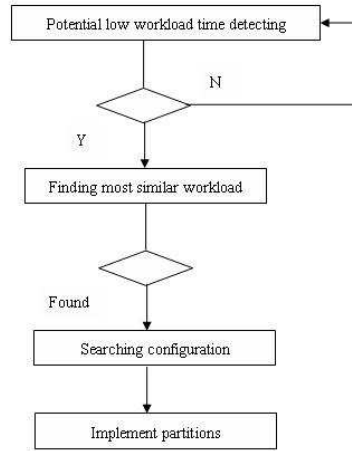


Figure 4.5: Overview of dynamic partitioning

## 4.5 Potential Low Workload Time Detecting

An objective of this procedure is to detect the periods of time when the system is relatively idle and when it is possible to find which new vertical partitions will be needed in the future and which can be discarded. Evaluation of a workload is performed at every single unit of time  $t$ . If a workload at the continuous  $k$  units is below  $v_{low}$  then the system starts a procedure that tries to anticipate how much time of a low workload is left and what changes to the vertical partitioning should be done. If at any of the following time units a workload increases above  $v_{low}$  then re-partitioning procedure is stopped and the most up to date state of vertical partitioning is preserved at that point in time.

```

input  : ArrayofUnitWorkload WArr, support  $v_{low}$ 
output: Boolean

Set Flag = True;
for  $i = 1$  to WArr.last do
    if WArr[ $i$ ]  $\leq v_{low}$  then Return True;
    else
        Return False;
        Exit
    end
end

```

**Algorithm 4:** Detecting a potential low workload time algorithm

## 4.6 Finding Most Similar Workload

The identification of new partitions and the ones that are no longer needed starts from finding in the history of executions all moments which are the most similar to the present one. There are four steps finding the most similar workload time in the past executions. To find the most similar workload we analyse the contents of a trace file and we pick up the moments when  $k$  continuous unit workload was below  $v_{low}$ . Next, we compare the signatures between the occurrences found and the present workload and then pick up the one with highest similarity level. If there are several signatures having the same similarity as the current one, we choose the most recent workload. If there is no workload similar to the current one then the repartitioning procedure is terminated. When the most similar workload is found we analyse the history of executions following the workload in order to find what query and data manipulation operations will be performed in the future.

First, we introduce the function to calculate the similarity of the two signatures.

### 4.6.1 Function to calculate similarity of two signatures

Given two signatures, we can calculate the similarity rate by comparing the characters in the signatures. The more different characters the two signatures have, the less similar they are. Note that "-" means difference,  $\cup$  means set union and " $||$ " means the number of the elements in a set.

**input** : Signature  $s1, s2$

**output**: Similarity Ratio

Set  $i = 0$ ;

$i = 1 - (|(s1 - s2)| + |(s2 - s1)|) / |(s1 + s2)|$ ;

Return  $i$ ;

**Algorithm 5**: Calculating Similarity of Signatures algorithm

## 4.6.2 Most Similar Workload Searching

In the tracefile, we pick up the moments when  $k$  continuous unit workload is below  $v_{low}$ . Then we compare the similarity in signatures of the current workload with signatures of the workloads we found. If there are several signatures similar to current one, we choose the most recent. If there is no similar workload as is found, we will not consider reconfiguring the partitions. The following algorithm shows how to search for the most similar workload.

**input** : Current Signature  $cs$ , ArrayofSignature  $AS$ , support of similarity  $ss$

**output**: Signature  $s$

```

for  $i = 1$  to  $AS.last$  do
    if  $Similarity(AS[i], cs) < ss$  then
        | Remove  $AS[i]$ 
    end
end
if  $AS$  is not Empty then
    | Sort  $AS$  by Similarity and time;
    | Set  $s = AS[1]$ ;
end

```

**Algorithm 6**: Searching Most Similar Workload

## 4.7 Finding configuration

### 4.7.1 Choosing partitions

The information about the anticipated database operations found while searching for the most similar workload is used to generate a set of candidate vertical partitions.

A vertical partition becomes a candidate when it provides the highest benefits from a collection of the still remaining candidate partitions and its implementation does not exceed the assumed storage threshold and time limitation.

**input** : ArrayofPartition AP

**output**: ArrayofPartition P

Variable(*ArrayofTriple*: CP, which  $cp_i = (api, benefit, size)$ );

**repeat**

**for**  $i = 1$  to CP.last **do**

**if**  $CP[i].time + TimeofBuilding(P) > TimeLimit$  **then**

            Remove CP[i]

**end**

**end**

**if** CP is Empty **then**

        Exit;

        Run Merge Algorithm;

**end**

**for**  $i = 1$  to CP.last **do**

**if**  $CP[i].size + Sizeof(AP) > Storage$  **then**

            Remove CP[i]

**end**

**end**

**if** CP is Empty **then**

        Exit;

        Run Algorithm;

**end**

**for**  $i = 1$  to CP.last **do**

        Calculate the benefit for CP[i];

