UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

DYNAMIC DATABASE VERTICAL PARTITIONING ON SINGLE COMPUTERS AND CLUSTER COMPUTERS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

LIANGZHE LI
Norman, Oklahoma
2013

DYNAMIC DATABASE VERTICAL PARTITIONING ON SINGLE COMPUTERS AND CLUSTER COMPUTERS

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY

Acknowledgements

I have been working on this research for two years and it is impossible to complete the task on my own. Many people provided their support, guidance and encouragement to me.

First I want to thank all the members of my thesis committee: Dr. Le Gruenwald, Dr. Qi Cheng and Dr. Sudarshan Dhall. Without their guidance I could not successfully finish my research and had my results published. I also want to thank the original member of my team, Miguel Pardo Marin, for helping me on the program coding task when I just joined the project. A special gratitude I owed to Dr. Gruenwald is that she accepted me, a person without any experience in research field, into the OU Database Group and spent a lot of time advising me on my work. I will always grateful for her assistance.

However, I wish to give my greatest gratitude to my family, my parents and my wife. I was supposed to stay with you every day but you have seen me less for the past two years because of my busy work. Finally, and the most important, I thank my daughter, Sunny Shi Li, though you just came to this world and had no chance of helping me on my task you brought so many happiness to me. I hope you could know that I love you so much.

Other Database Group members provided support on the way during the two years. They are Jaemyeong Jeon and Shiblee MD. Sadik. I offer them my thanks. A lot of friends in Computer Science department offered their help, too. I cannot list all names here, I thank you all.

Table of Contents

List of Tables

List of Figures

Abstract

Today, databases and cluster computers are widely used in many applications. With the volume of data getting bigger and bigger and the velocity of data getting faster and faster, it is important to develop techniques that can improve query response time to meet applications' needs. Database vertical partitioning that splits a database table into smaller tables containing fewer attributes in order to reduce disk I/Os is one of those techniques. While many algorithms have been developed for database vertical partitioning, none of them is designed to partition the database stored in cluster computers dynamically, i.e., without human interference and without fixed query workloads. To fill this gap, this thesis develops the dynamic algorithm, SMOPD-C, that can vertically partition a database, determine when a database re-partitioning is needed, and re-partition the database accordingly.

SMOPD-C makes uses of two novel algorithms, Filtered AutoClust and SMOPD, developed in this thesis, Filtered AutoClust is designed for statically clustering the database attributes using a fixed query workload on both single and cluster computers. It uses data mining to mine all possible groups (partitions) of attributes that are frequently accessed together in queries, implements the partitions on computing nodes, and routes each incoming query to the computing node containing the partitions that give the best estimated cost for the query execution. The second algorithm, SMOPD, is designed for dynamically clustering the database attributes on a single computer without using any fixed query workload. It uses the statistic information from the system views to determine physical read mainly queries from those queries that have been processed since the last partitioning to build a new query set. Then it reevaluates the average estimated query cost for each database table using the new query set to determine whether or not re-partitioning is needed for a database table. If a database table needs re-partitioning, then Filtered AutoClust is called to re-partition this database table. SMOPD-C then uses both SMOPD and Filtered AutoClust to do vertical partitioning dynamically for the database stored in a cluster computer. In SMOPD-C, each computing node runs SMOPD to monitor the performance of queries accessing its database tables. When SMOPD-C determines that for a particular database table, the query performance is decreasing on more than half of the computing nodes, it invokes Filtered AutoClust to re-partition this database table.

Comprehensive experiments were conducted in order to study the performance of Filtered AutoClust, SMOPD and SMOPD-C using the TPC-H benchmark as well as a synthetic database and synthetic queries on the cluster computer at OSCER (OU Supercomputing Center for Education and Research). The experiment results show that Filtered AutoClust gives better estimated query cost than existing static vertical partitioning algorithms; and SMOPD as well as SMOPD-C are capable of performing database re-partitioning dynamically with high accuracy to provide better estimated query cost than the current partitioning configuration.

Chapter 1: Introduction

1.1 Objectives and Motivation

Nowadays almost all activities in many applications, such as commerce, manufacturing and education, are relevant to database. The structure of data is getting more and more complex; the size of data is getting bigger and bigger; and the velocity of data is getting faster and faster. Due to those reasons database systems need to keep improving in order to satisfy today's needs. Since database systems are becoming more and more complicated, it is impossible to ask novice users and it is also expensive to hire experienced Database Administrators (DBAs) to manage all the settings in a database by themselves. This leads to the important research topic of self-managing database management systems.

Recently researchers have been paying more and more attention to the developments of self-managing database algorithms which include self-managing database indexing ([2], [3]), self-managing database caching [4], self-managing database partitioning [5], self-tuning database parameters [6], etc. In this thesis we focus on database partitioning.

As we know a database application's performance highly depends on how quickly data could be retrieved from the database. When the database size is small, the time spent on reading/writing data from/to disk and operating (selecting, merging, filtering, etc.) data is usually small, and thus its impacts on query response time may not be very critical. However, today with database size getting bigger and bigger like those in bio-science, finance and medicine, if the database is not organized properly, such time overhead could yield unacceptable query response time.

Vertical partitioning and horizontal partitioning are two major techniques which can considerably improve query response time when physical database design is performed [7]. Today, most database systems support horizontal partitioning [8]. Three common horizontal partitioning approaches that are used by most database developers are range partitioning, list partitioning and hash partitioning [7]; but it is rare to find a database system that has a sophisticated algorithm, especially a dynamic algorithm, to support vertical partitioning without human interference for cluster computers. So dynamic vertical database partitioning is the topic addressed in this thesis.

The primary objective of this research is to develop an algorithm which can perform vertical partitioning on database tables dynamically on both single computers and cluster computers in order to reduce the cost of I/Os. To accomplish this, four major tasks are performed in this thesis. The first task is to design a new algorithm which should be able to vertically partition the database tables in such a way that it would have better performance than existing vertical partitioning algorithms and has a good execution time as well. The second task is to provide the new algorithm a dynamic ability to automatically detect the database performance's trend and re-partition the database tables when performance is degrading. The third task is to extend the algorithm so that it can work on both single computers and cluster computers. Finally, the fourth task is to implement the designed algorithms and conduct experiments to evaluate their performance using the TPC-H benchmark [9] as well as synthetic databases and queries.

1.2 Background of Database Partitioning for Single Computers

As query response time is composed of I/O time and CPU time, reduction on either of them can lead to an improvement of the database application's performance. Without partitioning the database, when the database system is trying to fetch data from hard disks, the whole database table, rather than those attributes or tuples just queried, will be read. This means most of time, the system cannot avoid reading unnecessary data from hard disks when processing queries. However, we know that if more data are read from disks, more time the system will take. If we cannot organize data well on hard disks, then the performance of the database will be seriously impacted. That is why database partitioning techniques come into existence.

Database partitioning techniques can be classified into two major categories: horizontal partitioning and vertical partitioning [7]. In horizontal partitioning, tuples are saved in the same disk block according to some specific criteria. If users want to fetch all tuples based on those criteria, the database system can easily locate those tuples on disks. Current horizontal partitioning paradigms mainly include hash-partitioning and range-partitioning [13]; but a problem for both of the two paradigms is that neither can avoid unnecessary readings of attributes that are not queried by queries. Vertical partitioning can help solve this problem.

In vertical partitioning, attributes are clustered together based on how often they are accessed together in a query set. If we reorganize database tables in such a way that each table is partitioned vertically into sub-tables and the database system, when executing the query, database will access only the relevant sub-table that contains the attributes in the query, then fewer pages from disks will be accessed to process the query [14], which reduces I/O time, and thus can lead to a better query response time.

Today there are many ways to categorize vertical partitioning techniques, such as fully automatic and semi-automatic, optimizer-independent and optimizer-integrated, etc. For a fully automatic partitioning algorithm, it does not need human interference for feedback when the algorithm is running, while a semi-automatic algorithm does. For an optimizer-independent algorithm, the query optimizer of a database system is not involved in the algorithm, while for an optimizer-integrated algorithm, the query optimizer is used to evaluate the candidate solutions when the algorithm is running. Such an algorithm largely uses the query optimizer as a black-box to perform the optimization. The query optimizer performs the *what-if* calls [15] to select a better execution plan which can be used to find out the best possible partitioning solution.

In our thesis we categorize vertical partitioning algorithms into two categories: dynamic and static. A dynamic partitioning algorithm is an online algorithm that keeps running all the time and has the ability to track the database performance in order to perform re-partitioning at the right time. A static partitioning algorithm is unable to do re-partitioning automatically. It uses a fixed workload to decide the partitioning results, which means that the future workload must be very similar to the one used to generate the current partitioning results; otherwise the partitioning results generated by this workload might not perform well in the future. Using static partitioning algorithms, if people wish to re-partition the database table, they need to monitor the database performance and make the re-partitioning decision by themselves. However, in many cases, the workload keeps changing, which would require the DBA to spend much time on monitoring the database performance. This is a waste of labor resources. So it is necessary to have a dynamic partitioning algorithm – an algorithm that is able to monitor the database performance, determine when a re-partitioning action is needed, and perform re-partitioning accordingly. A dynamic algorithm should be able to perform all these three actions automatically by itself without help from the DBA.

In our thesis, we first propose an improved version, called Filtered AutoClust, of the existing optimizer-integrated static vertical partitioning algorithm, AutoClust, and then we make it dynamic for a single computer, so we have the new algorithm, SMOPD. We then develop an extended version of SMOPD for cluster computers, which we call SMOPD-C. Three of the algorithms are described in details in Chapters 3 and 4.

1.3 Background of Database Partitioning for Cluster Computers

In recent years, cluster computers have been widely adopted by many organizations to handle their computation and data processing operations. A cluster computer consists of a collection of interconnected stand-alone computers working together as a single, integrated computing resource [16]. A network composed of many cluster computers can provide excellent computing performance with low cost. Due to this reason more and more parallel applications are running on cluster computers, though varied Ethernet architectures may impact the performance [17] since the communication problem can limit the performance over standard LANs.

An important usage of cluster computers is in the database field. Cluster computers can be used to deploy databases to improve performance. In recent years distributed databases are widely used and usually implemented on computer clusters to form a massively parallel processing system (MPP) [18]. In a MPP system, each node is an independent computer in itself which at least should contain one processor, its own memory and the connection to the other computers. How to optimize the database design on a cluster system becomes extremely important, and database partitioning on a cluster system is one of those important research topics in the physical database design area.

Generally a database table can be partitioned in different ways. Each way may be optimal for some specific query types but not for all query types. When we implement a database system on a cluster computer, we can deploy different partitioning solutions on different computing nodes so that a query can be processed by the node with the optimal partitioning solution whenever possible. It is obvious that this way is more efficient than the way of implementing the same partitioning solution on each node; but unfortunately, for all current existing vertical partitioning algorithms which we have reviewed (Bond Energy [21], Navathe'sVertical Partitioning [14], Optimal Binary Vertical Partitioning [22], Graph Search Vertical Partitioning [23], AutoStore [5] and Amossen Algorithms [27]), they can provide only one partitioning solution, i.e., if users want to deploy the algorithm on a cluster computer, they have to implement the same partitioning solution on every node. As we have discussed, as the query set may change over time, the best partitioning solution may not be the best forever for all queries; so when we work on a cluster computer, it is better to generate multiple partitioning solutions and find which solutions are the best for which nodes and implement them accordingly in order to improve the overall query costs. This is the goal that we have achieved with our vertical partitioning algorithms, Filtered AutoClust, SMOPD and SMOPD-C, which we present in Chapters 3 and 4.

1.4 Issues Concerning Database Vertical Partitioning Algorithms

1.4.1. Static vs. dynamic

In a static vertical database partitioning algorithm, the input query workload is fixed. The algorithm uses this fixed workload to generate the result partitioning solution for each database table. However, in many real applications, the input workload often changes over time, which means there might not be a fixed pattern for the workload. A dynamic vertical database partitioning algorithm should know how to automatically collect a query set and use this query set to judge the performance trend to determine whether or not re-partitioning is needed. Most of the existing database vertical partitioning algorithms ([11], [14], [21], [22], [23], [24], [27]) assume that the future workload has the very similar pattern with the workload used for partitioning. The input of such algorithms is a fixed workload and the resulting partitioning solution of each database table totally depends on the pattern of the input workload. If the future workload's pattern is very similar to the one used as the input by the algorithm, then the partitioning solution may have good performance; but if the future workload's pattern changes a lot, then the partitioning solution may have very bad performance and even worse than the case when no partitioning is performed. We know in reality, incoming queries of many database applications usually change over time. If we use a static vertical partitioning algorithm, the performance may be good at the beginning; but along with the changes of the workload, the partitioning solution may lose accuracy without people's awareness, and finally cause bad query response time. So a dynamic vertical partitioning algorithm is more suitable for today's database applications.

1.4.2. Single partitioning solution vs. multiple partitioning solutions

When people are trying to partition a database table on a single computer, it is obvious that the final partitioning solution applied on a database table is just one. There is no reason to keep multiple partitioning solutions for the same database table on a single computer since it means redundant data. So whether the algorithm can generate only one partitioning solution or multiple partitioning solutions will not impact the final partitioning result. If the algorithm generates multiple partitioning solutions, we just need to select the best one. Today a single computer usually cannot satisfy business or research requirements. People rely on cluster computers more and more. Cluster computers can provide more computing power to solve very complex problems. Database vertical partitioning on cluster computers becomes more important than database vertical partitioning on a single computer. As we know a cluster computer is made up of many single computing nodes. Each computing node may process different queries. The same database table may have different optimal partitioning solutions for different queries. So different partitioning solutions of the same database table should be implemented on different computing nodes in order to have each query processed by the optimal solution. To the best of our knowledge, no existing vertical database partitioning algorithm provides multiple partitioning solutions. This is because no existing vertical database partitioning algorithm is designed for cluster computers. It is important to develop such an algorithm.

1.5 Contribution

Database vertical partitioning is an important method for database performance tuning; unfortunately most existing database vertical partitioning algorithms can only partition database tables statically. Very few techniques can partition database tables dynamically. For those techniques that provide dynamic solutions, they are designed for a single computer, but not for a cluster computer. We propose three algorithms in this thesis. First we propose Filtered AutoClust which improves the original static vertical partitioning algorithm, AutoClust [19]. Similar to AutoClust, it can generate different partitioning solutions for the same database table, determine which solution is suitable for which node, and implement it accordingly; but it has better running time. Then we develop SMOPD and SMOPD-C. These two algorithms are the dynamic versions of Filtered AutoClust for a single computer and a cluster computer, respectively. To the best of our knowledge, SMOPD-C is the first dynamic vertical partitioning algorithm that is designed for cluster computers.

1.6 Organization

This thesis is divided into six chapters. The following paragraphs provide an overview of each chapter.

Chapter 2 reviews the literature in two areas. First, the literature associated with the general database performance tuning is briefly reviewed as it is related but is not the focus of this thesis; and second, the literature associated with database partitioning is comprehensively reviewed. .

When the existing basic vertical partitioning algorithm, AutoClust, was first proposed in [19], the authors provided the algorithm for a single computer. It contains the general ideas for vertical partitioning on cluster computers but not formalizes the ideas. In Chapter 3 we formalize the ideas and discuss an issue when the AutoClust is dealing with a large number of closed item sets (an item set is called closed if it has no superset having the same support which is the fraction of queries in a data set where the item set appears as a subset [10]). We describe how to improve it and introduce our algorithm called Filtered AutoClust, which is able to filter out unnecessary closed item sets and run faster. This chapter is divided into two subsections. The first section describes how the AutoClust algorithm is extended to cluster computers. The second section describes how the Filtered AutoClust algorithm works and how cluster computers can benefit the algorithm greatly.

In Chapter 4, we introduce SMOPD and SMOPD-C, the dynamic versions of Filtered AutoClust for single and cluster computers, respectively. We describe the function of each of the three components of SMOPD: query collector, statistics analyzer and database cluster, showing how the three components communicate with each other so that the partitioning work could be automatic without human interference during the running process. Then we present how SMOPD-C makes use of Filtered AutoClust and SMOPD to re-partition database tables on cluster computers. This chapter is divided into two subsections, one describing SMOPD and one describing SMOPD-C.

Once we have described all algorithm designs, in Chapter 5, we present experimental performance evaluations of all the proposed algorithms using the TPC-H benchmark [9] and a synthetic database and synthetic queries. The TPC-H benchmark is a famous benchmark used by many database researchers to evaluate the performance of their algorithms. It includes a database containing 8 different size database tables and a query set containing 22 different query types. Finally, Chapter 6 contains the conclusions drawn from this research and suggestions for future work.

Chapter 2: Literature Review

2.1 Introduction

As discussed in the previous chapter, the main purpose of this research is to develop a dynamic vertical database partitioning algorithm such that this algorithm can automatically monitor the database performance on single and cluster computers, and generate multiple partitioning solutions for necessary database tables when the database performance is degrading and implement them on appropriate computing nodes in order to get better performance. So the literature review focuses on database partitioning and related topics.

The literature review is divided into five main sections. Since database vertical partitioning is a subfield of the database performance tuning area, firstly Section 2.2 briefly discusses the existing work done in the database performance tuning area. After that, Section 2.3 and Section 2.4 focus on the static and dynamic vertical database partitioning algorithms. Section 2.5 discusses database partitioning techniques for cluster computers. The last section, Section 2.6, makes conclusions for the literature review.

2.2 Database Performance Tuning Techniques

Research for database performance tuning is a branch of the self-managing database design area. Generally this area includes self-managing database indexing ([2], [3]), self-managing database caching [4], self-managing database partitioning [5], self-tuning database parameters [6], etc. Some of those algorithms are fully automatic while others are partially automatic (or semi-automatic). A fully automatic algorithm does not need human interference for feedback when the algorithm is running, while a semi-automatic algorithm does. For example, the algorithm called WFIT in [2] is a semi-automatic one. It needs the feedback V (either positive or negative) of the DBA according to a workload stream Q. A positive feedback on an index “a” implies that the DBA favors the algorithm’s index recommendations that contain “a”; otherwise the DBA gives a negative feedback. The algorithm called AutoStore in [5] is a fully automatic one. Such algorithms have the ability to know when the database performance is degrading and how to tune the database on one or more aspects to get better performance. Generally we call it the re-tuning ability of a database system. Re-tuning can be re-partitioning, re-indexing, re-caching, etc. The algorithm we introduced in this thesis is such an algorithm that has a fully automatic online approach and focuses on the vertical database partitioning and re-partitioning problem.