**end**

**if** CP is Empty **then**

        Exit;

        Run Merge Algorithm;

**end**

    Sort CP by benefit;

    Add CP[1] into P;

    Remove CP[1] from CP;

**until** (AP is null);

**Algorithm 7:** Partition greedy selection algorithm



## 4.8 Merging the partitions

In this step, we merge the pairs of similar partitions in order to save some disk space for further partitioning. To evaluate the similarity of two partitions, we can use the following formula:

$$i = 1 - (|(s1 - s2)| + |(s2 - s1)|) / |(s1 \cup s2)|$$

The higher similarity, the higher is the possibility of getting benefit from combining the two partitions. Merging two partitions with a similarity above a certain support could enhance the quality of configuration.

```

for  $E$  do
  | a
end
choriginal Table  $T_k$  for  $E$  do
  | a
end
ch $p_i, p_j \subset T_k$  if benefit of merging  $p_i, p_j$   $> 0$  then Merge( $p_i, p_j$ );
Run Partition greedy selection algorithm;
else if benefit of (merging  $p_i, p_j$  and adding  $cp_{max}$ ) and new Partition  $\leq c_{max}$ 
 $> 0$  then
  | Merge( $p_i, p_j$ );
  | Run Partition greedy selection algorithm;
else
  | exit;
end

```

**Algorithm 8:** Partition merging algorithm

## 4.9 Partition Implementation

The operations related to the implementation are not counted as the operations that contribute towards the continuous evaluation of the present workload. This is to avoid the 'oscillations' caused by an accidental abortion of the vertical partitioning due to the current workload increasing by re-partitioning above  $v_{low}$  and therefore restarting the procedure when a workload goes below the threshold level again. The entire process of re-partitioning is also not recorded in a history of executions in order to avoid the distortions to form the past executions.

## 4.10 Comparison between Static and Dynamic partitioning

Compared with a static partitioning algorithm, dynamic partitioning does not recycle the original table. As a result, a vertical partitioning generated by the algorithm may not be as good as static partitioning. The static partitioning algorithms generate the partitions for the long term workloads. Then, the transparency of vertical partitioning is not required and application programmers have enough time to incorporate information about the partitions into the database applications. Under the constraints of dynamic partitioning, the vertical partitions must be transparent to applications because there is no time for re-implementation of a query and data manipulation operations. Currently, commercial database management system can enforce the transparency of partition by implementing it as a materialized view or an index. Because a materialized view or index relies on the original table, recycling of the original tables cannot be achieved based on current database management systems. In the next chapter, we introduce the ways to implement partitions. Also, we propose a solution to make recycling original tables possible which requires only minor modifications of commercial database management systems.

## 4.11 Summary

This chapter presents a dynamic vertical partitions algorithm for dynamic workload under storage constraints and implementation time limitations. The basic idea of the solution is to improve the database performance in high workload time by creating partitions to reduce the unnecessary read for queries. Since the creating partitions will increase the workload, we implement the partitions in low workload time.

Firstly, we introduced how the workload changes. Then we introduced the basic definitions for the dynamic algorithm. In section 4, we introduced a way to detect a moment which could be low workload time. When the potential low workload time has come, we start to find a moment in the past having the most similar situation as the current one based on the *Signature* of workload. Once the most similar workload has been located, we validate if the current time is a low workload time or not. If so, we use a greedy algorithm to find the configuration for the coming high workload time. Finally, we attempt to avoid affecting performance in a low workload time by scheduling the partitions creating.

---

The contributions of this chapter are as follows:

1. we show how workloads change in a periodic way, and use the signature of the workload to locate a moment in the past having a similar situation as current time.
2. we proposed a way to generate the partitions dynamically to fit the workload changes so that we can auto tune the query processing.

Because the configuration changes with the workload, the partitions should be completely transparent to database applications. How to implement partitions is critical to database auto tuning. We will be discuss this problem in the next chapter.

# Chapter 5

---

## Implementation of Dynamic Vertical Partitioning

In an ideal case, the changes of configuration of partitions should be completely transparent to the applications and require no intervention or hints. Therefore, the way of implementing a partition is critical for tuning database systems by vertical partitioning. At the moment only indices, and in the more advanced systems, materialized views are transparently considered by a query processor when generating the query execution plans. The other important property needed for the implementation of dynamic vertical partitioning is the ability of a query processor to dynamically replace the stored old query execution plans with new ones generated after the vertical partitions are built.

In this chapter, we will firstly discuss the issues of implementation of partitions, and then we present the ways of implementing a partition as an index and as a materialized view. To compare the two ways of implementing partitions, we provide not only some criteria for comparison between an index and a materialized view, but also establish cost models for the criteria.

### 5.1 Issues of implementing partitions

To achieve an auto tuning database system by vertical partitioning, partitions should be completely transparent to applications. Otherwise, changes of partitions will require modification of applications. As a result, dynamically optimizing a database by the use of vertical partitions cannot be achieved.

In current commercial database management system, the partition can be implemented as a relational table, a relational view, an index or a materialized view. If the partition is implemented as a relational table, it may cause a problem of optimal choice of partition for a query. For example, suppose we have table  $T(A, B, C, D)$ , partitions of  $T$ :  $P1(A,B)$ ,  $P2(A,B,D)$  where  $A$  is the primary key. Consider a query

```
SELECT A, B FROM T
```

The query of selection from T cannot be transformed to selection from P1 by query optimizer automatically.

If the partition is implemented as a materialized view, the query processor in the commercial database management system can detect the optimal materialized view for a query and be able to rewrite the query to access the optimal materialized view. If the partitions are implemented as indexes over the relational tables, a query processor is able to detect that horizontal traversal of an index is equivalent to a full scan of a partition. Therefore, implementing partitions either as a materialized view or index allows the changes of the partition as transparent to the applications.

In the next section, we show the ideas of index-based and materialized view-based implementation.

### 5.1.1 Materialized View-based Implementation

A materialized view is a preserved result obtained from the query processing in a database. Therefore, a query is executed much faster by accessing a materialized view than by accessing a normal view. The advanced query processors in the commercial database management systems are capable of rewriting a query such that a materialized view is used instead of a relational table when it improves the performance of the query processing. As a consequence, the mechanism is completely transparent to query processing and it allows for the dynamic implementation of vertical partitions.

Take table T, partitions P1, P2 and the same query in figure 5.1 as an example again. The partitions P1 and P2 are implemented as MV1, MV2 respectively. When a query is executed, the query optimizer will analyse whether it is faster to access the materialized view or the underlying tables where the data resides. If the optimizer accessing the materialized view is a better solution, the optimizer rewrites the query to use a materialized view.

However, the data in materialized views are not immediately updated together with the data in a master table. This causes a problem of preserving the consistency between a master table and a materialized view built as a projection of the master table. Basically, we have two ways to refresh a materialized view, Fast Refresh and Complete Refresh. A Fast Refresh can update the materialized view quicker than Complete Refresh but it cannot be used in some cases. To perform a Fast Refresh, requires a log on the change of the master table. However, some types of bulk load operation on a master table do not write log. For example, the bulk load operation includes some INSERT statements with an APPEND hint and some INSERT ... SELECT

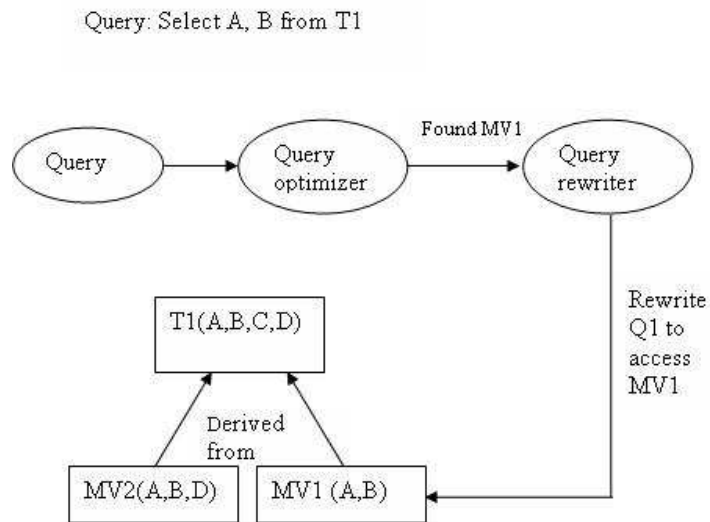


Figure 5.1: Example of Materialized View

\* FROM statements. This causes the consistency problem of the master table and the materialized view. Complete Refresh reloads data again. It takes a substantially longer amount of time to synchronise the materialized view and master table but this can guarantee the consistency between them. If we use Complete Refresh, it could aggravate the burden during high workload time.

### 5.1.2 Index based Implementation

The implementation of a vertical partition as a composite key index is possible when a query processor is able to detect and to apply the index-only processing of a query. In such a case, a query is processed by the horizontal traversal of a leaf level of B\*-Tree index without access to a relational table. At a logical level it is equivalent to a sequential scan of a vertical partition. As long as a partition is implemented as an index over a relational table, the query processor is able to invoke a horizontal traversal of the index automatically.

The B-tree includes the branch level and leaf level. A branch node contains pointers to leaf nodes or other branch nodes. The leaf level contains every indexed data value and a corresponding Rowid used to locate the actual row. In such a case, rewriting a query in the original application is not necessary. Figure 5.2 illustrates the data in a leaf level being used in a horizontal traversal of index.