2.3 Static Database Vertical Partitioning

As discussed in [19], how to minimize disk I/Os is an important topic since the early 1970s. From that time on, algorithms have been developed to reduce I/Os by clustering huge complex data arrays in order to reduce page reading from secondary memory [20]. The early database vertical partitioning algorithms are mainly static. This means in early time people use a fixed workload to predict the future workload of a database system. These two workloads should have very similar query patterns so that the partitioning result can work well in the future. We describe different static partitioning algorithms in the following sections.

2.3.1 The Bond Energy Algorithm

The first well-known attribute clustering algorithm was introduced in 1972 with the name of Bond Energy Algorithm (BEA) [21]. This algorithm is an attribute affinity based algorithm. It uses a two-dimension array to represent the relationship between two different kinds of variables, row variable and column variable. Each column represents one kind of variable and each row represents the other kind of variable. Each element in the array is represented by a numerical value, which usually is an integer, to show the relationship between the row and column variables corresponding to this element. This algorithm permutates rows and columns of the array in order to group elements with similar values together. At the end of the algorithm, elements with similar values are located in the same block in the array and each block can be considered as a cluster. When doing permutation on the array, the algorithm needs user’s subjective judgment to tell the similarity of elements; so this algorithm is hard to implement without human’s interpretation. Sometimes blocks may have overlaps and some elements do not belong to any block. It means the clustering result is not always as good as what people expect.

2.3.2 The Navathe’s Vertical Partitioning Algorithm

Later after the development of BEA, another new important algorithm emerged, which was called Navathe’s Vertical Partitioning (NVP) [14]. NVP is also an attribute affinity matrix based algorithm. This was the first time that a clustering algorithm considers the frequency of queries and reflects the frequency in the attribute affinity matrix on which clustering was performed. NVP is an extension and improvement of BEA. This algorithm repeatedly does binary vertical partitioning (BVP) on a larger fragment, which is gotten from the previous BVP, to form two fragments. This process will not stop until the fragment cannot be partitioned further. An evaluation function is used to determine which fragment should be selected and whether it can be partitioned further. This algorithm is only suitable for a small query set because of the O(2^n) time complexity where *n* is the number of times the binary partitioning (which is proportional to the number of queries) is repeated. If fragment overlapping is allowed, the time complexity will be even bigger than that.

2.3.3 The Optimal Binary Vertical Partitioning Algorithm

Later after the Navathe's Vertical Partitioning algorithm, a new algorithm called the Optimal Binary Partitioning algorithm was proposed [22]. This algorithm uses the branch and bound method [12] to construct a binary tree in which each node represents a query. The left branch of a node represents those attributes being queried by the query that are included in a reasonable cut (a reasonable cut is a binary cut that partitions the attributes into two sets; in these two sets at least one of them is a contained fragment which is the union of a set of attributes that the query accesses). The right branch of a node represents the remaining attributes. If all attributes of an unassigned query are contained in the fragment of the current node then this query needs not be considered as the child of the current node. This algorithm focuses on a set of important queries rather than attributes themselves. It does reduce time complexity compared to the Navathe's Vertical Partitioning algorithm but it does not consider the impact of query frequency, and also its run time still grows exponentially with the number of queries.

2.3.4 The Graph Search Vertical Partitioning Algorithm

Some algorithms use a graph search technique when doing data clustering. [23] is one of the examples. It is a graph theory based clustering technique. The attributes that are usually queried together are used to form a similarity graph. Vertices of the graph are elements and edges connect elements that have similarity values higher than a predefined threshold. Clusters are the sub graphs with edge connectivity containing more than half of the number of vertices. When this technique is implemented for database vertical partitioning, a vertex represents an attribute and an edge represents how often the two attributes connected by this edge will appear together in the same query. Then the algorithm will traverse the graph and divide the graph into several sub graphs, each of which represents a cluster. This technique considers frequent queries and infrequent queries to be the same and this may lead to inefficient partitioning results. This is because attributes that are usually accessed together in infrequent queries but are not accessed together in frequent queries may be put in the same fragment if all queries are considered to be the same.

2.3.5 The Eltayeb's Vertical Partitioning Algorithm

A more recent attribute clustering algorithm was introduced in [11], which uses the idea of performing clustering based on an attributes affinity matrix from [14]. This algorithm starts with a vertex V that satisfies the least degree of reflexivity and then finds a vertex with the maximum degree of symmetry among V's neighbors. Once such a neighbor is found, both V and its neighbor are put into a subset. The neighbor becomes the new V. The process continues to search for neighbors of the most recent V recursively until a cycle is formed or no vertex is left. After that, the fragments will be refined using a hit ratio function. The disadvantage of this technique is similar to the disadvantage of the algorithm proposed in [23] since infrequent queries are treated the same as frequent queries.

2.3.6 The AutoClust Vertical Partitioning Algorithm

Along with the increase of processor's speed and the sophistication of software, database systems become cleverer and more powerful than ever before. Researchers then realized that a database system itself can give a lot of help on physical database design to developers. It gives the researchers a new direction when they are working on physical database design. Such an example is presented in [24]. In this paper, a new idea of using the query optimizer of a database system for automated physical design was proposed. The authors introduced a cost estimation technique which uses the query optimizer of a database system for physical database design. A query optimizer can gather useful statistic information from system views and perform *what-if* calls [15] to help the database system to make a decision on selection of the best query execution plan among multiple query execution plans without running the query.

The AutoClust algorithm [19] is an example of using a query optimizer to generate partitioning solutions. Totally there are five steps in the AutoClust algorithm. In Step 1, an attribute usage matrix is built based on a query set indicating which query accesses which attributes. In Step 2, the closed item sets (CIS) [10] of attributes are mined. An item set is called closed if it has no superset having the same support which is the fraction of queries in a data set where the item set appears as a subset [10]. CIS can tell us which attributes are accessed frequently by the same query. We want to keep such attributes in the same cluster (partition) together as much as possible. In Step 3, augmentations to add the primary key of the original database table to each existing closed item set are done to form the augmented closed item set (ACIS) which is a combination of CIS and the primary key. Then duplicate ACIS are removed. In Step 4 an execution tree is generated where each leaf represents a candidate attribute clustering (or vertical database partitioning) solution. Finally, in Step 5, the solutions are submitted to the query optimizer of the database system that will process the queries for cost estimation and the solution with the best estimated query cost is chosen as the final vertical database partitioning solution. The authors also proposed some ideas of how to extend AutoClust to cluster computers. According to the authors' ideas, multiple partitioning solutions are selected from the candidate partitioning solution pool. These selected partitioning solutions are the best solutions (i.e. the ones that have the best average query estimated costs) and are implemented on the computing nodes in a round robin order. Every future incoming query will be routed to the computing node containing the partition that gives the best estimated query cost for the query execution. AutoClust is a static or semi-automatic partitioning algorithm. This algorithm uses a fixed query set as the algorithm input and mines CIS from that query set to generate multiple partitioning solutions. This algorithm runs only once; if users want to do re-partitioning they have to monitor the database performance and trigger AutoClust by themselves. The authors did not present any performance results of their algorithm.

2.4 Dynamic Database Vertical Partitioning

All of the partitioning algorithms reviewed so far use a fixed query set to perform vertical partitioning. In those algorithms, the fixed query set has to be very similar to the query set used in the future. If the future query set is very different from the one currently used by the algorithms, then the future partitioning solution generated by the algorithms will not be accurate. Once the solution is implemented on the database system, the performance may become even worse. As we know it is very possible that queries change over time in many today's applications, such as ad-hoc business analysis and retail distribution management. So the above static partitioning algorithms are not good choices for today’s database systems. Dynamic database partitioning algorithms are thus needed. The following sections review these algorithms.

2.4.1 The DYVEP Vertical Partitioning Algorithm

The DYVEP vertical partitioning algorithm was proposed in [27]. It is a dynamic vertical partitioning algorithm for distributed database systems. DYVEP monitors queries in order to accumulate relevant statistics for the vertical partitioning process. It analyzes the statistics in order to determine if a new partitioning is necessary; if yes, it triggers a vertical partitioning technique (VPT) to generate a new partitioning solution. The VPT could be any existing VPT that can make use of the available statistics. The algorithm then checks to see if the new partitioning solution is better than the one in place; if yes, then the system reorganizes the database according to the new partitioning solution. This algorithm depends heavily on the VPT being used and the set of rules that it develops to decide when to trigger the VPT. The algorithm does not address how it would take advantage of distributed databases that have partial or full replication so that queries can be directed to nodes that yield the best query costs to execute them. DYVEP is not a new algorithm to be more exact as it cannot work without a VPT algorithm. It just gives a way of how to re-run an existing VPT algorithm automatically.

2.4.2 The AutoStore Vertical Partitioning Algorithm

To the best of our knowledge, AutoStore is the first true dynamic vertical partitioning algorithm that has recently been presented in [5]. AutoStore is a self-tuning data store that can free DBAs from monitoring the current workload. This algorithm has the ability to automatically collect queries and partition the data at checkpoint time intervals. When enough queries are collected, the algorithm will update the old attribute affinity matrix and do permutation on this matrix to make the matrix have the best quality (the quality can be calculated using BEA [21]). Then AutoStore will do clustering on the new matrix and use the greedy method to find out the best way to cluster the attributes in the new matrix based on the estimated cost from the query optimizer. Once the best clustering (partitioning) solution is found, the costs of building the new partitions and the estimated benefit brought by the new partitions will be calculated separately. If the benefit is larger, the re-partitioning action will be triggered; otherwise re-partitioning will not be triggered.

AutoStore is the first solution to solve the vertical database partitioning problem with a fully automatic online approach. Unfortunately this algorithm has several problems. The first one is that the authors did not give any clue of how many queries (in the article this number is called CheckpointSize) we need to collect so that we will have enough statistics to permutate and cluster the attribute affinity matrix. The second problem, which is a very serious one, is that this algorithm will run re-partitioning checking (i.e. checking to see if re-partitioning is needed) every time the number of queries collected is equal to the CheckpointSize no matter what performance trend it has at that time. This means that the re-partitioning checking process will be triggered even when the performance is still good. As we know re-partitioning checking is very expensive and should not be run too often; but AutoStore does re-partitioning checking before checking the performance trend. This is not an inefficient way to do re-partitioning.

2.5 Database Partitioning on Cluster Computers

Though there are many database vertical partitioning algorithms, how to improve response time of ad-hoc heavy-weight On-Line Analytical Processing (OLAP) queries is still an open problem. Today cluster computers are widely used for solving such a problem and database partitioning on such computers has gained much interest for various database applications [28]. Some database partitioning algorithms have been developed for distributed databases on cluster computers (e.g. [29] and [25]). Some of them are database table level algorithms (e.g. [25]) and some of them are database schema level algorithms (e.g. [29]).

Along with the development of new partitioning algorithms, some evaluation work has been done on evaluating the performance of distributed databases on cluster computers. The results show that distributed databases can greatly improve the performance and satisfy business requirements [30]. Because of this, distributed databases have become widely used and important for many applications, which call for more research to find ways to improve their physical database design. This section reviews existing database partitioning algorithms for cluster computers.

2.5.1 The ElasTraS Algorithm

A recent database partitioning algorithm, ElasTras, on cluster computers was introduced in [29]. This algorithm is a database schema level partitioning algorithm. The key idea of database schema level partitioning is that for a large number of database schemas and applications, transactions only access a small number of related rows which can be potentially spread across a number of database tables. ElasTraS takes the root database table of a tree structure database schema as the primary partitioning database table and other nodes of the tree as the secondary partitioning database table. The primary partitioning database table is partitioned independently of the other database tables using its primary key. Because the primary database table's key is part of the keys of all the secondary database tables, the secondary partitioning database tables are partitioned based on the primary database table's partition key. Then all partitions will be spread across several Owning Transaction Managers, which own one or more partitions and provide transactional guarantees on them. Analyzing a database schema is much more difficult than analyzing a database table and this algorithm is generally configured for static partitioning purposes.

2.5.2 The FINDER Algorithm

The problem of determining data placement is an important one in shared-nothing MPP database systems. In [37], the authors proposed an algorithm called FINDER that aims to find the optimal distribution policy for a set of database tables given a target workload. The assumption of this algorithm is that the workload is given and the future workload should be very similar to the one used by the algorithm. So it is a static algorithm. For a given database table set T = {T_1, …,T_t}, this algorithm can find the distribution policy D = {X_1, …, X_t} where X_i is a set of attributes and Ti is distributed based on X_i. The tuples of a database table will be assigned to different segments according to the hash value of X_i. We can see that this algorithm is used to statically and horizontally partition the database tables on cluster computers.

2.5.3 The Amossen Algorithm

Generally an OLAP application contains lots of many-row aggregates and likely benefit from parallelizing its queries on multiple sites and exchanging small sub results between the sites after the aggregations. It means that the queries happening on such system are usually very complex. In an OLTP application, on the other hand, there are many short queries with no many-row aggregates or few-row aggregates and the queries only gather all attributes from the same site. It means that the queries happening on such system are usually very simple. The Amossen algorithm [15] is a partitioning algorithm used only on OLTP applications. In [15] the authors present a cost model and then use simulated annealing to find the close-to-optimal vertical partitioning with respect to the cost model. In this algorithm, the queries must be very simple and have to avoid breaking single-sitedness. So we can also regard this algorithm as a static algorithm.

2.6 Conclusions for Literature Review

Database table level distributed database partitioning on cluster computers is easier than database schema level partitioning since all single node partitioning algorithms, like those we discussed earlier ([11], [14], [21], [22], [23]), can be deployed on multiple nodes; however, their disadvantages still exist on each node. Researchers have recognized that vertical partitioning for databases on cluster computers needs an algorithm which can generate multiple solutions at a time so that different solutions can be deployed on different nodes for appropriate queries. This means that for a particular query it can always be directed to the node with the best solution for processing.

We have reviewed many database partitioning algorithms. Those proposed in ([9], [11], [14], [21], [22], [23], [25]) are static as they use a fixed query set, i.e. there is no way for those algorithms to monitor new queries and perform re-partitioning based on the new queries automatically. Among those algorithms only the one proposed in [25] is designed for cluster computers. It has a specific scheduling and routing algorithm for cluster computers; however, it has no ability to monitor the database performance, i.e. it cannot re-partition the database automatically. The partitioning algorithms proposed in ([5], [27]) are dynamic but they are not designed for cluster computers. So we can see that none of the algorithms we have reviewed can dynamically partition databases on cluster computers.

Chapter 3: The Proposed Algorithm for Static Vertical Database Partitioning

In this chapter, we present our algorithm for static vertical database partitioning, called Filtered AutoClust. This algorithm is an improvement of the existing algorithm, AutoClust, proposed in [19]. In the following sections, we first describe AutoClust, and then discuss its weaknesses and present our Filtered AutoClust.

3.1 AutoClust

The AutoClust algorithm was introduced in [19]. It makes use of data mining and database query optimization to do vertical partitioning. It works for both single and cluster computers. However, as AutoClust was still in its early stage of research, no performance studies were presented in that paper. Below we first describe how AutoClust works for a single computer and then we show how the authors have extended it for cluster computers.

The AutoClust algorithm for a single computer consists of five steps. In Step 1, an attribute usage matrix is built based on a query set showing which attributes are accessed by which queries. In Step 2, the closed item sets (CIS) [10] of attributes are mined in order to identify which attributes are accessed frequently by the same query. An item set is called closed if it has no superset having the same support [10] which is the fraction of queries in the query set where the item set appears as a subset. For such attribute sets, they need to be kept together as an independent cluster (partition) as much as possible. In Step 3, augmentations to add the primary key of the original database table to each existing closed item set are done to form the augment closed item set (ACIS) which is a combination of CIS and the primary key. Then duplications in ACIS are removed. In Step 4 an execution tree is generated where each leaf represents a candidate attribute clustering solution. Finally, in Step 5, the solutions are submitted to the query optimizer of the database system that will process the queries for query cost estimation and the solution with the best average query cost estimation is chosen as the final solution. The cost is based on a special unit of workload used by the query optimizer to estimate how much work needs to be done in order to process the query. This query cost includes I/O cost and operator (or CPU) cost. We use “cost” to represent the total of the two types of costs in our examples and experimental results unless we specify the individual type of cost specifically.

When a database application is deployed on a cluster computing system, AutoClust can be extended as discussed in [19] to perform vertical partitioning on the database tables. Here AutoClust considers the case when the entire database is fully replicated on every computing node. The algorithm can generate multiple vertical partitioning solutions at a time and a different solution may have a different estimated query cost for the same query. Different solutions are implemented on different nodes so that a query can always be executed on the node with the best solution for the query if possible. If this node happens to be busy then the query can be executed on the node with the second best solution, and so on. This can greatly benefit the system performance.

To illustrate how AutoClust works on cluster computers, we use an example to explain it. The database table and queries of the example come from the TPC-H benchmark [9]. In order to avoid the impact of indexing on the partitioning result, we do not index the database tables. This means that the query optimizer will do a full database table scan when it generates the execution plans for the query which needs to retrieve data from the database table. So if a query is trying to access N attributes of a database table (those N attributes may be either from the selection part or the condition part of the query), then this query can be re-written in the format of selecting N attributes from the database table. The query optimizer will give the same estimated query cost for the query before re-writing and for the query after re-writing. We use the database table SUPPLIER as shown in Table 1 to describe the algorithm. The ten queries out of the twenty-two TPC-H queries accessing the SUPPLIER database table are shown in Table 2. The queries' frequencies are generated randomly.