The implementation of a partition as a composite key index requires additional space for recording index information beside spending some space on table columns. As a result, the storage costs of an index-based implementation are higher than the

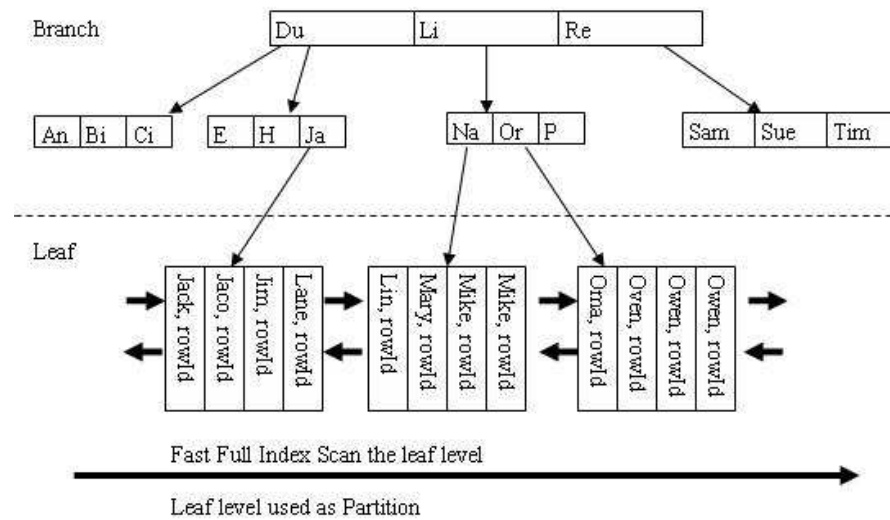


Figure 5.2: Horizontal traversal of index

costs of a materialized view-based implementation.

The storage cost model in the next section shows the difference between index and materialized view.

## 5.2 Comparison of Index-based and Materialized view-based

Either implementing a partition as an index or as a materialized view will be transparent to the application and the query processor will automatically find the optimal data source for the query. There are three main criteria for comparing implementation as an index or as a materialized view.

1. Time cost for maintaining partition consistency with the original table on which it relies.
2. Storage cost for building a partition by an index or by a materialized view.
3. Time cost for building a partition by an index or by a materialized view.

### 5.2.1 Consistency Control Comparison

Commercial database systems have two basic methods to maintain the consistency of a materialized view: Fast Refresh and Complete Refresh. Since Fast Refresh cannot apply changes which are caused by some types of bulk load operation on a master table,

we have to use Complete Refresh to guarantee the consistency for materialized view. If we use Complete Refresh, we have to rebuild the whole materialized view which may require a heavy read and write operation.

Compared with a materialized view, an index has no need to rebuild to keep it consistent with its master table. An index will be automatically updated simultaneously in a current commercial database system when the index key in the table has been changed. The time cost of maintaining an index includes two parts: traversing the index tree and updating the index tree.

Because the index will be updated immediately with the index key modification, it may lead to overhead update the index. In some cases, the data in the partitions may not necessarily be updated immediately with every single modification by a SQL statement. We may update the partition at a particular time such as every four hours. In such a case, implementing a partition as a materialized view take advantage of the index. However, if the modification of data is sensitive to the coming queries, the change of partition and table must be synchronized at the same time. In such a case, an index will be a better choice than a materialized view.

### 5.2.2 Storage Cost Comparison

#### Storage Cost by Materialized View

Suppose the size of block is  $b$ , the length of partition columns is  $c$  and extra space to record materialized view information. And there are  $r$  rows in table  $T$ . Therefore, we can deduce that:

The number of fully packed blocks to store the partition is:

$$n = r / (b / (c + e))$$

The storage cost for a materialized view is approximately equal to the total rows of a table multiplied by the row size of a partition. To enforce the integrity of a materialized view, we consider Complete Refresh since Fast Refresh cannot apply changes for some bulk loads. Therefore, the cost of update, insertion, and deletion is equal to twice of partition size.

#### Storage Cost by Index

All the partitions are implemented as non-clustered B-tree index. Here is the formula to evaluate the storage cost.



The index holds the key value and then the address of the row in the table with this value. Each row in the table must have an entry in the leaf blocks of the index. The branch blocks are used as an index to the leaf nodes. So each block for the leaf nodes needs to be addressed by the next level up of the branch nodes.

Suppose the size of block is  $b$ , the length of index columns is  $c$  and extra space  $e$  to record an index pointer. And there are  $r$  rows in table  $T$ . Therefore, we can deduce that:

1. The number of rows can be stored in a fully packed block

$$n = b/(c + e)$$

2. The total blocks for leaf level:

$$l = r/n$$

3. The total blocks for branch(fanout):

$$f = l/n$$

So, the formula to estimate an index size is:

$$IndexSize = l + f$$

### 5.2.3 Comparison by Time Cost

#### Time cost by Materialized View

Suppose the size of block is  $b$ , the length of partition columns is  $c$  and extra space to record materialized view information. There are  $r$  rows in table  $T$ , therefore, we can deduce that:

$$n = r/(b/(c + e))$$

The time cost for a materialized view is the total read and write blocks of building it.

#### Time cost by index

The time cost of building an index is the total of read and write blocks for inserting. Suppose there are  $k$  index keys on the index of table  $T$ . When we create the index for

table T, we have to take some time to insert  $k$  rows with a new value of index key, meanwhile we have to insert  $(r-k)$  rows with an existing value of index key.

The time of inserting a row with a new value of index key is:

$$t_1 = 2 * (1 + \log_f k)$$

The time of inserting a row with an existing value of index key is:

$$t_2 = 1 + \log_f k$$

Therefore, suppose every index key has the same number of rows which is  $r/k$ . The cost of building an index goes up with more and more index key being inserted in the tree. For each index keys, we spend one  $t_1$  and  $(r/k - 1)*t_2$ . Then the total time spending on creating an index is:

$$T = (2*(1+\log_f 1)) + (r/k-1)*(1+\log_f 1) + \dots + (2*(1+\log_f k)) + (r/k-1)*(1+\log_f k)$$

$$T = (n/k + 1) * (\log_f 1 + \dots + \log_f k) + n$$

## 5.3 Open problems

Because both the materialized view and index rely on the master table, the tables in the original schema cannot be recycled for further vertical partitioning. To make the change of partitions transparent to all applications and to recycle the original tables, we proposed a solution which requires commercial database system to process the following two abilities:

1. The database system should have query rewrite ability for a view;
2. The database system should have a view update ability.

Firstly, we define a view to present the original table. The view is joining of all partitioned tables which belong to the original table. When a query accesses the view, the database system should rewrite the query to access the optimal partition. Since current commercial database systems can rewrite the query to use the optimal materialized view, the commercial database system in the future should have the same mechanism to rewrite the query using the optimal partition.

Once the original table is replaced by a view, the database system should be able to update the partitions when the view is required to be updated. INSTEAD OF Trigger

in commercial database systems can be used to synchronise the partitions of a view. For insertion of a view, we simply insert the new value into respective columns in the partitions. For update and deletion of a view, firstly we have to find out the primary keys of the columns which are changed. To find the primary keys, we may have to join some or all of the partitions of the view to verify the conditions to update or delete. Based on the primary keys we find, we can update or delete the relevant record in the partitions.

## 5.4 Experiments

All experiments have been conducted with an off-the-shelf, commercially available database server Oracle 10g, release 1 running on a single processor 2-GHz Intel CPU box with 512 MB of main memory and 40-GB hard drive. A sample database implemented TPC-R[3] benchmark database with data generated according to the benchmark specifications.

The TPC-R database comprises 8 tables. We experimented with the workloads consisting of 20 queries accessing two of the largest relational tables in a sample database. The size of the largest table is about 2 Gbytes and its schema consists of 16 attributes. The size of the second largest table is about 650MB and its schema consists of 9 attributes. Both tables have the non-clustered B-tree indexes automatically constructed on the primary keys.

### 5.4.1 Query Rewrite of Materialized View

In this experiment, figure 5.3 shows how the query is rewritten to use a materialized view which is better than using a table. Firstly, we need to enable a query rewrite in Oracle. Then, we create a materialized view `MV_Nation(N_nationkey, N_nationname)` for table `Nation`. The figure shows how Oracle rewrites the query to access `MV_Nation` instead of `Nation`.

### 5.4.2 Fast Full Index Scan

This experiment shows how a commercial database system uses a horizontal traversal of an index. In oracle, the horizontal traversal of an index is called Fast Full Index Scan. Fast Full Index Scan will replace a full scan table for a query automatically when the query optimizer detects it will take advantage over a full scan table. In this

```

SQL> create materialized view mv_nation
2  enable query rewrite as
3  select n_nationkey, n_name from nation;

Materialized view created.

SQL> ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE;

Session altered.

SQL> set autotrace traceonly explain
SQL> select n_name from nation;

Execution Plan
-----
 0  SELECT STATEMENT Optimizer=ALL_ROWS (Cost=3 Card=25 Bytes=67
    5)
 1  0  MAT_VIEW REWRITE ACCESS (FULL) OF 'MV_NATION'
    (MAT_VIEW REWRITE) (Cost=3 Card=25 Bytes=675)

```

Figure 5.3: Query Rewrite of Materialized View

```

SQL> create index idx_nation on nation(n_name) compute statistics;

Index created.

SQL> select /*+ index_ffs(nation idx_nation) */ n_name from nation;

Execution Plan
-----
 0  SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=25
    Bytes=625)
 1  0  INDEX (FAST FULL SCAN) OF 'IDX_NATION' (INDEX) (Cost=2
    Card=25 Bytes=625)

```

Figure 5.4: Fast Full Index Scan

experiment, we build an index for nation called `idx_nation`. As figure 5.4 shows, the selection of nation will automatically use Fast Full Index Scan.

### 5.4.3 Comparison of Materialized View and Index

In this experiment, we compare the time and storage of building a Materialized View and an Index. As figure 5.5 shows, the cost of time and storage of building a materialized view is less than it is for an index.