Table 1. Attribute Information of the Database Table SUPPLIER

Attribute Name	Database Table	Data Type
S_ACCTBAL (A)	Supplier	Decimal
S_ADDRESS (B)	Supplier	Varchar
S_COMMENT (C)	Supplier	Varchar
S_NAME (D)	Supplier	Char
S_NATIONKEY (E)	Supplier	Int
S_PHONE(F)	Supplier	Char
S_SUPPKEY(G)	Supplier	Int

Table 2. Attribute Usage Matrix

Queries	Attributes							Frequency (%)
	A	B	C	D	E	F	G	
q_1	1	1	1	1	1	1	1	14.09
q_2	0	0	0	0	1	0	1	16.96
q_3	0	0	0	0	1	0	1	4.01
q_4	0	0	0	0	1	0	1	1.06
q_5	0	0	0	0	1	0	1	0.12
q_6	0	0	0	0	1	0	1	14.26
q_7	0	1	0	1	0	1	1	1.06
q_8	0	0	1	0	0	0	1	19.26
q_9	0	1	0	1	1	0	1	14.46
q_{10}	0	0	0	1	1	0	0	14.72

Step 1: run Steps 1 – 4 of the AutoClust algorithm for a single computer. This step will produce the possible vertical partitioning solutions for one computing node. The AutoClust algorithm for a single computer will generate the execution tree to get the candidate vertical partitioning solutions S_i's where each S_i is a set of attributes. The execution tree is shown in Figure 1. Once the execution tree is built successfully, the estimated query costs are produced as shown in Table 3.

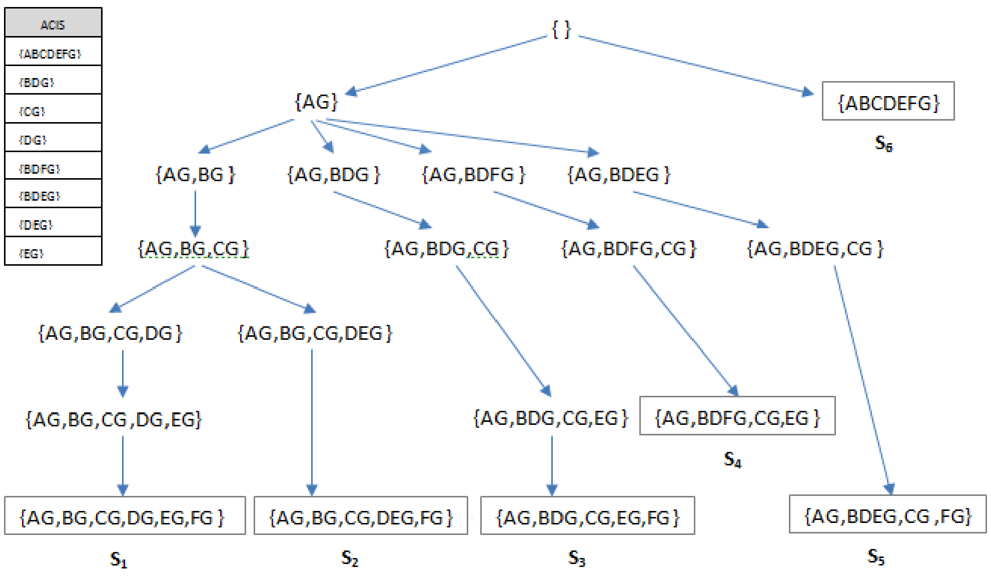

Figure 1.Execution Tree Generated by AutoClust for a Single Computer on the SUPPLIER Database Table

Table 3. Estimated Query Costs for Candidate Vertical Partitioning Solutions

Solution	**Estimated Query Cost Produced by Query Optimizer**										**Aggregate Cost**
	q_1	**q_2**	**q_3**	**q_4**	**q_5**	**q_6**	**q_7**	**q_8**	**q_9**	**q_{10}**	
S_1	98	7	7	7	7	7	49	32	44	26	33
S_2	90	18	18	18	18	18	49	32	36	18	34
S_3	89	7	7	7	7	7	40	32	35	35	32
S_4	82	7	7	7	7	7	33	32	41	41	33
S_5	82	28	28	28	28	28	41	32	28	28	37
S_6	68	68	68	68	68	68	68	68	68	68	68

Step 2: rank the solutions in increasing order based on their aggregate estimated query costs (we call aggregate cost for short) as the solutions will be chosen later in

Step 4 based on their increasing order of aggregate costs for implementation on computing nodes (i.e. the best solution will be chosen first). Applying Step 2 on Table 3, we get Table 4.

Table 4. Estimated Query Costs for Possible Candidate Vertical Partitioning Solutions after Ranking

Solution	Estimated Query Cost Produced by the Query Optimizer										Aggregate Cost
	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	
S_3	89	7	7	7	7	7	40	32	35	35	32
S_4	82	7	7	7	7	7	33	32	41	41	33
S_1	98	7	7	7	7	7	49	32	44	26	33
S_2	90	18	18	18	18	18	49	32	36	18	34
S_5	82	28	28	28	28	28	41	32	28	28	37
S_6	68	68	68	68	68	68	68	68	68	68	68

Step 3: remove the solutions the costs of which are larger than the cost of No Partition (i.e. the cost when no partition is performed on the database table). When we do vertical partitioning on a database we can decrease the I/O cost, but the trade-off is that we might increase the chance of join operations, which leads to an increase in the operator cost. So, this removal is necessary to avoid the case of over-partitioning. For the SUPPLIER database table in our example, the estimated query cost is 68 for No Partition, and as we can see in Table 4, no solution has an aggregate cost larger than 68, so we do not need to remove any solution in this step.

Step 4: from the solutions resulted from Step 3, implement them on the computing nodes according to the increasing order of the solutions' aggregate costs so that the best solution is implemented on the first node, the second best solution on the second node, and so on. If there are still nodes remaining after all the solutions have been implemented, then repeat the loop again, i.e., implementing the best solution on the first remaining node, the second best solution on the second remaining node, and so on. As an example, if our cluster computers have three nodes, node 1, node 2 and node 3, then we will implement the first three solutions highlighted in Table 4 on these three nodes, i.e., S_3 on node 1, S_4 on node 2, and S_1 on node 3.

Step 5: from the results generated in Step 1, construct a vertical partitioning solution choice table as shown in Table 5 that shows which solution is the best, second best, and so on for each query. Then from this table and from the solution implementation in Step 4, construct a query routing table as shown in Table 6 that shows which implemented solution (and on which node it is implemented) is the best, second best, and so on for each query so that when a query arrives, it will be directed to the node where its best solution is implemented. If this node is busy, then the query will be routed to its second best node, and so on. For example, when query q_1 arrives, it will be directed to the first node of its first choice, which is node S_4 in Table 6. If this node is busy, then q_1 will be directed to the implemented node of its second choice, which is node S_3.

Table 5. Best Vertical Partitioning Choices for Each Query

Choices	**Queries**									
	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}
1st	S_6	S_3	S_3	S_3	S_3	S_3	S_4	S_3	S_5	S_2
2nd	S_4	S_4	S_4	S_4	S_4	S_4	S_3	S_4	S_3	S_1
3rd	S_5	S_1	S_1	S_1	S_1	S_1	S_5	S_1	S_2	S_5
4th	S_3	S_2	S_2	S_2	S_2	S_2	S_1	S_2	S_4	S_3
5th	S_2	S_5	S_5	S_5	S_5	S_5	S_2	S_5	S_1	S_4
6th	S_1	S_6	S_6	S_6	S_6	S_6	S_6	S_6	S_6	S_6

Table 6. Query Routing Table for a Three Nodes Cluster Computer

Choices	**Queries**									
	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}
1st	S_4	S_3	S_3	S_3	S_3	S_3	S_4	S_3	S_3	S_1
2nd	S_3	S_4	S_4	S_4	S_4	S_4	S_3	S_4	S_4	S_3
3rd	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_1	S_4

From the procedure of AutoClust, we can see that multiple queries can run in parallel on the most efficient nodes. The node with a better aggregate estimated query cost for a query has a higher priority to be selected to process the query. So the total response time of the query set can be greatly decreased.

3.2 The Weaknesses of AutoClust

AutoClust was first introduced in [19] without any performance study. In [25], we presented an experimental performance evaluation of this algorithm. However at that time AutoClust was just compared with a baseline case where the database tables are in their original forms without any partitioning. Also an issue that we discovered is that AutoClust takes a long time to run when the database table that needs to be partitioned is extremely large on both column size and row size. In our experiments using the TPC-H benchmark [9], the largest database table is the database table called LINEITEM which contains 16 attributes and more than 6,000,000 rows with the total size of 742MB. AutoClust spent more than 24 hours getting the needed solutions. In many cases, AutoClust does not need to be run very often, but in the two cases when either the query set on the database changes substantially over time or the database has been heavily updated (e.g. new database tables added and old database tables removed, or attributes change), the database tables do often need a new partitioning solution. In such situations, it is not acceptable to have a partitioning algorithm that requires a long time to finish. So we need to improve the running time performance of AutoClust.

In the following paragraphs we discuss the weaknesses of AutoClust that cause the long running time and how our algorithm, called Filtered AutoClust, modifies AutoClust to improve on these weaknesses.

First let us examine the following steps that are performed when AutoClust works on cluster computers:

Step 1: Run Steps 1 – 4 of the AutoClust algorithm for a single computer to produce all possible vertical partitioning solutions. Those 4 steps are the following:

1.1. Generate an attribute usage matrix based on the query set. This matrix shows which attribute is accessed by which query.

1.2. Mine closed item sets from the matrix generated from Step 1.1 in order to identify which attributes are accessed more frequently by the same query.

1.3. Augment each subset in the CIS by adding the primary key to form the augment closed item sets (ACIS). Duplicate subsets in ACIS are then removed.

1.4. Generate the execution tree where each leaf represents a candidate vertical partitioning solution.

Step 2: estimate the query cost for each solution generated in Step 1 using the query optimizer and rank the solutions in increasing order based on their aggregate costs.

Step 3: remove those solutions the costs of which are larger than the cost of No Partition to avoid the over-partitioning problem.

Step 4: implement the solutions resulted from Step 3 according to the increasing order of the solutions' aggregate costs so that the best solution is implemented on the first node, the second best solution is implemented on the second node, and so on.

Step 5: construct a query routing table for the solutions implemented in Step 4 so that the system can know which query should be executed on which node.

When we ran AutoClust on the LINEITEM database table on our 32 node cluster computer and monitored the running process, we got the algorithm's running time results as shown in Table 7. From Table 7 we can see that Step 2 and Step 4 are busy most of the time. The reason is that in order to get the estimated query cost from the query optimizer for a vertical partitioning solution, we have to create the corresponding sub database tables for this solution so that the query optimizer can generate the query execution plan steps represented in the MEMO status ([31], [32]) which is the data structure of a query optimizer and calculate the estimated cost for the query execution plan. For the LINEITEM database table, it has more than 6,000,000 rows, if a vertical partitioning solution contains 5 partitions, the database system has to create 5 sub database tables with more than 6,000,000 rows in each sub database table. This is a time consuming process especially when we have a large number of vertical partitioning solutions and the whole process in done by one node in AutoClust. For the LINEITEM table, the number of candidate vertical partitioning solutions is 7963, but most of them are over partitioned, i.e. most of the candidate solutions perform worse than No Partition. So if we can eliminate most over partitioned solutions before sending them to the query optimizer and then distribute the solutions among the computing nodes to calculate the cost in parallel, then we can save a lot of running time. In Step 4, the final vertical partitioning solutions are implemented on 32 computing nodes and this step involves sub database table creations which cannot be avoided since it is our final goal. So we can summarize the two weaknesses of AutoClust as follows:

Weakness 1: AutoClust can pass too many over-partitioned candidate solutions to the query optimizer. The query optimizer has to spend a lot of time on creating useless sub database tables in order to calculate the average query estimated cost for those over-partitioned solutions.

Weakness 2: All the query cost estimation tasks are done by one computing node. The query optimizers located on other computing nodes are not fully utilized.

Table 7. AutoClust's Running Time for the LINEITEM Database Table on a 32 Node Cluster Computer

Step Number	Running Time
Step 1	2,889 ms
Step 2	>24 hours
Step 3	3 ms
Step 4	6,296,875 ms
Step 5	13 ms

In the next section, we introduce the improved version of AutoClust called Filtered AutoClust that eliminates its weaknesses identified above.

3.3 The Filtered AutoClust Algorithm

First we need to clarify the purpose of attribute clustering or vertical partitioning again. We know generally a query does not access all the attributes in a database table. But when the database system processes a query, it has to retrieve all the attributes of the database table from the disk, we call it an I/O operation. An I/O operation is a kind of mechanical operation and is much slower than a CPU operation. So we want to cluster those attributes accessed by the same query in order to reduce I/O operations, hence to improve the query response time.

We discover the following theorem: *the maximum attribute set As_i of a database table T accessed by the same query Q must be a closed item set (CIS) mined through the attribute usage matrix*. Below we prove this theorem using a contradiction.

We know that if an item set is a CIS, then it has no superset having the same support [10]. Suppose that As_i is not a CIS, then there must be an attribute set As_j with equal support which is the superset of As_i. So there must be at least one attribute in As_j which is from the database table T but not in As_i; but we have defined that As_i is the maximum attribute set from T accessed by query Q, this contradicts with what we have obtained above. So the theorem is correct.

Generally it is impossible to make every maximum attribute set of each query as a partition since there will be too much attribute overlap if we partition a database table in this way; but we can just focus on those "important" queries, which are the queries with higher occurrence frequency, to make sure that the attribute set of such queries can be partitioned as a single partition. In order to determine the "important" queries we define a parameter f_i which is a query frequency threshold. If a query's frequency is higher than f_i we consider this query as an "important" query. In our Filtered AutoClust, we calculate f_i in the following way: $i = \frac{100\%}{number\ of\ query\ types}$, so if the frequency of a query is not lower than the average frequency we say this query is an "important" query. For instance, given an attribute usage matrix as shown in Table 8, totally we have 8 queries but the number of different query types is 6. So the average frequency for this query set equals to $\frac{100\%}{6} = 16.7\%$ and the query with a frequency higher than 16.7% will be regarded as an "important" query. We have to try to keep the attributes of those "important" queries in the same partition. Then those subsets of this partition in CIS can be removed in order to reduce the size of the CIS set. Because a large CIS set can cause a large ACIS set, and a large ACIS set can cause a large candidate vertical partitioning solution set, which increases the running time. Now we introduce the steps of Filtered AutoClust and illustrate how they are applied to the example given in Table 8.

Table 8. Attribute Usage Matrix of the CUSTOMER Database Table from the TPC-H Benchmark

Queries	Attributes								Frequency (%)
	A	B	C	D	E	F	G	H	
q1	0	0	0	1	1	0	0	0	21.8
q2	0	0	0	1	0	0	1	0	24.5
q3	0	0	0	1	0	0	1	0	5.8
q4	0	0	0	1	0	0	1	0	1.5
q5	1	1	1	1	0	1	1	1	25
q6	0	0	0	1	0	0	0	0	3.3
q7	0	0	0	1	0	1	0	0	9.4
q8	1	0	0	1	0	0	0	1	8.7

Step 1: generate the set of attribute set *As* and the corresponding frequency set *Fs* for each query

1.1. Generate the attribute usage matrix based on the query set.

1.2. Create the attribute set *As* for each type of query based on the matrix resulted from Step 1.1

1.3. Create the frequency set *Fs* for each query based on *As* and the matrix from Step 1.1

After implementing Step 1 on Table 8 we can get:

As = {{D,E},{D,G},{A,B,C,D,F,G,H},{D},{D,F},{A,D,H}}

Fs = {21.8%, 31.8%, 25%, 3.3%, 9.4%, 8.7%}

Step 2: mine closed item sets from the matrix generated in Step 1.1. Once we apply Step 2 on Table 8 we get the CIS set as {{D,E},{D,G},{A,B,C,D,F,G,H},{D},{D,F},{A,D,H}}.

Step 3: filter the CIS set based on *As* and *Fs* in order to remove unnecessary CIS.

3.1. Remove the attribute set from *As* whose frequency is below the average frequency. Once this step is done the new *As* is {{D,E},{D,G},{A,B,C,D,F,G,H}}.

3.2. For each attribute set *As*$_i$ in the new *As* remove its subset in the CIS set to form the new CIS set. Once this step is done we get the new CIS set as {{D,E},{D,G},{A,B,C,D,F,G,H}}.

3.3. Union the new *As* and the new CIS set to get the final CIS set.

Step 4: augment each subset in the final CIS set by adding the primary key to form the augment closed item set (ACIS). Remove the duplicate subsets in the ACIS set. When this step is applied on the final CIS set resulted from Step 3.3, we have the ACIS set as {{D,E},{D,G},{A,B,C,D,F,G,H}} (D is the primary key).

Step 5: generate the execution tree where each leaf represents a candidate vertical partitioning solution.

Step 6: distribute the candidate vertical partitioning solutions in the round robin order to each computing node to calculate the aggregate cost (average estimated query cost) for each solution using the query optimizer on each node. Then send the solutions with their costs back to the control node.

Step 7: rank the solutions in increasing order based on their aggregate costs and remove those solutions the costs of which are larger than the cost of No Partition.

Step 8: implement the solutions resulted from Step 7 according to the increasing order of the solutions' aggregate costs so that the best solution is implemented on the first node, the second best solution is implemented on the second node, and so on.

Step 9: construct the query routing table for the solutions implemented in Step 8 so that the system can know which query should be executed on which node.

The structure of the Filtered AutoClust algorithm on cluster computers is shown in Figure 3 and the pseudo code of the algorithm is shown in Figure 4.

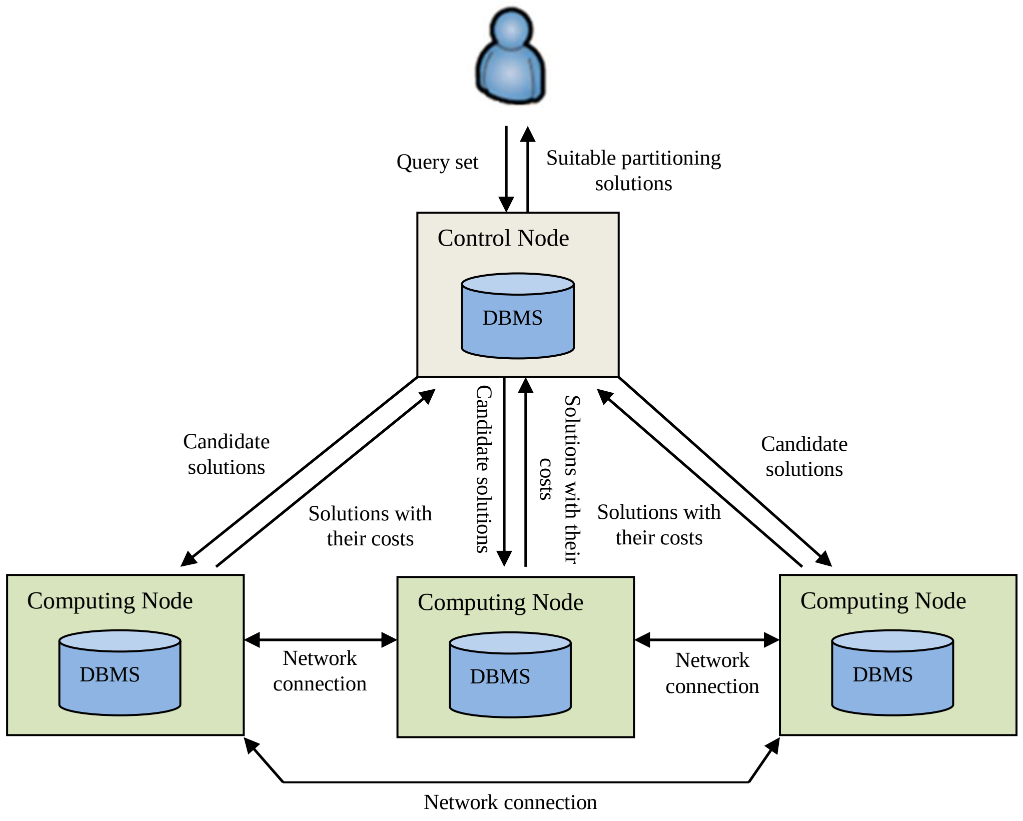

Figure 2. Architecture of Filtered AutoClust on Cluster Computers

```
Run Steps 1-2 of AutoClust for one node to generate the attribute usage matrix and CIS set;
Create the maximum attributes set As for each type of query and the corresponding frequency set Fs from attribute usage matrix;
Set filter query frequency threshold $f_i$ as 100%/Fs.size;
//find out the important queries
For each frequency $Fs_i$ in Fs
    if $Fs_i < f_i$ then
        remove the corresponding sub attribute set in As
    End if
End for
//filter CIS set
For each set $CIS_i$ in CIS
    if $CIS_i$ is a subset of any set in As then
        remove $CIS_i$ from CIS
    End if
End for
//expand CIS by adding As to it to avoid losing attributes
Union As and CIS to form the final CIS
Run step 3-4 of AutoClust for one node to generate the set S of all possible partitioning solutions;
Distribute solutions in S in round robin order to each computing node
For each computing node
    Calculate the average query estimated cost for each solution and send all solutions with their average query estimated cost back to control node
End for
Run step 2-5 of AutoClust for cluster computers to finish the best solutions implement on each computing node.
```

Figure 3. The Filtered AutoClust Algorithm on Cluster Computers

Filtered AutoClust is designed for cluster computers; it can be used on single computers but it will be more greatly benefited when using on cluster computers. Filtered AutoClust does not change the main idea of AutoClust; instead it just adds a few more steps to reduce the number of CIS and distributes the query cost estimation work to all computing nodes to reduce running time. That is why we call it an improvement of AutoClust.

3.3 Conclusions

In this chapter we formally describe the existing AutoClust algorithm proposed in [19] which can be used to vertically partition database tables on both single and cluster computers. This is the first database vertical partitioning algorithm that can provide multiple vertical partitioning solutions, but it is a static algorithm. The future query set pattern must be very similar to the one used to partition the database tables. This algorithm takes a long time to run if the database table which needs to be partitioned contains many attributes. So we present an improved version of AutoClust called Filtered AutoClust. Filtered AutoClust provides a way of filtering out the over-partitioned candidate solutions before submitting them to the query optimizer for estimated query cost calculation. As we know when the query optimizer calculates the query estimated cost, it does not really run the query, instead the query optimizer collects the statistic information from the system dictionary database tables or views to generate the estimated cost. However, we still need to create the sub tables which represent the sub partitions for each candidate partitioning solution so that the database system can store the new sub database table's information in the system views. Without those database tables' information, the query optimizer has no idea of how to calculate the estimated query cost. If the original database table is big, then the sub database table creation process will take a lot of time. This is caused by too many over-partitioned solutions. Our algorithm, Filtered AutoClust, solves this problem; it removes most over-partitioned solutions so that the query optimizer has less work to do. A key weakness of Filtered AutoClust is that it is a static algorithm. In the next chapter, we introduce SMOPD and SMOPD-C. They are the dynamic versions of Filtered AutoClust for single computers and cluster computers, respectively.

Chapter 4: The Proposed Techniques for Dynamic Vertical Database Partitioning: SMOPD and SMOPD-C

In Chapter 3 we introduced a static database vertical partitioning algorithm on cluster computers, Filtered AutoClust. In this chapter we show how to make this algorithm dynamic so that it can automatically monitor the database performance and re-partition the necessary database tables when the average performance is degrading. The single computer version of the dynamic vertical partitioning algorithm is called SMOPD (Self Managing Online Partitioner for Databases), and the cluster computer version is called SMOPD-C (Self Managing Online Partitioner for Databases on Cluster computers).

4.1 Self Managing Online Partitioner for Databases (SMOPD) on Single Computers

4.1.1 Structure Overview of SMOPD

In this section, we present the components of SMOPD which are shown in Figure 5. SMOPD is a vertical database partitioning algorithm which is designed with an online approach and has the ability to dynamically make the re-partitioning decision based on the statistic data collected from system statistic views. This system only partitions the database tables that need re-partitioning. For those database tables the current partitioning results of which still work well, the system can filter them out, and does not consider them for re-partitioning. This technique can save a lot of time since calculating partitioning candidates is always time-consuming.

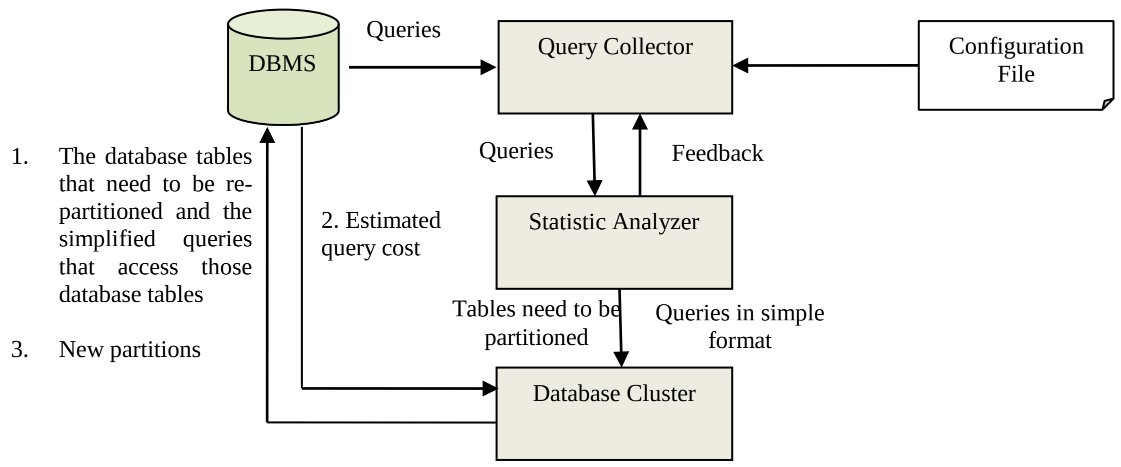

Figure 4. Architecture of SMOPD

In the SMOPD algorithm, there are three major components: query collector, statistics analyzer and database cluster, which we introduce in the following sections.

4.1.1.1 The Query Collector Component of SMOPD

First we must understand when we should do re-partitioning and why we want to do that. The answer is clear to us, that is we want to do re-partitioning when the current partitions' performance is not good any more. In order to achieve our goal we need to collect some statistics from the database system which can be used to tell us whether the system performance is good or bad. In our case we use the estimated query cost provided by the query collector optimizer to measure the system performance. As we have discussed in Chapter 2, researchers have proposed a new way of measuring the query performance for database partitioning by using the query optimizer component in a database system. A query optimizer can evaluate a query by building different execution plans and choosing the one with the least estimated cost since less estimated cost generally means less execution time. By doing this we can estimate how the query would perform if a new database partitioning solution, instead of the old database partitioning solution, is used without really running the query.

An important question is then whether we should collect the estimated query cost for all queries. The answer is not. We know that when a query is processed by a database system, some information may be retrieved from a hard disk and some information may be retrieved from the main memory. Information retrieval from the main memory is much faster than information retrieval from a hard disk. A vertical partitioning technique can help the database system improve the physical I/O performance significantly but not the logical I/O performance. Physical I/Os are the I/Os measured when data are accessed from a hard disk, while logical I/Os are the I/Os measured when data are accessed from the main memory. Here we introduce the first configuration parameter we use for our system, r , which is the physical read ratio threshold of a query. If the physical read ratio of a query is bigger than *r* we consider this query as a physical read mainly query and use 1 to mark this query; otherwise we consider this query as a logical read mainly query and use 0 to mark this query. Then the query workload can be transferred to a set containing 1's and/or 0's.

Let $f_n = \frac{q_1+q_2+q_3+\cdots+q_n}{n}$ where $q_i = \begin{Bmatrix} 1\ if\ physical\ read\ mainly \\ 0\ if\ logical\ read\ mainly \end{Bmatrix}$ be the second configuration parameter we use for our algorithm where q_i is the *ith* query and *n* is the total number of queries collected. This parameter represents the threshold of the percentage of physical read mainly queries in the whole query set we have. Only if the percentage of the physical read mainly queries is larger than the threshold f_n , we can say that we have enough queries collected by the query collector component for the statistics analyzer component to use.

From the discussion above we can see that the query distribution of a workload can be regarded as a binomial distribution if we use either 1 or 0 to represent a query. According to [34], when $n \geq 50, nf_n \geq 10, n(1-f_n) \geq 10,$ then a binomial distribution can be rounded with a normal distribution $N(np, \sqrt{np(1-p)})$.

The probability that a query is a physical read mainly query can be represented in the following format:

$$p = f_n \pm c_\alpha \, where \begin{Bmatrix} c_\alpha = z_\alpha \sqrt{\frac{f_n(1-f_n)}{n}} \\ f_n = \frac{q_1+q_2+q_3+\cdots+q_n}{n} \end{Bmatrix} \quad (1)$$

where c_α is the confidence interval of p and is the third configuration parameter of our algorithm, and z_α is a function of α which is the fourth configuration parameter that represents the confidence level of p. Since we already defined $c_\alpha, z_\alpha \, and \, f_n$, we can calculate *n* using the following formula:

$$n = \frac{f_n(1-f_n) \times z_\alpha^2}{c_\alpha^2} \quad (2)$$

Now we know that at least we need to collect *n* queries to ensure that the percentage of the physical read mainly queries is above the threshold f_n. In other words, in order to have enough statistics we need to collect at least *n* queries as given in Equation (2).

Once we collect *n* queries we need to consider the outlier queries in this query set. Outlier queries are those that occur rarely. Since the core algorithm of SMOPD depends on closed item sets mining and outlier queries may impact the whole closed item sets, and thus may impact the performance of SMOPD, we need to identify and filter out outlier queries. So we define the last parameter f_t which is the query frequency threshold. If a query's frequency is less than or equal to f_t, we consider such query as an outlier query and will filter it out from the query set. f_t can be defined by the DBA manually or calculated by the system automatically using some classic outlier detection technique. Below we give two ways to define this parameter.

The first way is to manually give an exact threshold value to f_t based on the average query frequency. Here we give an example to show how to calculate this value; this way is also used in Section 5.2.1 of Chapter 5 that discusses the experiment model to evaluate SMOPD. For each run, we randomly select 60% of the TPC-H queries, which are 13 different types of queries out of 22 types of queries in the TPC-H benchmark. The average query frequency (*Avg_Freq*) is$\frac{100\%}{13} = 7.7\%$. We define the f_t as the 10% of the average query frequency, so $f_t = 10\% * 7.7\% = 0.77\%$. The manual configuration way is easy to use when the DBA knows the structure of the query workload and already has an idea about the importance for each type of query. The importance of a query means the support or frequency of a query in the workload. We use this configuration method in our experiments presented in Chapter 5 since the number of different types of queries in the TPC-H benchmark is not large and we know all the queries in this benchmark.

The second way of defining f_t is to have the algorithm automatically calculate its value using some specific outlier detection algorithm. In our algorithm, for instance, we can use the classic Grubbs' outlier test (GOT) algorithm [35]. In the GOT algorithm, there are three approaches to find an outlier. The first one is left-sided detection in which the algorithm only checks the smallest value to see whether it is an outlier; the second one is right-sided detection in which the algorithm only checks the biggest value to see whether it is an outlier; and the third one is double-sided detection in which the algorithm checks both the smallest and biggest values to see whether they are outliers. In a query set we consider a query with a much smaller frequency than the average query frequency to be a possible outlier query since such kind of query occurs very rarely. Hence we need to use left-sided detection in our case. If a query has a very high frequency, it means that this query is very important and will be executed very often. This kind of query should not be regarded as an outlier query.

The test statistics g which is used to determine whether or not a query sample is an outlier is calculated according to the following formula [35]:

$$g = \frac{\bar{x}}{s} - \frac{x_{min}}{s} \qquad (3)$$

where $\bar{x}$ is the average query frequency, x_{min} is the minimal query frequency and s is the standard deviation of the query workload according to query frequencies. If the resulting test statistic g is greater than the critical value which can be found in the Grubbs' critical value statistical table [36], then it means that an outlier query is detected and we need to filter this query out. This algorithm can only find one outlier query for each run, so the above calculation has to be run multiple times occasionally in order to find all outlier queries. This configuration method for f_t is good to use when the DBA has no idea of how the future query workload will look like and what the frequency distribution of each query is. After filtering the less important queries out, finally we will get a query set *QS* which can be passed to the next component to do statistics analysis.

The whole process of query collector can be done by querying the system statistic view of a database. For example, in the database system MS Server, this view is saved as SYS.DM_EXEC_QUERY_STATS and in Oracle, this view is saved as

V_$SQLAREA. We can analyze these system views and get the query statistic information in order to calculate *n*.

4.1.1.2 The Statistics Analyzer Component of SMOPD

Once the query set *QS* is passed to the second component, Statistics Analyzer, this component will separate the *QS* into two subsets, *QS0* and *QS1*. *QS1* contains all the physical read mainly queries and *QS0* contains all the logical read mainly queries. *QS1* is the set we will use to determine the query performance trend as database partitioning might impact it.

In order to make the next component, Database Cluster, understand the query easily, the Statistic Analyzer will simplify each query, a process that we call query simplification. During this process, each query will be transformed into a simple format which contains only the original database tables and their attributes accessed by the query. The Statistics Analyzer first simplifies each query in *QS1* and then uses those simplified queries to rebuild the attribute usage matrix according to each original database table in the database. For example, in *QS1* we have the following query:

SELECT O_ORDERDATE, O_CUSTKEY, O_SHIPPRIORITY FROM ORDERS_1 O1, ORDERS_2 O2 WHERE O1.O_ORDERKEY=O2.O_ORDERKEY

After simplification this query will become:

ORDERS.O_ORDERDATE ORDERS.O_CUSTKEY ORDERS.O_SHIPPRIORITY ORDERS.O_ORDERKEY

ORDERS_1 and ORDERS_2 are the two current sub partition database tables (also called sub partitions for short) of the original ORDERS database table. We want to re-partition the original database table, so we need to replace the sub partition database tables with the original database table. The simplified format of the query can be read and understood by the Database Cluster component. For a complex query, such as a query that has an inner query, the query optimizer will convert it into a lot of small simple queries. For each simple query we can simply convert it into the above format.

Once the simplification work is done, an attribute usage matrix is built for each original database table as done in Filtered AutoClust. Each row of the attribute usage matrix represents the attributes that are accessed by the corresponding query and the percentage the query takes in the whole query set. Then the query optimizer will estimate the estimated query cost for each type of query in the matrix, and then a new estimated query cost is generated by calculating the average estimated query cost for all types of queries in the attribute usage matrix. These two steps are discussed in detail in Chapter 3. If for a database table, the new estimated query cost which is generated based on the new query set is bigger than the old estimated cost which was generated from the old query set during the previous partitioning process, then it means that the current partitioning solution's performance is degrading and a new database partitioning solution should be derived for this database table. Then the Statistics Analyzer will save this database table in a database table set *T*. The Statistics Analyzer examines all the database tables included in the set *QS1* so that it can find all the database tables of which the current partitioning solution does not perform as well as before and save them in *T* which will be passed to the next component, Database Cluster, to do re-partitioning checking.

At this point the work of the Statistics Analyzer component is completed, and *QS1* and *T* are then passed to the Database Cluster component to run the re-partitioning process. The key function of this component is to determine the time point when the re-partitioning process should be triggered. Re-partitioning is time consuming and should not be run very often. If the performance is still good then we should not carry out the re-partitioning action, which is why we want to check the performance trend first. Once we detect the decrease of performance trend, we need to know what database tables cause such a trend. Only those database tables need re-partitioning. The Statistics Analyze component is thus used to find the right time - when to do re-partitioning, and the right targets - what database tables need re-partitioning.