#### 5.4.4 View Update

Figure 5.6 shows how a view can be updated in a current commercial database management system. There are two tables `Nat_name(N_NATIONKEY, N_NAME)`, `Nat_Others(N_NATIONKEY, N_REGIONKEY, N_COMMENT)` and view of `v_Nation` which joins tables `Nat_name`

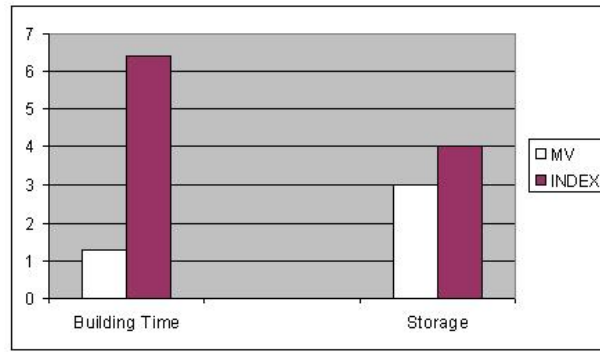


Figure 5.5: Comparison of MV and Index

and Nat\_Others by N\_NATIONKEY. In our proposed solution, v\_Nation represents the original table while Nat\_name and Nat\_Others represent the vertical partitions. If v\_Nation is required to be updated, we should have INSTEAD OF as a trigger to update its underlying tables Nat\_name and Nat\_Others. For each row to be updated, we record its primary key which is the link to the underlying tables. Then we update the corresponding values of the underlying table.

## 5.5 Summary

This chapter has shown the possible solutions of implementing partitions for an auto-tuning database. Currently, to implement partitions as tables leads to query rewriting which requires the applications to adapt to changes of configuration. By comparison, to implement partitions as indices and materialized views make the changes of configuration transparent to an application. The query processors in commercial database systems will automatically pick up the optimal data destination for the query. We proposed cost models for estimating implementation of partitions as materialized views and indices. To reduce the data redundancy, we proposed a solution to make recycling of the original table possible. If the database system can rewrite the query for a view, we can decompose an original table into partitions and create a view of the original table. We may leverage the current rewrite mechanism of a materialized view to achieve the ability of rewriting a query for a view. The chapter ends with some experiments which show:

1. A partition can be transparent to applications if it is implemented as an index or implemented as a materialized view.
2. The comparisons of costs to implementing a partition as a materialized view or

```

SQL> CREATE or REPLACE TRIGGER tr_upd instead of update on V_NATION
    for each row
    declare
    V_NATIONKEY NUMBER;
    Begin
        select N.N_NATIONKEY into V_NATIONKEY
        from NAT_NAME N, NAT_OTHERS O
        where :old.N_name = N.N_name and :old.N_comment = O.N_comment
        AND :OLD.N_REGIONKEY = O.N_REGIONKEY;

        update NAT_NAME set N_name = :new.N_name
        where N_NATIONKEY = v_NATIONKEY;

        update NAT_OTHERS set N_REGIONKEY = :new.N_REGIONKEY,
            N_COMMENT = :new.N_comment
        where N_NATIONKEY = V_NATIONKEY;
    end tr_upd;

SQL> select * from v_nation where n_name = 'PERU';
N_NATIONKEY N_NAME                N_REGIONKEY
-----
N_COMMENT
-----
          17 PERU                      1
final, final accounts sleep slyly across the requests.

SQL> select * from nat_name where n_name = 'PERU';
N_NATIONKEY  N_NAME
          17    PERU
SQL> select * from nat_others where n_nationkey = 17;
N_NATIONKEY N_REGIONKEY  N_COMMENT
-----
          17          1    final, final accounts sleep slyly across the requests.

SQL> update v_nation set n_name = 'PERU 1', n_comment = 'Good Country',
n_regionkey=2 where n_name = 'PERU' and n_regionkey = 1;
SQL> select * from v_nation where n_name = 'PERU';
no rows selected

SQL> select * from v_nation where n_name = 'PERU 1';
N_NATIONKEY  N_NAME                N_REGIONKEY  N_COMMENT
          17    PERU 1                      2    Good Country

```

Figure 5.6: View Update

an index.

3. A view can be updated so that the solution we proposed can be achieved if a current commercial database management system makes a minor modification on the rewrite ability for a view.

# Chapter 6

---

## Conclusions and Future work

### 6.1 Conclusions

This thesis began with an overview of the performance tuning problems and the demand for an auto-tuning database, along with a brief introduction of autonomic database system features. Then we introduced major approaches to optimise query performance. Since it is impossible to have a one-fits-all design that satisfies sometimes contradictory requirements of complex applications, an adaptive reorganising physical database structure allows for better utilisation of hardware resources. Our work in this thesis has only focussed on dynamic vertical partitioning which denotes the automated projections of relational tables in order to reduce the time for exhaustive table scans.

By comparing partitioning techniques, we have acknowledged that vertical partitioning takes advantage over horizontal partitioning because vertical partitioning is much easier when matching predicates between queries and partitions than horizontal partitioning. We reviewed the past work in vertical partitioning and to our best knowledge found there were no attempts to investigate the applications of dynamic vertical partitioning.

In chapter 3, we propose a algorithm to choose the optimal partitions for a given long term workload. We presented a detailed cost model for the precise estimating at the cost of insertion, deletion and update operations. However, the vertical partitioning algorithm cannot make an automatic repartition of a database in a response to the changing database loads.

In chapter 4, we proposed a solution, which is based on an assumption that the workload on a database server is repetitive in nature. This means that it is possible to anticipate during the periods of low workload what database applications will be executed in the future and to create the vertical partitions that will have a positive impact on future performance. The thesis describes the algorithm that finds the expected



workload and how it decided which vertical partitions should be created.

Finally, we investigated two implementation techniques: one based on materialized views and the other one based on indexing. Both of these techniques strongly rely on the master table. As a result, the original table cannot be recycled for further partitioning. We proposed to keep the original table as a view and synchronise the partitions by a view updating. However, the solution requires that the current commercial databases have the ability to rewrite a query for view.

## 6.2 Future work

In this thesis, dynamic vertical partitioning is a first step towards setting up a framework to optimise database performance by the dynamic modification of physical structure. Other physical database structure tuning methodologies such as reorganization of index, horizontal partitioning and clusters are expected to be adapted to this model. The thesis proposed implementing an original table as a view which is a joint of vertical partitioning. It would be exciting to come up with a commercial database management system processing the query rewriting ability, which can find an optimal vertical partitioning for the query.

In addition, the metric used in the cost model analysed in Chapter 3, 4 may be transferred to DBTime which has been proposed recently as a common metrics of performance impact. By using such a common metrics, the performance improvement in different components over the whole system can be gauged. With only limited resources, we can allocate those resources in the best way to optimise the database system.

Another interesting issue is the combination of other workload monitoring approaches to improve our work in this thesis. The thesis focuses only on workloads on a database server which are repetitive in nature. However, there are always some accidents to affect the workload behaviours. The monitoring and diagnosing of workloads is still an open question. In conclusion, we summarize possible future work as follows:

1. Integrate other physical database structure tuning methodologies such as index, horizontal partitioning and clusters;
2. Implement the rewrite ability of view for vertical partitioning in commercial database management systems;

3. Combine the metrics DBTime in our cost model in order to identify the performance improvement in different system components;
4. Cooperate with other workload monitoring and diagnostic approaches.

The further research on the above open problem will improve the quality of optimization by vertical partitioning, ensuring predictable performance and eliminating the need for manual tuning.

# Bibliography

---

- [1] Autopart: Automating schema design for large scientific databases using data partitioning. In *SSDBM '04: Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*, page 383, Washington, DC, USA, 2004. IEEE Computer Society.
- [2] Db2 design advisor: Integrated automatic physical database design. In *VLDB'2004: Proceedings of the 30th international conference on Very large data bases*, pages 1087–1097, 2006.
- [3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Hippocratic databases. In *28th International Conference on Very Large Databases (VLDB), Hong Kong*, 2002.
- [4] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Materialized view and index selection tool for microsoft sql server 2000. In *SIGMOD Conference*, page 608, 2001.
- [5] Sanjay Agrawal, Eric Chu, and Vivek R. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD Conference*, pages 683–694, 2006.
- [6] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370, New York, NY, USA, 2004. ACM Press.
- [7] A. Brown and D. Patterson. To err is human. In *Proceedings of the First Workshop on Evaluating and Architecting System dependability (EASY '01), July 2001.*, 2001.
- [8] Kurt P. Brown, Manish Mehta, Michael J. Carey, and Miron Livny. Towards Automated Performance Tuning For Complex Workloads. In *Proceedings of the*

- Twentieth International Conference on Very Large Databases*, pages 72–84, Santiago, Chile, 1994.
- [9] Silvana Castano, Maria Grazia Fugini, Giancarlo Martella, and Pierangela Samarati. *Database Security*. Addison-Wesley & ACM Press, 1995.
- [10] Surajit Chaudhuri, Arnd Christian König, and Vivek Narasayya. SQLCM: A continuous monitoring framework for relational database engines. *ICDE*, 00:473, 2004.
- [11] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Random sampling for histogram construction: How much is enough? In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 436–447. ACM Press, 1998.
- [12] Surajit Chaudhuri and Vivek Narasayya. Automating statistics management for query optimizers. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):7–20, 2001.
- [13] Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB*, pages 1–10, 2000.
- [14] W. W. Chu and I. T. Ieong. A transaction-based approach to vertical partitioning for relational database systems. *IEEE Transactions on Software Engineering*, 19(8):804–812, 1993.
- [15] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. Automatic performance diagnosis and tuning in oracle. In *CIDR*, pages 84–94, 2005.
- [16] Said Elnaffar, Wendy Powley, Darcy G. Benoit, and T. Patrick Martin. Today’s dbmss: How autonomic are they? In *DEXA Workshops*, pages 651–655, 2003.
- [17] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM System Journal*, 42(1):5–18, 2003.
- [18] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, and Yun Wang. Query optimization in the ibm db2 family. *IEEE Data Engineering Bulletin*, 16(4):4–18, 1993.