4.1.2.3 The Database Cluster Component of SMOPD

The third component, Database Cluster, runs our Filtered AutoClust algorithm that we have introduced in Chapter 3. This component uses *T* and *QS1* as input to generate multiple candidate database partitioning solutions and uses the query optimizer to select the best solution for each database table. There is a possibility that the new partitioning solution is the same as the old partitioning solution because, even though the change of the whole query workload causes the performance to be not as good as before, this partitioning solution may still be the best one among all the partitioning solutions generated by Filtered AutoClust.

The whole SMOPD algorithm is shown in Figure 10. First the Query Collector reads the pre-defined parameters, which are c_a, a and f_n, from the configuration file to calculate the minimum number of queries to be collected (lines 1-2). After that it starts to read the queries from the database system views that contain the detail information of the queries executed and puts all queries into a set *QS* (lines 3-6). When *QS* contains enough queries, the Statistics Analyzer starts to run and puts all the physical read mainly queries in *QS* into a set *QS1* (lines 7-19). In order to make the Database Cluster component understand the queries in *QS1*, the Statistics Analyzer creates a set *FQS* (Filtered Query Set) containing the simplified format of each query in *QS1* and replaces the sub partition database tables with the original database tables (lines 20-22 and 28-30). During this process the Statistics Analyzer reads the parameter *ft* from the configuration file and any query with a frequency less than *ft* is removed from *FQS* (lines 23-26, 31). Now *FQS* contains all the queries that will heavily impact the partitioning solution, so the Statistics Analyzer calculates their new estimated query costs and get the average estimated query cost based on the query frequencies, then compares this cost with the old average estimated query cost read from C_{old} in the configuration file (lines 32-39). If the new average estimated query cost is bigger than the old one, the original database table corresponding to the old cost needs to be re-partitioned and this database table will be kept in set *T*; otherwise the database table will be removed from further consideration for re-partitioning. If *T* is not empty then the Database Cluster component knows that some database tables must be re-partitioned; so it starts to run Filtered AutoClust on each database table in *T* to find a better partitioning solution to replace the current partitioning solution (line 40).

Input:

1. Database tables- *T*
2. Queries running on DB
3. Current estimate cost set for each database table in *T*- C_{old}
4. Physical read ratio threshold of a query- *r*
5. The ratio threshold of number of queries that satisfies *r*- *fn*
6. Query frequency threshold- *ft* (a query must occur at least *ft* percent in the whole query set)
7. Confidence interval- c_α
8. Confidence level- α

Output:

Partitioning results for each database table in *T* when this database table needs to be repartitioned

```
Step 1: estimate the number of queries N need to be read
$N = z_\alpha^2 * \frac{fn(1-fn)}{c_\alpha^2}$
Step 2: collect queries
Initialize two empty set QS0, QS1 and an integer parameter k = 0;
While $\frac{QS1.szie}{k*N} < fn$ or k = 0 do
  Read N queries and put them in the set QS
  For each query $q_i$ in QS
    Get $r_i$ of $q_i$;
    If $r_i<r$ then
      Put $q_i$ in QS0;
      Remove $q_i$ from QS;
    End if
    Else
      Put $q_i$ in QS1;
      Remove $q_i$ from QS;
    End else
  End for
k++;
End do
Step 3: simplify and filter queries in QS1

Simplify queries in QS1 and calculate frequency for each type of query, put each type of query with
its frequency in a new set FQS;
For each list FQS[i] in FQS
  If FQS[i].frequency <ft then
    Remove FQS[i] from FQS;
  End if
  Else
    Extract all database tables' names in FQS[i].query and replace them with original tables'
    names and put each original database table in set T;
  End else
End for
Step 4: evaluate the performance of current partitions
For each database table $T_i$ in T
  Construct attribute usage matrix $M_i$ using FQS;
  Get the estimate cost $C_{new}$ for $T_i$ using $M_i$;
  If $C_{new}$<= $C_{old}$[i] then
    Remove $T_i$ from T
  End if
End for
Step 5: run Filtered AutoClust using T and FQS
```

Figure 5. The SMOPD Algorithm

4.2 SMOPD for Cluster Computers: SMOPD-C

Till now, we have described each component and its functionality of the SMOPD algorithm and how it works on a single computer. As we know in a cluster computer, each computing node can be regarded as a single computing node and the SMOPD algorithm can be deployed on each computing node to monitor the performance of this node; but SMOPD does not have the ability to let a computing node communicate with other computing nodes, i.e. one computing node cannot detect the re-partitioning action of other computing nodes. The change of partitions on a computing node means the change of the estimated query cost for all queries running on this node. According to the query routing schedule algorithm we use in our Filtered AutoClust algorithm, a query is always routed to the computing node that can provide the best estimated query cost for this query (if this node is busy, the routing schedule will try the second best node and so on). If a computing node's partitions change, we must reconsider the routing scheme; but SMOPD cannot solve such a problem. So we need to develop a new algorithm that can cooperate with SMOPD to handle the query routing problem.

Before we describe the new algorithm which we called SMOPD-C, we should review how Filtered AutoClust deals with the query routing problem if we use a fixed query work load. In Chapter 3, we have described how Filtered AutoClust works on cluster computers. According to the assumption in Filtered AutoClust, the query set used in the future is the same as the one currently used, i.e. the queries in the routing table will not change. So the query routing table will keep unchanged in the future. This means we just need to construct the query routing table once when we first partition the database tables. That is why we did not consider the query routing table synchronization problem in Filtered AutoClust; However, in practice, queries (either query contents or query frequencies) change over time. This will cause a change in the query routing table finally. So, for our dynamic vertical partitioning algorithm on a single computer, SMOPD, to be extended correctly to a cluster computer, we cannot ignore the change in the extension. This means that the new query routing tables have to be constructed to replace the old query routing tables when re-partitioning happens.

In the new algorithm SMOPD-C, we treat all computing nodes as a whole system together. We use estimated query cost to represent the query response time, if the average estimated query cost of a query set for a database table is larger than the average estimated query cost of another query set for the same database table, then we say the current partitions of this database table might not be good any more, i.e., the performance of the current partitions of this database table might not be good any more. If the performance of a database table's partitions on the major number of computing nodes is not good then we say the average performance of all computing nodes is not good. We measure the average performance of all computing nodes and decide whether re-partitioning is needed or not. We only consider a bad performance to be happening on the whole cluster computer when at least the performance of half of the computing nodes is becoming worse.

In this algorithm one computing node will be used as a control node to count how many queries have been processed by the cluster computer so that SMOPD-C can start to check the performance trend for each computing node in the cluster computer once enough queries are collected on it. Now we describe each step of SMOPD-C one by one using an example. In the example we deploy the TPC-H database on a cluster computer with 8 computing nodes. We show how this algorithm works on the PART database table. Before we run this algorithm we need to partition the PART database table once. The query set we use is shown in Table 9.

Table 9. Query Set Used to Generate the Original Partitions

Query No.	Number of Query Runs	Query Text
2	939	SELECT L_RETURNFLAG, L_LINESTATUS, SUM(L_QUANTITY) ….
4	915	SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT……
6	951	SELECT SUM(L_EXTENDEDPRICE*L_DISCOUNT) AS……
8	54	SELECT O_YEAR, SUM(CASE WHEN NATION = 'BRAZIL'……
9	6	SELECT NATION, O_YEAR, SUM(AMOUNT) AS SUM_PROFIT……
12	195	SELECT L_SHIPMODE, SUM(CASE WHEN O_ORDERPRIORITY …
13	115	SELECT C_COUNT, COUNT(*) AS CUSTDIST FROM (SELECT……
14	133	SELECT 100.00 * SUM(CASE WHEN P_TYPE LIKE 'PROMO%'……
15	979	SELECT P_BRAND, P_TYPE, P_SIZE, COUNT(DISTINCT ……
16	915	SELECT SUM(L_EXTENDEDPRICE) / 7.0 AS AVG_YEARLY……
17	331	SELECT C_NAME, C_CUSTKEY, O6.O_ORDERKEY, O_ORD ……
19	735	SELECT S_NAME, S_ADDRESS FROM SUPPLIER_2 S2, SUPP……

After we run Filtered AutoClust using the above query set we got two candidate vertical partitioning solutions:

Candidate solution S1:

[{P_COMMENT,P_PARTKEY},{P_MFGR,P_PARTKEY},{P_PARTKEY,P_RETAILPRICE},{P_BRAND,P_PARTKEY,P_SIZE,P_TYPE},{P_CONTAINER,P_PARTKEY}, {P_NAME,P_PARTKEY}]

S1	Query 8	Query 9	Query 14	Query 15	Query 16	Query 19
Est. cost	354	339	354	354	1054	339
Avg. cost	577					

Candidate solution S2:

[{P_COMMENT,P_PARTKEY},{P_MFGR,P_PARTKEY},{P_PARTKEY,P_RETAILPRICE}, {P_BRAND,P_CONTAINER,P_PARTKEY}, {P_NAME,P_PARTKEY}, {P_PARTKEY,P_SIZE}, {P_PARTKEY,P_TYPE}]

S2	Query 8	Query 9	Query 14	Query 15	Query 16	Query 19
Est. cost	246	339	246	2135	250	339
Avg. cost	927					

The way of deploying the two partitioning solutions on the cluster computer is shown in Table 10. In Table 10 we can see the estimated cost of each query for each computing node.

Table 10. Estimated Query Costs of Two Partitioning Solutions on Different Nodes

		Query 8	Query 9	Query 14	Query 15	Query 16	Query 19
Node1	S1	354	339	354	354	1054	339
Node2	S2	246	339	246	2135	250	339
Node3	S1	354	339	354	354	1054	339
Node4	S2	246	339	246	2135	250	339
Node5	S1	354	339	354	354	1054	339
Node6	S2	246	339	246	2135	250	339
Node7	S1	354	339	354	354	1054	339
Node8	S2	246	339	246	2135	250	339

We set the five parameters in SMOPD, *a (*confidence level), c_a (confidence interval), *r* (physical read ratio threshold), f_n (ratio threshold of number of queries that satisfies *r)*, and *ft* (query frequency threshold) to 95%,5%, 20%, 20% and 100%, respectively. Then we start SMOPD-C which consists of the following steps:

Step 1: estimate the number of queries N that needs to be collected.

After the first step we can get N, which is 245. Then we randomly select 60% of the TPC-H queries and assign each query a random frequency. So we get the following query set as shown in Table 11.

Table 11. Query Set Used to Re-partition Database Tables

Query No.	Freq. %	Average Physical Read Ratio	Count	Query Text
1	14.5%	27%	36	SELECT S_ACCTBAL, S_NAME, N_NAME, P_PARTKEY……
2	6.7%	50%	16	SELECT L_RETURNFLAG, L_LINESTATUS, SUM(L_QUANTITY) ….
3	9.3%	50%	23	SELECT L_ORDERKEY, SUM(L_EXTENDEDPRICE……
4	9.9%	51%	24	SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT……
5	12.6%	50%	31	SELECT N_NAME, SUM(L_EXTENDEDPRICE * (1 - L_DIS……
6	3.6%	50%	9	SELECT SUM(L_EXTENDEDPRICE*L_DISCOUNT) AS……
7	10.3%	50%	25	SELECT SUPP_NATION, CUST_NATION, L_YEAR……
8	5.8%	61%	14	SELECT O_YEAR, SUM(CASE WHEN NATION = 'BRAZIL'……
9	7.1%	57%	17	SELECT NATION, O_YEAR, SUM(AMOUNT) AS SUM_PROFIT……
10	3.7%	50%	9	SELECT C_CUSTKEY, C_NAME, SUM(L_EXTENDEDPRICE * ……
11	9.7%	47%	24	SELECT PS_PARTKEY, SUM(PS_SUPPLYCOST * PS_AVAILQT……
13	2.9%	51%	7	SELECT C_COUNT, COUNT(*) AS CUSTDIST FROM (SELECT……
15	3.9%	5%	10	SELECT P_BRAND, P_TYPE, P_SIZE, COUNT(DISTINCT ……

Step 2: count the queries processed by each computing node to see whether N queries have been collected.

In this step we randomly select a query from the above query set and send it to a random computing node to process it (we have described the routing scheme in Filtered AutoClust in Chapter 3) until all queries are processed.

Step 3: run Steps 2-4 of the SMOPD algorithm on each computing node to check the performance once N queries have been processed by the cluster computer.

In this step the new average estimated query cost for each computing node is calculated and shown in Table 12.

Table 12. Different Cost Results of the Partitioning Solutions on Different Nodes

		New Average Cost	Old Average Cost	Total Cost
Node1	S1	53	30	689
Node2	S2	11	36	286
Node3	S1	42	30	672
Node4	S2	20	36	420
Node5	S1	42	30	672
Node6	S2	32	36	832
Node7	S1	42	30	672
Node8	S2	37	36	851

Step 4: check the number of computing nodes that have performance getting worse; if the performance of half of the computing nodes becomes worse, go to Step 5; otherwise go back to Step 2.

In this step we can see that among 8 computing nodes, the performance of 5 computing nodes is getting worse; so re-partitioning will be triggered. Then we go to the last step.

Step 5: run all steps of Filtered AutoClust for the cluster computer using the N queries to partition the database table.

Once we run Step 5 we get the new partitioning solutions for the PART database table as follows:

New partitioning solution S1_new:

[{P_COMMENT,P_PARTKEY},{P_CONTAINER,P_PARTKEY},{P_PARTKEY,P_RETAILPRICE},{P_BRAND,P_PARTKEY,P_SIZE,P_TYPE},{P_MFGR,P_PARTKEY}, {P_NAME,P_PARTKEY}]

New partitioning solution S2_new:

[{P_COMMENT,P_PARTKEY},{P_CONTAINER,P_PARTKEY},{P_PARTKEY,P_RETAILPRICE},{P_BRAND,P_PARTKEY},{P_MFGR,P_PARTKEY,P_SIZE,P_TYPE}, {P_NAME,P_PARTKEY}]

The above new partitioning solutions will be implemented on each computing node and the query routing table will be re-constructed. Since we have described those detail steps in Filtered AutoClust we will not describe them here anymore. The whole algorithm is shown in Figure 6.

```
Input parameters:
    1. Physical read ratio threshold of a query- r
    2. The ratio threshold of number of queries that satisfies r- fn
    3. Query frequency threshold- ft (a query must occur at least ft percent in the whole query set)
    4. Confidence interval- $c_\alpha$
    5. Confidence level- α
Output:
Suitable partitioning solutions for each database table which needs re-partitioning
Step1: calculate N which is the number of queries that need to be collected
1       $N = z_\alpha^2 * \frac{fn(1-fn)}{c_\alpha^2}$
Step 2: monitor the queries that are processed since the last partitioning until N queries are reached
2       while less than N queries have been processed
3         continue monitoring
4       end while
Step 3: performance checking on each node
5       let $p_i$ to be the performance of the current partitions of a database table on node i
6       run steps 2-4 of SMOPD to check the performance of the current partitions of a database table on
        each node
7       set $p_i$ to 1 if the performance is still good, otherwise set it to 0
Step 4: re-partitioning checking
8       if $\frac{\sum_{i=1}^{n} \mathrm{p_i}}{n} \geq 50\%$, where n is the total number of computing nodes
9         goto step 5
10      else
11        goto step 2
12      end if
Step 5: run Filtered AutoClust using the new query set on the database table
```

Figure 6. the SMOPD-C Algorithm

From the description above we can see in the algorithm SMOPD-C, we just combine SMOPD and Filtered AutoClust together. The advantage is that we can avoid the routing table synchronization problem among different computing nodes and just update it one time when we do re-partitioning for a database table. The disadvantage is that we do unnecessary database table re-partitioning on those computing nodes for which the performance is still good. Also the N queries may not be evenly distributed to each computing node.

4.3 Conclusions

In this chapter we introduced the SMOPD algorithm which makes our vertical database partitioning algorithm, Filtered AutoClust, dynamic on a single computer. Then we proposed an algorithm called SMOPD-C to combine SMOPD and Filtered AutoClust to dynamically partition database tables on a cluster computer. Together with the three algorithms, the database performance can be monitored automatically and the database re-partitioning process can be performed dynamically without human interference when the database performance is degrading.

Chapter 5: Performance Evaluation

We conduct experiments evaluating the performance of the proposed algorithms: Filtered AutoClust, SMOPD and SMOPD-C. In this chapter, we present the experiment models, performance metrics and the experiment results.

5.1 Static Vertical Partitioning

5.1.1 Experiment Model

We build an experiment model to compare the performance of our static version partitioning algorithm, Filtered AutoClust, with the performance of the three algorithms: the baseline case which we call No Partition (i.e. the case when the database tables are not partitioned), the AutoClust algorithm [19], and the Eltayeb's algorithm [5]. We use the TPC-H benchmark database tables as our dataset and the TPC-H queries [9] as the query set. As we described in Section 3.1 in Chapter 3, we assume there is no index impact on the database table, so we re-write the TPC-H queries in a different format with the database table names followed by the attributes accessed by the queries only. Totally there are 8 database tables: ORDERS, SUPPLIER, LINEITEM, PARTSUPP, CUSTOMER, PART, NATION and REGION. For the NATION and REGION database tables, they are very small as they have only 5-25 rows, so we do not consider them in our experiments. The database system we use is Oracle 11g Express Edition. When we test our algorithm on a single computer we run it on a desktop computer with a processor of Intel Core 2 Quad Q8400, RAM of 3 GB and hard disk of 300 GB. When we run our algorithm on a cluster computer we run it on OSCER [33]. It has Red Hat Linux Enterprise 6 running on it. For each general computing node in OSCER, it has a Sandy Bridge E5-2650 2.0 GHz CPU, 32 GB RAM and 250 GB SATA disk. The algorithms are implemented in Java. In our experiments, all the query frequencies are randomly assigned.