- 
- [19] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast incremental maintenance of approximate histograms. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *Proceedings 23rd International Conference Very Large Data Bases, VLDB*, pages 466–475. Morgan Kaufmann, 25–27 1997.
  - [20] Jim Gray, Alex S. Szalay, Ani R. Thakar, Peter Z. Kunszt, Christopher Stoughton, Don Slutz, and Jan vandenBerg. Data mining the sdss skyserver database, 2002.
  - [21] Sylvain Guinepain and Le Gruenwald. Research issues in automatic database clustering. *SIGMOD Record*, 34(1):33–38, 2005.
  - [22] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, pages 98–112, 1997.
  - [23] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize under a maintenance cost constraint. *Lecture Notes in Computer Science*, 1540:453–470, 1999.
  - [24] Michael Hammer and Bahram Niamir. A heuristic approach to attribute partitioning. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 93–101, New York, NY, USA, 1979. ACM Press.
  - [25] Joseph L. Hellerstein. Automated tuning systems: Beyond decision support. In *International CMG Conference*, pages 263–270, 1997.
  - [26] Joseph L. Hellerstein, David Hart, and Po Yue. Automated drill down: An approach to automated problem isolation for performance management. In *International CMG Conference*, pages 376–384, 1999.
  - [27] Jeffrey A. Hoffer and Dennis G. Severance. The use of cluster analysis in physical data base design. In Douglas S. Kerr, editor, *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*, pages 69–86. ACM, 1975.
  - [28] <http://www.tpc.org>.
  - [29] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD '98: Proceedings of the 1998 ACM*

- SIGMOD international conference on Management of data*, pages 106–117, New York, NY, USA, 1998. ACM Press.
- [30] Zhenjie Liu and Janusz R. Getta. Optimization of query processing through constrained vertical partitioning of relational tables. In *DBA'06: Proceedings of the 24th IASTED international conference on Database and applications*, pages 221–227, Anaheim, CA, USA, 2006. ACTA Press.
- [31] V. Markl, G. M. Lohman, and V. Raman. Leo: An autonomic query optimizer for db2. *IBM System Journal*, 42(1):98–106, 2003.
- [32] McCormick, WT Schweitzer, J Paul, and TW White. Problem decomposition and data reorganization by a clustering technique. In *Operations Research*, pages 993–1009, 1972.
- [33] Dushyanth Narayanan, Eno Thereska, and Anastassia Ailamaki. Continuous resource monitoring for self-predicting dbms. In *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 239–248, Washington, DC, USA, 2005. IEEE Computer Society.
- [34] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Transaction Database System*, 9(4):680–710, 1984.
- [35] Shamkant B. Navathe and Minyoung Ra. Vertical partitioning for database design: A graphical algorithm. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pages 440–450. ACM Press, 1989.
- [36] Vincent Ng, Dik Man Law, Narasimhaiah Gorla, and Chi Kong Chan. Applying genetic algorithms in database partitioning. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 544–549, New York, NY, USA, 2003. ACM Press.
- [37] Daniel Paul, Sudhakar Yalamanchili, Karsten Schwan, and Rakesh Jha. Decision models for adaptive resource management in multiprocessor systems.

- 
- [38] Viswanath Poosala, Peter J. Haas, Yannis E. Ioannidis, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 294–305, New York, NY, USA, 1996. ACM Press.
  - [39] Ron Rymon. Goal-directed diagnosis-diagnostic reasoning in exploratory-corrective domains. In *IJCAI*, pages 1488–1493, 1993.
  - [40] Kai-Uwe Sattler, Ingolf Geist, and Eike Schallehn. Quiet: Continuous query-driven index tuning. In *VLDB*, pages 1129–1132, 2003.
  - [41] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 488–499, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.
  - [42] Jin Hyun Son and Myoung Ho Kim. An adaptable vertical partitioning method in distributed systems. *Journal of Systems and Software*, 73(3):551–561, 2004.
  - [43] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *VLDB'2006: Proceedings of the 32nd international conference on Very large data bases*, pages 1081–1092. VLDB Endowment, 2006.
  - [44] Alexander S. Szalay, Jim Gray, Ani Thakar, Peter Z. Kunszt, Tanu Malik, Jordan Raddick, Christopher Stoughton, and Jan vandenBerg. The sdss skyserver: public access to the sloan digital sky server data. In *SIGMOD Conference*, pages 570–581, 2002.
  - [45] Gerhard Weikum, Axel Mönkeberg, Christof Hasse, and Peter Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB*, pages 20–31, 2002.