5.1.2 Performance Metrics

We measure the performance of the static vertical partitioning algorithms based on three performance metrics: the total estimated query cost by the query optimizer of the Oracle database system we use, the estimated query cost improvement and the algorithm’s running time.

Total Estimated Query Cost:

When we test our algorithm on a cluster computer, we calculate the total estimated query cost for the whole query set. The query optimizer uses the estimated query cost to estimate the performance of a query plan. Generally the less estimated query cost means the less query response time; so we can use the total estimated query cost to represent the total response time of the whole query set. If the total estimated query cost of a partitioning algorithm A is less than the total estimated query cost of a partitioning algorithm B, then it means the partitioning result of algorithm A can provide better query response time than that of algorithm B. In other words, the total estimated query cost is a key factor to measure how well a partitioning algorithm works.

Estimated Query Cost Improvement:

Once we know the total estimated query costs of two algorithms that we want to compare against each other, we also want to know how much better one algorithm works compared to the other algorithm. We can use the estimated query cost improvement metric to measure the improvement. The estimated query cost improvement of an algorithm A over an algorithm B is calculated using the formula,

$\frac{100\%.(EstCostB - EstCostA)}{EstCostB}$ where EstCostB is the average estimated cost of a query set when algorithm B is used to partition the database and EstCostA is the average estimated cost of the same query set when algorithm A is used to partition the same database. If the result is bigger than zero, then we say that the partitioning result of algorithm A provides better average query response time than algorithm B.

Real Running Time:

When we test the improvement of Filtered AutoClust over AutoClust, we need to use the real running times of the two algorithms since Filtered AutoClust is designed to improve the real running time of AutoClust. In the rest of the thesis we use “running time” to refer to “real running time” of an algorithm.

5.1.3 Experiment Results

5.1.3.1 Performance Study of Static Vertical Partitioning Algorithms Based on Estimated Query Cost

a. Impacts of Number of Computing Nodes

When we study the impact of number of computing nodes we need to fix the database table. We use the ORDERS database table as the test database table since this database table has the performance improvement close to the average and has large numbers of rows and attributes. We measure the estimated query cost of Filtered AutoClust, AutoClust, Eltayeb and No Partition when running 10,000 queries with random frequency. The results are shown in Figure 7. From Figure 7 we can derive the following results:

1. The performance in terms of estimated query cost of all four algorithms is getting better as the number of computing nodes increases. More computing nodes means

less work for each computing node so the performance improves when the number of computing nodes is higher.

2. Eltayeb performs even worse than No Partition for the ORDERS database table. The reason is that Eltayeb does not consider the impact of query frequency; it treats different queries with different frequencies equally.
3. The performance of AutoClust is the same as the performance of Filtered AutoClust because both algorithms give the same partitioning solutions.
4. AutoClust and Filtered AutoClust have the best performance, No Partition has the second best performance, and Eltayeb has the worst performance on the ORDERS database table.

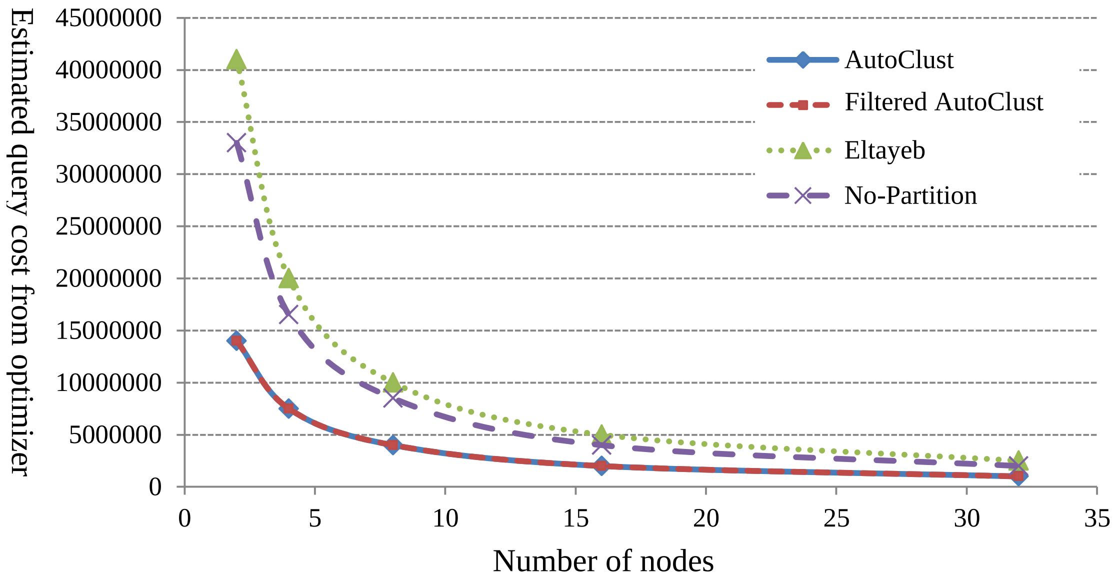

Figure 7. Impact of Number of Computing Nodes in a Cluster Computer for ORDERS Database Table

b. Impacts of Database Table Size

When we study the impact of database table size we need to fix the number of nodes. We set the number of computing nodes to 16. From Figure 8 we can get the following results:

1. Filtered AutoClust can give 7%-62% performance improvement in terms of estimated query cost comparing with Eltayeb.
2. For the ORDERS database table, Filtered AutoClust provides the best performance improvement; and for the PARTSUPP database table, Filtered AutoClust provides the least performance improvement.
3. Filtered AutoClust provides the same partitioning solutions as Eltayeb for the CUSTOMER database table; so there is no performance improvement for this database table.
4. The average performance improvement of Filtered AutoClust over Eltayeb for all the database tables is 32%.

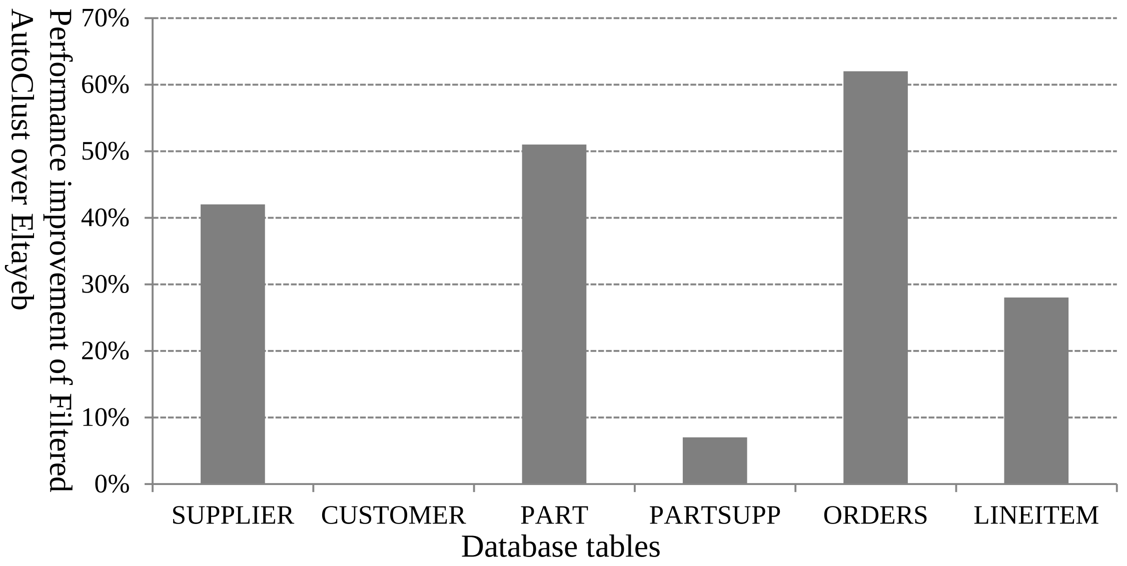

Figure 8. Performance improvement of Filtered AutoClust over Eltayeb on a 16 Nodes Cluster Computer for Different Database Tables

Filtered AutoClust is much more efficient than AutoClust in terms of algorithm's running time. We report the experiment results that support this conclusion in the next section.

5.1.3.2 Performance Study of Static Vertical Partitioning Algorithms Based on Running Time

a. Tests Using the TPC-H Benchmark

The comparison on real running time is only meaningful when the database table is large enough. We know that every database partitioning algorithm, whether it is a vertical partitioning algorithm or a horizontal partitioning algorithm, does not run as often as queries. It runs only when big changes are applied on the database. Such big changes include database table insertions, attribute insertions, database table deletions, substantial changes in query workloads , etc. In other words, a partitioning algorithm runs occasionally and thus its running time is not that important if the database table is not large. We conduct tests using the TPC-H benchmark for AutoClust, Filtered AutoClust and Eltayeb using 16 computing nodes. The results are shown in Table 13. From Table 13 we can see that which algorithm we use on small database tables really does not matter if those algorithms can provide the same partitioning solutions; but as we discussed in Section 5.1.3.1, Eltayeb cannot guarantee a better vertical partitioning solution in terms of estimated query cost than No Partition for some cases; so either Filtered AutoClust or AutoClust can be used when the database table size is small. For larger database tables (such as LINEITEM), the running time (greater than 172,813 seconds) of AutoClust is not acceptable if re-partitioning is needed very often.

Table 13. Real Running Time of Filtered AutoClust, AutoClust and Eltayeb Using 16 Computing Nodes

Database Table Name	Total Size (KB)	Number of Attributes	Number of Tuples	Algorithm's Running Time (in seconds)			
				No Partition	Eltayeb	Filtered AutoClust	AutoClust
REGION	1	3	5	0	1	2	3
NATION	3	4	25	0	1	1	1
SUPPLIER	1,377	7	10,000	0	1	2	8
CUSTOMER	23,776	8	150,000	0	1	8	50
PART	23,570	9	200,000	0	1	17	65
PARTSUPP	116,196	9	800,000	0	1	97	295
ORDERS	167,923	9	1,500,000	0	1	149	1,689
LINEITEM	742,054	16	6,000,000	0	2	1,575	>172,813

We also conduct tests for different numbers of computing nodes using the largest database table LINEITEM from the TPC-H benchmark. Figure 9 shows the running time results of Filtered AutoClust, AutoClust and Eltayeb. From Figure 9 we can derive the following results:

1. AutoClust has the worst running time (the running time of AutoClust should be larger than the time shown in Figure 9, but we just use 2,880 minutes to represent the running time since we configure the system to terminate any session of which the execution time is larger than 48 hours).
2. Eltayeb has the best running time compared with Filtered AutoClust and AutoClust. This is because Eltayeb is not a query optimizer integrated algorithm. It does not use estimated query cost to measure the performance, i.e., during the running of Eltayeb algorithm, it does not create any database table partitions to evaluate the performance except for the final partitioning solution. As we know table creation is very time consuming, but in a query optimizer integrated algorithm, this step cannot be voided. Both Filtered AutoClust and AutoClust are query optimizer integrated algorithms, so their running time is much longer than that of Eltayeb.

3. When the number of computing nodes increases the running time of Filtered AutoClust is decreasing but the running time of Eltayeb does not change. This is because Filtered AutoClust is designed for cluster computers and the query cost estimation work is distributed to all computing nodes. Eltayeb is designed for a single computer and cannot be benefitted on cluster computers. From Table 13 we can see that Filtered AutoClust takes about 26 minutes to generate the suitable partitioning solution and Eltayeb takes just 2 seconds to generate the suitable partitioning solution. However, from Figure 8 we can see that the partitioning solution provided by Filtered AutoClust has a performance improvement of 29%. It means that for every query from the query set used by our algorithm, if it wants to access the LINEITEM table using the partitioning solution provided by Eltayeb, it has to spend 29% more time than using the partitioning solutions provided by Filtered AutoClust. As these two algorithms are designed for applications in which query workloads are fixed or do not change frequently, database partitioning is not expected to take place often which means the DBA is not expected to have to run these algorithms often. At the same time, many of these applications have high query arrival rates; so fast query response time (i.e. low estimated query cost) is critical. Therefore, the improvement of Filtered AutoClust over Eltayeb in terms of estimated query cost outweighs the long running time of Filtered AutoClust.

There are two keys reasons of the poor running time performance of AutoClust: (1) AutoClust does not have any function to filter over-partitioned candidate solutions; (2) AutoClust does not distribute candidate solutions to computing nodes for query cost estimation, i.e. all query cost estimation work is done on the control node; but for Filtered AutoClust, it is a different story. Filtered AutoClust is designed for cluster computers and fully uses the powerful computation ability of cluster computers. It can distribute the query cost estimation work to each computing node to fully use the computing resources.

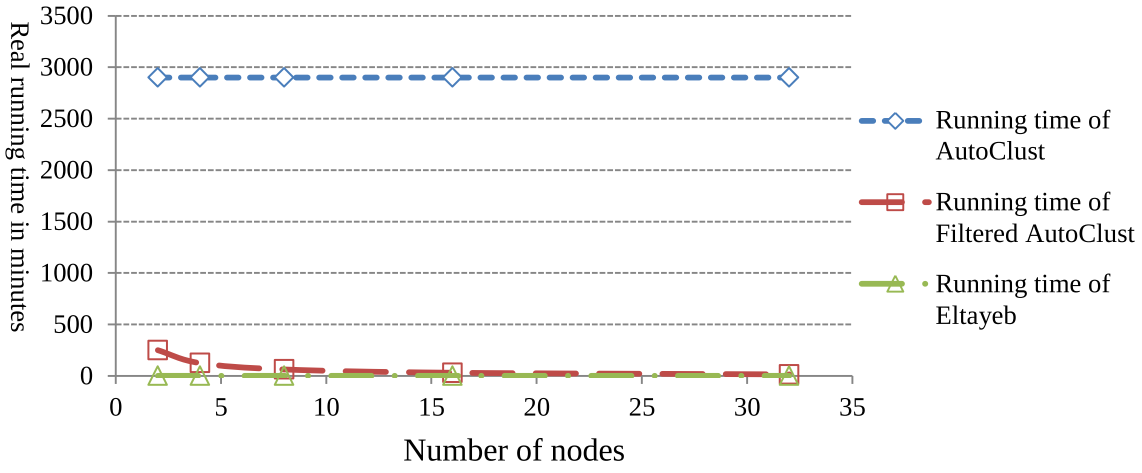

Figure 9. Running Time of AutoClust, Filtered AutoClust and Eltayeb on Cluster Computers of 2, 4, 8, 16 and 32 Nodes for the LINEITEM Database Table

b. Tests Using a Synthetic Database and Synthetic Queries

In order to study how different numbers of query types and different numbers of attributes impact the running time of Filtered AutoClust, we create a synthetic database to perform our tests. This is because the TPC-H benchmark cannot be used to conduct such studies as its largest database table, LINEITEM, has only 16 attributes, and the other database tables have so few attributes. The database schema we create is shown in Figure 10 where“TABLE_9” denotes a database table of 9 attributes; “TABLE_11” denotes a database table of 11 attributes and so on. The arrow in the schema means the primary key in the database table at the start of the arrow is related with the foreign key in that database table at the end of the arrow. This schema is very similar to the one of the TPC-H benchmark.

The attributes of each database table are shown in Table 14. The query set we use for our test is shown in Table 15. The first column of the query set table represents the query number (ID), the second column represents how many times the corresponding query is executed, and the third column represents the attributes this query accesses from the corresponding database tables. First we conduct experiments to test the impact of different numbers of attributes by running both AutoClust and Filtered AutoClust on a single computer for 8 different query types. The results are shown in Figure 11. From Figure 11 we can see that the running time of Filtered AutoClust is always better than the running time of AutoClust. If the number of attributes is large, the running time improvement of using Filtered AutoClust is always great.

Then we conduct experiments to study the impact of different numbers of query types on Filtered AutoClust. In these experiments we fix the test database table as TABLE_21. The results are shown in Figure 12. From the results we can see that when the number of different query types grows the running time improvement of Filtered AutoClust over AutoClust increases from 73% to 96%. The average running time improvement of Filtered AutoClust over AutoClust is about 82%. Filtered AutoClust can always provide great running time improvement.

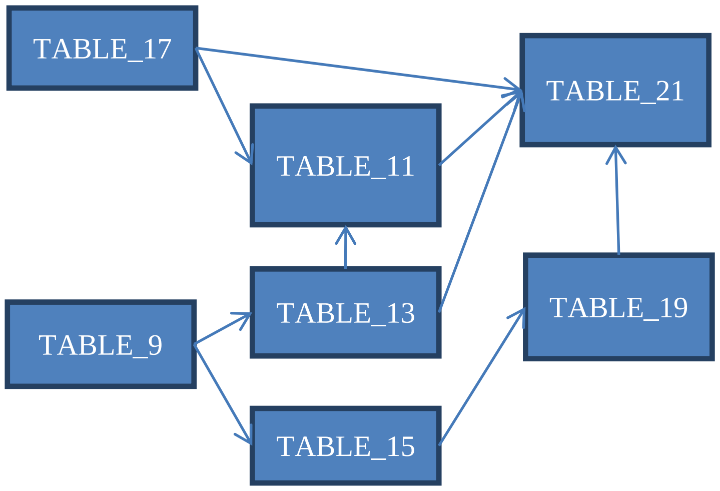

Figure 10. The Synthetic Database Schema Created for Our Tests

Table 14. Attribute List for Each Database Table in Our Experiments

Database Table Name	Attribute List
TABLE_9	A1,A2,A3,A4,A5,A6,A7,A8,A9
TABLE_11	B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11
TABLE_13	C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13
TABLE_15	D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15
TABLE_17	E1,E2,E3,E4,E5,E6,E7,E8,E9,E10,E11,E12,E13,E14,E15,E16,E17
TABLE_19	F1,F2,F3,F4,F5,F6,F7,F8,F9,F10,F11,F12,F13,F14,F15,F16,F17,F18,F19
TABLE_21	G1,G2,G3,G4,G5,G6,G7,G8,G9,G10,G11,G12,G13,G14,G15,G16,G17,G18,G19,G20,G21

Table 15. Query Set Used in Our Experiments

Query No.	Query Times	Attributes Accessed by the Query
1	939	G2,G3,G7,G8,G9,G11,G15,G16,G21
2	716	A1,C1,C2,B2,B1,E1,A3,A4,A7,A9,B4,B5,B6,C4,C5,C8,C9,C10,E2,E3,E5,E13,E14
3	769	D1,F1,F2,G4,D11,D13,F3,F5,F6,F8,F16,F17,G7,G8,G16
4	915	F1,G3,F13,F16,F17,G2,G6,G13,G14
5	862	A1,C2,D2,D1,F2,F1,G20,A4,A8,C1,C4,C5,C6,C12,D5,D14,F7,F16,G7,G8
6	951	G7,G8,G11,G13,G16
7	204	A1,C2,D2,D1,F2,F1,G20,A6,A7,C1,C6,C9,C12,C13,D5,D6,D15,F3,F5,F8,F17,F18,G4,G7 ,G8,G12,G16
8	881	A1,D2,D1,F2,F1,G3,A3,A7,A8,D4,D8,D9,D10,D12,D14,F7,F16,F17,G2,G7,G8,G13,G15
9	725	A1,C2,C1,B2,A5,A6,A9,B5,B6,B8,B9,B11,C5,C6,C12
10	195	F1,G3,F9,F14,F16,G6,G14,G16,G19
11	115	D1,F2,D5,D6,F1,F7,F11,F18
12	113	E1,G2,E6,E13,E16,G7,G10,G16,G3, B5,B8,B9
13	54	C1,G4,C5,C8,C10,C11,G8,G12,G16,B5,B6,B10
14	979	C1,B1,B2,E1,B5,B6,C4,C7,C9,C12,C13,E2,E3,E8,E10,E13,E16
15	915	E1,G2,E7,E10,E12,G3,G8,G10,G11
16	331	D1,F2,F1,G4,D8,D14,D15,F5,F7,F16,F17,G11
17	661	E1,G2,E3,E7,E10,E12,E14,E15,G3,G7,G8,G10,G11,G18,G19
18	735	A1,C2,C1,B1,B2,B7,E1,G2,A5,A8,B8,B9,B11,C5,C8,C10,E9,E14,E16,G3,G10,G11,G16, G20
19	748	A1,C2,C5,G5,G4,F1,A5,A6,C4,C10,F9,F11,F12,F14,F18,G6,G14,G20
20	308	D1,F2,D5,D4,D12,F4,F10
21	255	D2,D4,D7,D8,D9,D14,A1,A4,A5,A8
22	213	B2,B4,B5,B8,E1,E5,E7,E8,E12,E14
23	422	B2,B5,B9,B10,E1,E8,E9,E16

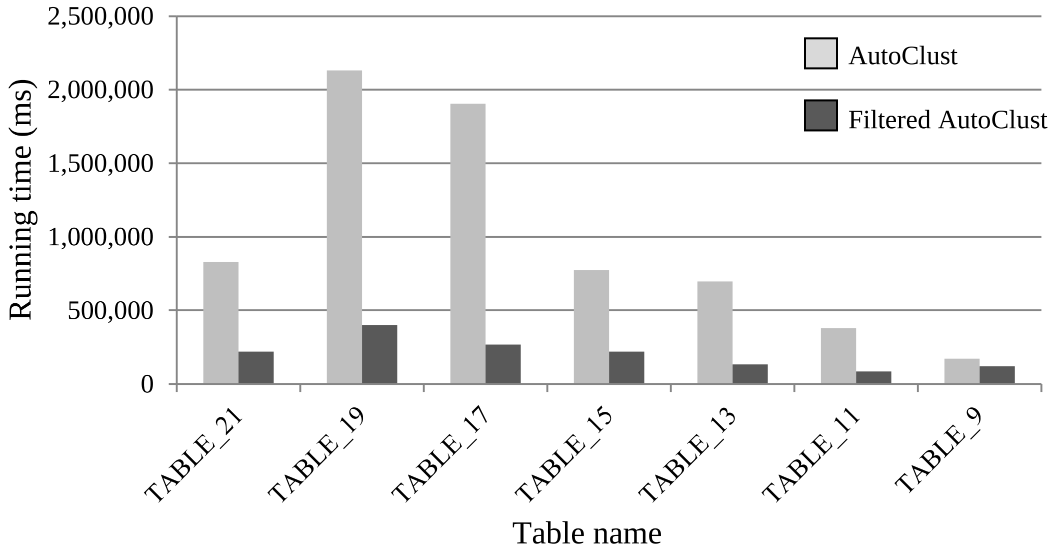

Figure 11. Running Time of AutoClust and Filtered AutoClust Using 8 Different Type Queries for Each Database Table in the Synthetic Database

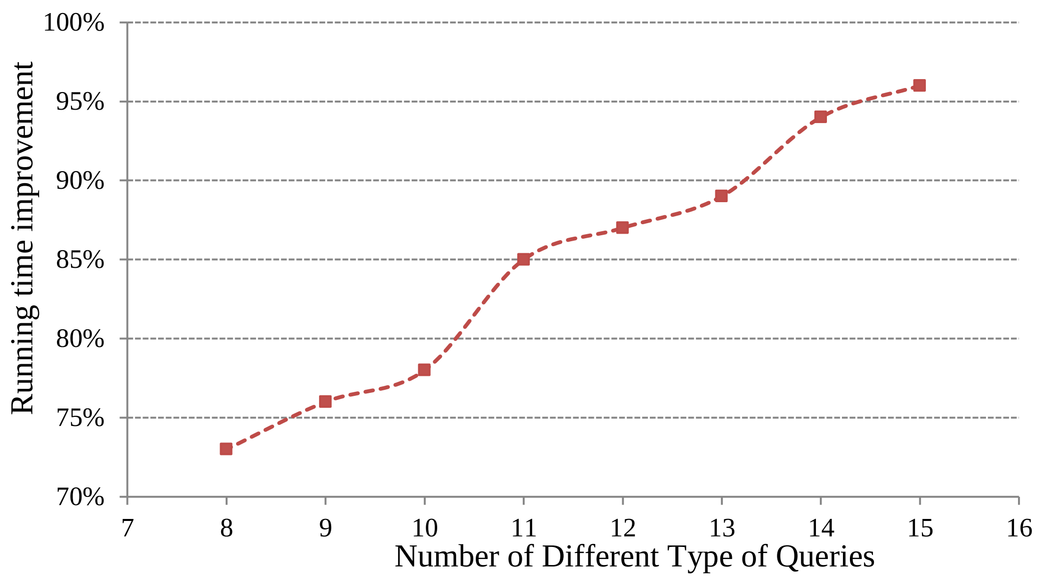

Figure 12. Running Time Improvement of Filtered AutoClust over AutoClust Using Different Numbers of Query Types on TABLE_21

5.2 Dynamic Vertical Partitioning

5.2.1 Experiment Model

The hardware, software, dataset and query set used in testing our dynamic vertical partitioning algorithms, SMOPD and SMOPD-C, are the same as those in testing the static vertical partitioning algorithms in Section 5.1, so we do not describe them again here; but we need to specify the new parameters in SMOPD and SMOPD-C and a cost model for results analysis.

Before we conduct experiments we need to set up all the parameters in a configuration file which will be read every time when SMOPD runs. The parameter setup needs to be done only once. The following Table 16 shows all parameters with their values. For each dynamic parameter, when we study its impacts on the algorithm's performance, we vary its values within a range and keep other dynamic parameters at their default values.

Table 16. Configuration File Parameters

Name	Meaning	Type	Value Range (for dynamic parameters)	Default Value
r	Physical read ratio threshold of a query	Dynamic	10%-30%	20%
f_n	The threshold of the percentage of queries that satisfies r	Dynamic	20%-50%	40%
f_t	Query frequency threshold: a query must occur at least f_t percent in the whole query set	Static	N/A	10%*Avg_Freq where Avg_Freg is Average query Frequency
c_α	Confidence interval	Static	N/A	1%
α	Confidence level	Static	N/A	95%

In order to test how the re-partitioning process performs, we need to partition the original database tables once at the beginning. We randomly generate a frequency for each query in the TPC-H benchmark. Totally we run the TPC-H queries 10,000 times; so the run time of each query can be calculated by using its frequency. Then we randomly select 60% of the query types from all TPC-H query types to run Filtered AutoClust once to get the first partitioning solution. The query set used to run Filtered AutoClust is shown in Table 17.

Both AutoClust and Filtered AutoClust are based on the closed item sets mining, and the resulting closed item sets is determined by the attribute usage matrix. If there is no change in the attribute usage matrix, then the resulting closed item sets will remain unchanged. So in order to reflect the general case, we have to use a new attribute usage matrix for each original TPC-H database table. Every time when we run SMOPD, we randomly select 60% of the TPC-H query types (this ensures that we have changes in query types) and generate a random frequency (this ensures that we have a change in query frequency which represents the percent of a query type in the whole query set) for each query selected. An example of a new query set is shown in Table 18. For each run of our test, we generate a new query set in the format of Table 18. After that we can start running SMOPD.

Table 17. Query Set Used in Running Filtered AutoClust

Number of Query Runs	Query Text
939	SELECT L_RETURNFLAG, L_LINESTATUS, SUM(L_QUANTITY) ….
915	SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT……
951	SELECT SUM(L_EXTENDEDPRICE*L_DISCOUNT) AS……
54	SELECT O_YEAR, SUM(CASE WHEN NATION = 'BRAZIL'……
6	SELECT NATION, O_YEAR, SUM(AMOUNT) AS SUM_PROFIT……
195	SELECT L_SHIPMODE, SUM(CASE WHEN O_ORDERPRIORITY …
115	SELECT C_COUNT, COUNT(*) AS CUSTDIST FROM (SELECT……
133	SELECT 100.00 * SUM(CASE WHEN P_TYPE LIKE 'PROMO%'……
979	SELECT P_BRAND, P_TYPE, P_SIZE, COUNT(DISTINCT ……
915	SELECT SUM(L_EXTENDEDPRICE) / 7.0 AS AVG_YEARLY……
331	SELECT C_NAME, C_CUSTKEY, O6.O_ORDERKEY, O_ORD ……
735	SELECT S_NAME, S_ADDRESS FROM SUPPLIER_2 S2, SUPP……

Table 18. A New Query Set Used in a Run of SMOPD

Average Physical Read Ratio	Query Frequency (%)	Number of Query Runs	Query Text
27%	3.2	292	SELECT S_ACCTBAL, S_NAME, N_NAME, P_PARTKEY……
50%	5.2	476	SELECT L_ORDERKEY, SUM(L_EXTENDEDPRICE……
50%	15.8	1453	SELECT N_NAME, SUM(L_EXTENDEDPRICE * (1 - L_DIS……
50%	0.7	62	SELECT SUPP_NATION, CUST_NATION, L_YEAR……
61%	3	279	SELECT O_YEAR, SUM(CASE WHEN NATION = 'BRAZIL'……
57%	17.3	1592	SELECT C_CUSTKEY, C_NAME, SUM(L_EXTENDEDPRICE * ……
47%	6.6	610	SELECT L_SHIPMODE, SUM(CASE WHEN O_ORDERPRIORITY
51%	3.1	287	SELECT 100.00 * SUM(CASE WHEN P_TYPE LIKE 'PROMO%'…
49%	4	371	SELECT P_BRAND, P_TYPE, P_SIZE, COUNT(DISTINCT ……
5%	8	739	SELECT SUM(L_EXTENDEDPRICE) / 7.0 AS AVG_YEARLY……
44%	3.9	362	SELECT C_NAME, C_CUSTKEY, O6.O_ORDERKEY, O_ORD……
50%	13.6	1252	SELECT SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT))……
53%	15.7	1447	SELECT S_NAME, S_ADDRESS FROM SUPPLIER_2 S2, SUPP……

Besides using different query sets to ensure the test quality, we need to introduce a cost model used in our experiments in order to understand the test results. The first cost is C_{old1} which represents the average estimated query cost for a particular database table based on the old database partitioning solution and the old query set. The second cost is C_{new1} which represents the average estimated query cost for a particular database table based on the old database partitioning solution and the new filtered query set, which is the query set that does not contain queries whose average physical read ratio less than r or whose frequency less than ft. The third cost is C_{new2} which represents the average estimated query cost of a particular database table based on the new database partitioning solution and the new unfiltered query set. The last cost is C_{old2} which represents the average estimated query cost of a particular database table based on the old database partitioning solution and the new unfiltered query set. We explain these costs further using the example query set in Table 18.

In Table 18, if we set r to 20% and ft to 10% of the average query frequency, then we can see that the query with 0.7% query frequency and the query with 5% average physical read ratio will be filtered out since 0.7% is less than 1/10 of the average query frequency and 5% is less than 20% (average physical read ratio threshold). So we get the new unfiltered query set Q_u as {1,3,5,6,7,10,12,14,15,16,17,18,19} where each number denotes a query number in the TPC-H benchmark, and the new filtered query set Q_f as {1,3,5,6,10,12,14,15,17,18,19}. Before we run SMOPD we already have a database partitioning solution for each TPC-H database table. This means that for the six TPC-H database tables, we already have the six corresponding partitioning solutions, P_{part_old}, $P_{customer\_old}$, $P_{partsupp\_old}$, $P_{supplier\_old}$, P_{orders_old} and $P_{lineitem\_old}$. If SMOPD wants to determine whether the PART database table should be re-partitioned, it will use Q_f and P_{part_old} to calculate a new average estimated query cost, C_{new1}, of the PART database table. If C_{new1} is bigger than C_{old1} then the re-partitioning process in

SMOPD will be triggered in which it calls the Filtered AutoClust algorithm to generate a new partitioning solution P_{part_new} for the PART database table. In order to compare the performance of the new partitioning solution with the old partitioning solution, we need to apply the whole query set on the two partitioning solutions. So we use Q_u and P_{part_new} to calculate C_{new2} and use Q_u and P_{part_old} to calculate C_{old2}. If C_{new2} is less than C_{old2} then it means SMOPD successfully finds a better partitioning solution. The meanings of four different costs are summarized in Table 19 below.

Table 19. Meaning Summaries of the Costs Used in SMOPD

	C_{old1}	C_{old2}	C_{new1}	C_{new2}
Partitions Used	old partitions	old partitions	old partitions	new partitions
Query Set Used	old query set	whole new query set	filtered new query set	whole new query set

SMOPD uses C_{old1} and C_{new1} to detect what the current performance trend is. If C_{new1} is not bigger than C_{old1}, then it means that the current performance is not degrading and thus we do not need to re-partition the corresponding database table. If C_{new1} is bigger than C_{old1}, then it means that the current performance is getting worse and thus re-partitioning is needed. After re-partitioning is done, we use C_{new2} and C_{old2} to test whether SMOPD finds a better re-partitioning solution successfully. If C_{new2} is less than $C_{old2,}$ then it means that SMOPD successfully finds a new partitioning solution to replace the old partitioning solution; otherwise the new partitioning solution should not be used to replace the old one.

5.2.2 Performance Metrics

We present the performance of our dynamic vertical partitioning algorithm on single computers, which is SMOPD, based on two performance metrics: the average estimated query cost of the old partitions based on the whole new query set and the average estimated query cost of the new partitions based on the whole new query set. Then we present the performance of our dynamic vertical partitioning algorithm on cluster computers, which is SMOPD-C, based on three performance metrics: the final estimated query cost of the old partitions based on the new query set, the final estimated query cost of the new partitions based on the new query set and the final estimated query cost improvement of the new partitions over the old partitions based on the new query set.

a. Performance Metrics for SMOPD

The Average Estimated Query Cost of the Old Partitions Based on the Whole New Query Set

We have explained this cost in our cost model in Section 5.2.1. This cost is used to measure the performance of the old partitions based on the whole new query set.

The Average Estimated Query Cost of the New Partitions Based on the Whole New Query Set

We have explained this cost in our cost model in Section 5.2.1, too. This cost is used to measure the performance of the new partitions based on the whole new query set. We compare these two costs in order to tell whether our re-partitioning process is successful. If the average estimated query cost of the new partitions is lower than the average query estimated cost of the old partitions then we say the re-partitioning process is triggered accurately and a better new partitioning solution is generated successfully; otherwise we say that the re-partitioning process is an inefficient action.

b. Performance Metrics for SMOPD-C

The Final Estimated Query Cost of the Old Partitions Based on the New Query Set

When we test our dynamic vertical partitioning algorithm on a cluster computer, we use the largest total estimated query cost among all computing nodes as the final estimated query cost of the cluster computer since all computing nodes are running in parallel. So this cost can be used to measure the performance of the old partitions on the cluster computer.

The Final Estimated Query Cost of the New Partitions Based on the New Query Set

The meaning of this cost is very similar to the cost we mentioned above. We use this cost to measure the performance of the new partitions on a cluster computer.

The Final Estimated Query Cost Improvement of the New Partitions over the Old Partitions Based on the New Query Set

We use this value to measure whether the performance of the new partitions is better than that of the old partitions on a cluster computer, and how much the improvement is. If this improvement is positive then we say that the new partitions can provide better performance than the old partitions on a cluster computer; otherwise the new partitions are useless.

5.2.3 Experiment Results

5.2.3.1 Performance Study of Our Dynamic Vertical Database Partitioning Algorithm on a Single Computer (SMOPD)

a. Impacts of the Threshold of The Percentage of Physical Read Mainly Queries (f_n)

In the first round of tests, we fix all parameters at their default values except for f_n. Table 20 shows the impacts of f_n on the performance of our algorithm. In this table we can see that for each original TPC-H database table and each value of *fn* there are five rows, among which four rows indicate the four different costs mentioned in Section 5.2.1 and one row ('New partitions') have the values defined as follows:

'Y': SMOPD decides that a re-partitioning action is needed and a partitioning solution with a better cost is found.

‘N’: SMOPD decides that a re-partitioning action is needed but a partitioning solution with a better cost is not found.

‘N/A’: SMOPD decides that a re-partitioning action is not needed.

Table 20. Impacts of *f_n*

f_n	PART		CUSTOMER		PARTSUPP		SUPPLIER		ORDERS		LINEITEM	
20 %	C_{old1}	286	C_{old1}	85	C_{old1}	585	C_{old1}	28	C_{old1}	4743	C_{old1}	18559
	C_{new1}	1011	C_{new1}	487	C_{new1}	1360	C_{new1}	39	C_{new1}	4164	C_{new1}	21613
	New partitions	Y	New partitions	N	New partitions	Y	New partitions	Y	New partitions	N/A	New partitions	Y
	C_{new2}	432	C_{new2}	487	C_{new2}	708	C_{new2}	31	C_{new2}	4164	C_{new2}	12509
	C_{old2}	916	C_{old2}	487	C_{old2}	1274	C_{old2}	39	C_{old2}	N/A	C_{old2}	21613
30 %	C_{old1}	286	C_{old1}	85	C_{old1}	585	C_{old1}	28	C_{old1}	4743	C_{old1}	18559
	C_{new1}	1005	C_{new1}	582	C_{new1}	1708	C_{new1}	35	C_{new1}	4322	C_{new1}	33994
	New partitions	Y	New partitions	Y	New partitions	Y	New partitions	Y	New partitions	N/A	New partitions	Y
	C_{new2}	499	C_{new2}	479	C_{new2}	708	C_{new2}	27	C_{new2}	3933	C_{new2}	18591
	C_{old2}	734	C_{old2}	576	C_{old2}	1291	C_{old2}	36	C_{old2}	N/A	C_{old2}	35338
40 %	C_{old1}	286	C_{old1}	85	C_{old1}	585	C_{old1}	28	C_{old1}	4743	C_{old1}	18559
	C_{new1}	714	C_{new1}	1212	C_{new1}	570	C_{new1}	32	C_{new1}	2362	C_{new1}	35019
	New partitions	Y	New partitions	Y	New partitions	N/A	New partitions	Y	New partitions	N/A	New partitions	Y
	C_{new2}	414	C_{new2}	886	C_{new2}	568	C_{new2}	28	C_{new2}	2362	C_{new2}	18593
	C_{old2}	677	C_{old2}	1195	C_{old2}	N/A	C_{old2}	33	C_{old2}	N/A	C_{old2}	34970
50 %	C_{old1}	286	C_{old1}	85	C_{old1}	585	C_{old1}	28	C_{old1}	4743	C_{old1}	18559
	C_{new1}	246	C_{new1}	848	C_{new1}	3084	C_{new1}	28	C_{new1}	4775	C_{new1}	18682
	New partitions	N/A	New partitions	Y	New partitions	Y	New partitions	N/A	New partitions	Y	New partitions	Y
	C_{new2}	N/A	C_{new2}	657	C_{new2}	708	C_{new2}	28	C_{new2}	3332	C_{new2}	14632
	C_{old2}	N/A	C_{old2}	848	C_{old2}	2748	C_{old2}	N/A	C_{old2}	4614	C_{old2}	19876

From Table 20 we can see that SMOPD can recognize which database tables are still having good partition performance and which ones are not. For instance, when f_n=50%, the new estimated query cost for the PART database table is 246 (the old partitioning solution of the PART database table is {[P_COMMENT,P_MFGR,P_RETAILPRICE,P_SIZE,P_PARTKEY],[P_BRAND,P_CONTAINER,P_PARTKEY],[P_NAME,P_PARTKEY],[P_TYPE,P_PARTKEY]}), which is less than the old estimated cost of 286, so the partitions’ performance of the PART database table is still good and SMOPD does not re-partition this database table. While the new estimated query cost for the LINEITEM database table is 18682 (the old partitioning solution of this database table is{[L_COMMITDATE,L_RECEIPTDATE, L_SUPPKEY,L_LINENUMBER,L_ORDERKEY],[L_PARTKEY,L_LINENUMBER, L_ORDERKEY],[L_DISCOUNT,L_EXTENDEDPRICE,L_LINESTATUS,L_QUAN TITY,L_RETURNFLAG,L_SHIPDATE,L_TAX,L_LINENUMBER,L_ORDERKEY], [L_SHIPINSTRUCT,L_LINENUMBER,L_ORDERKEY],[L_SHIPMODE,L_LINEN UMBER,L_ORDERKEY]}) which is bigger than the old estimated query cost of 18559, so the partitions' performance of the LINEITEM database table is getting worse and the algorithm re-partitions this database table. The new partitioning solution generated by our algorithm is {[L_LINENUMBER,L_ORDERKEY,L_COMMITDATE,L_DISCOUNT,L_EXTEND EDPRICE,L_PARTKEY,L_QUANTITY,L_RECEIPTDATE,L_RETURNFLAG,L_SH IPDATE,L_SHIPINSTRUCT,L_SHIPMODE,L_SUPPKEY,L_TAX],[L_LINENUMB ER,L_ORDERKEY,L_LINESTATUS,L_COMMENT]}. Once we apply the query set in Table 18 on this new partitioning solution, we can get the new average estimated query cost of 14632, which is better than the old cost. For those database tables the partitions' performance of which is getting worse, SMOPD automatically finds a better solution to replace the old one. Sometimes the new solution proposed by SMOPD is the same as the old partitioning solution (for instance, when f_n=20%, the new estimated cost for the CUSTOMER database table is 487 which is bigger than the old estimated cost of 85. This means that the partitions' performance is getting worse because of the change of the query workload, but this solution is still the best one that can be found by our algorithm comparing with other solutions). In such case, our algorithm performs an unnecessary run. From Table 20 we can see that the percentage of unnecessary runs is small, which happens only 1 out of 18 runs. SMOPD was able to correctly identify all cases when no repartitioning is needed (the “N/A” entries in Table 20). Overall our algorithm offers good performance improvement through its automatic decision on re-partitioning. The average performance improvement by SMOPD over all values of f_n is shown in Figure 13. From Figure 13 we can make the following conclusions:

1. SMOPD works well on all TPC-H database tables since the average performance improvement of all database tables is positive from 7% to 41%.
2. The best average performance improvement, which is 41%, is obtained for the PARTSUPP and LINEITEM database tables. The least performance improvement, which is 7%, is obtained for the ORDERS database table.
3. The average performance improvement for all database tables over all values of f_n is about 25%.

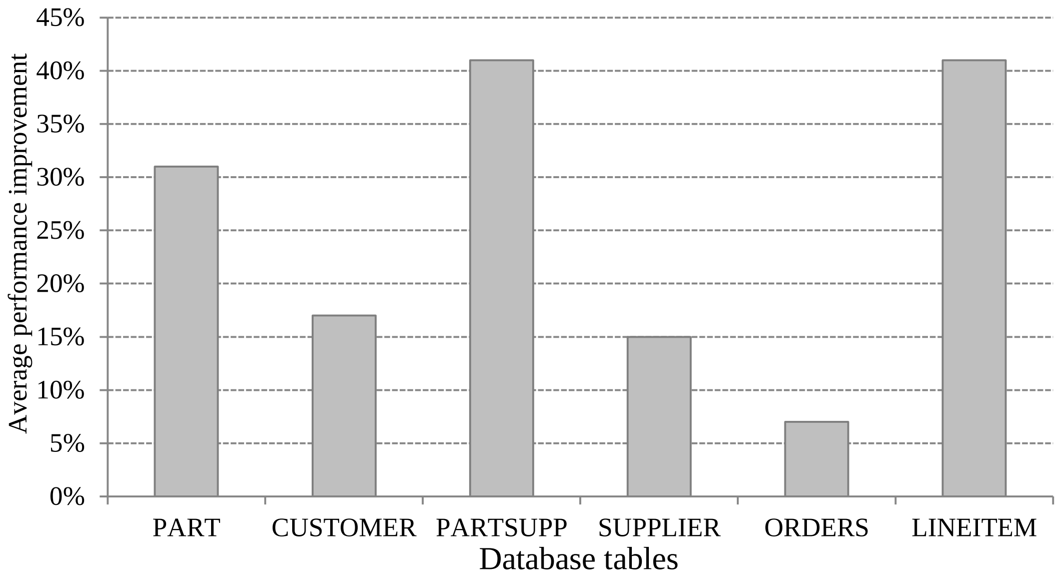

Figure 13. Average Performance Improvement Based on Estimated Query Cost over All Values of f_n for The TPC-H Database Tables

b. Impacts of Physical Read Ratio Threshold of a Query (*r*)

In the second round of tests, we fixed all parameters at their default values except for *r*. The impacts of *r* are shown in Table 21. From Table 21 we can see that when the query set is changing, i.e. when the system is monitoring the online query workload during different time periods, the estimated query cost always keeps changing for each database table. Those changes can be captured by our algorithm and used to analyze the system performance. As we see in Table 21, SMOPD can detect which database tables' partitions are still working well and which database tables' partitions are getting worse performance.

Table 21. Impacts of *r* on SMOPD Performance

r	PART		CUSTOMER		PARTSUPP		SUPPLIER		ORDERS		LINEITEM	
	C_{old1}	286	C_{old1}	85	C_{old1}	585	C_{old1}	28	C_{old1}	4743	C_{old1}	18559
	C_{new1}	246	C_{new1}	606	C_{new1}	3084	C_{new1}	28	C_{new1}	2262	C_{new1}	25151
10%	New partitions	N/A	New partitions	N	New partitions	Y	New partitions	N/A	New partitions	N/A	New partitions	Y
	C_{new2}	286	C_{new2}	606	C_{new2}	708	C_{new2}	28	C_{new2}	N/A	2187	14626
	C_{old2}	N/A	C_{old2}	606	C_{old2}	3084	C_{old2}	N/A	C_{old2}	N/A	C_{old2}	27646
	C_{old1}	286	C_{old1}	85	C_{old1}	585	C_{old1}	28	C_{old1}	4743	C_{old1}	18559
	C_{new1}	565	C_{new1}	1265	C_{new1}	573	C_{new1}	43	C_{new1}	4685	C_{new1}	20319
20%	New partitions	Y	New partitions	Y	New partitions	N/A	New partitions	N	New partitions	N/A	New partitions	Y
	C_{new2}	316	C_{new2}	617	C_{new2}	571	C_{new2}	40	C_{new2}	4280	C_{new2}	14626
	C_{old2}	505	C_{old2}	1265	C_{old2}	N/A	C_{old2}	40	C_{old2}	N/A	C_{old2}	22485
	C_{old1}	286	C_{old1}	85	C_{old1}	585	C_{old1}	28	C_{old1}	4743	C_{old1}	18559
	C_{new1}	284	C_{new1}	297	C_{new1}	1476	C_{new1}	28	C_{new1}	3634	C_{new1}	27287
30%	New partitions	N/A	New partitions	Y	New partitions	Y	New partitions	N/A	New partitions	N/A	New partitions	Y
	C_{new2}	583	C_{new2}	147	C_{new2}	708	C_{new2}	28	C_{new2}	3634	C_{new2}	9786
	C_{old2}	N/A	C_{old2}	297	C_{old2}	1131	C_{old2}	N/A	C_{old2}	N/A	C_{old2}	27444

Then, the algorithm finds a new partitioning solution once enough data is collected. We can see that our algorithm performs 18 cost comparisons and 10 re-partitioning actions. Out of 10 re-partitioning actions, 8 actions are correct, but the new solutions provided by our algorithm are always the best ones based on the current query set. For the 8 cases where SMOPD decides that re-partitioning is not needed, 7 cases are correct. The average performance improvement of our algorithm over all values of *r* tested for each TPC-H benchmark database table is shown in Figure 14. From Figure 14 we can make the following conclusions:

1. The new partitioning solutions of LINEITEM database table can provide the best average performance improvement comparing with the old partitioning solutions over all values of *r*. The new partitioning solutions of the PART database table provide the least average performance improvement comparing with the old partitioning solutions over all values of *r*.
2. There is no average performance improvement for the SUPPLIER and ORDERS database tables since re-partitioning is not triggered for these two database tables.
3. The average performance improvement for all database tables over all values of *r* is about 22%.

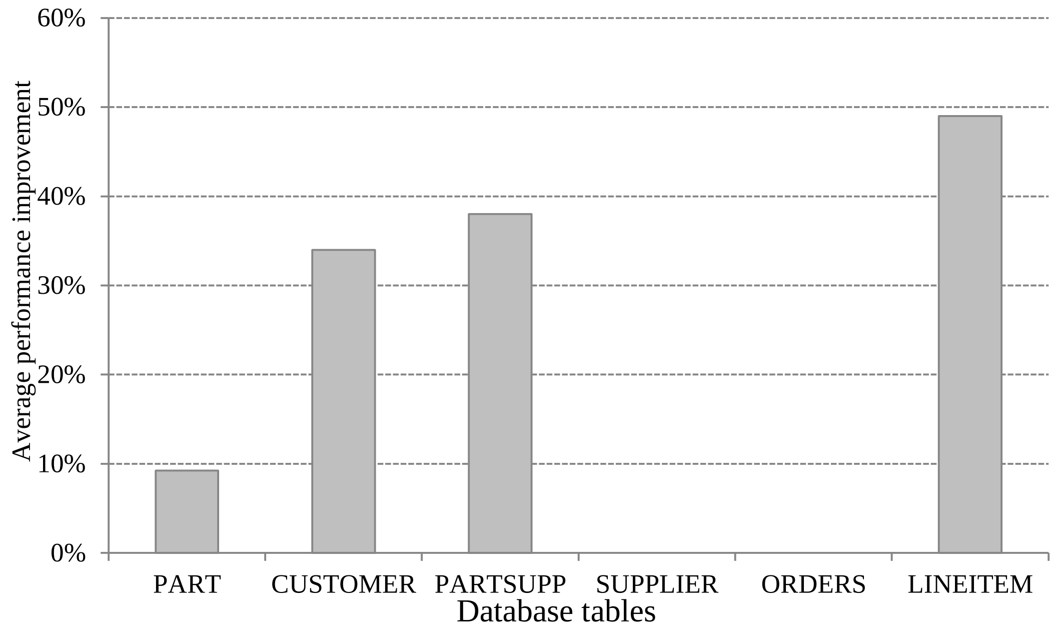

Figure 14. Average Performance Improvement Based on Estimated Query Cost over All Values of *r* for The TPC-H Database Tables

5.2.3.2 Performance Study of Our Dynamic Vertical Database Partitioning Algorithm on Cluster Computers (SMOPD-C)

a. Impacts of Number of Computing Nodes

When we study the impact of number of computing nodes we need to fix the database table. We use the PART database table as the test database table. We first measure the final estimated query cost for the old partitions based on the new query set and then measure the final estimated query cost for the new partitions based on the new query set. If the final cost of the new partitions is less than the final cost of the old partitions with the same number of computing nodes, then we say that the re-partitioning action was successfully done; otherwise the re-partitioning action is an inefficient run. The test results are shown in Figure 15. From Figure 15 we can make the following conclusions:

1. When the number of computing nodes increases, the final estimated query costs of both the old partitions and the new partitions decrease for the PART database table. That is because all computing nodes are running in parallel and less work will be processed by each computing node if more computing nodes are available.
2. For the same number of computing nodes, the final estimated query cost of the new partitions is always less than the final estimated query cost of the old partitions. This means that our algorithm is always able to find out when re-partitioning is needed and the re-partitioning results always provide better query response time than the old partitions.

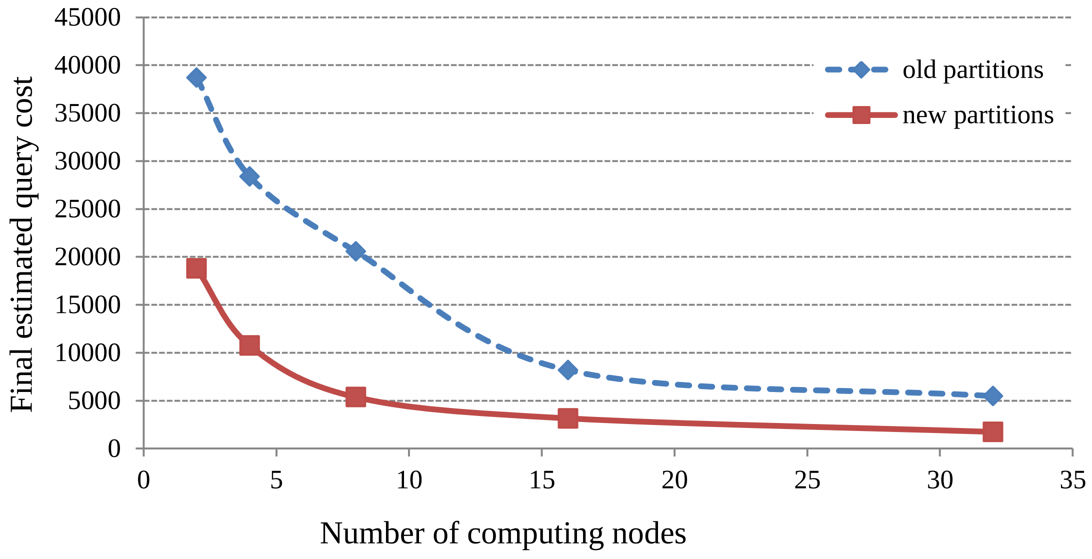

Figure 15. Final Estimated Query Cost of the Old and New Partitions of the PART Database Table on Cluster Computers Based on Different Numbers of Computing Nodes

b. Impacts of Database Table Size

When we study the impact of the database table size, we need to fix the number of nodes. We set the number of computing nodes to 8. From Figure 16 we can make the following conclusions:

1. For the ORDERS database table, the performance of more than half of the computing nodes is still good, so re-partitioning is not triggered for this database table, i.e. the new partitions are the same as the old partitions. So we can see there is no final estimated cost improvement for the ORDERS database table.
2. The new partitioning solution of the PART database table gives the best performance improvement comparing with its old partitioning solution. The new partitioning solution of the SUPPLIER database table gives the least performance improvement comparing with its old partitioning solution.

3. The final estimated query cost improvement is from 8% to 74% and the average final estimated query cost improvement is 38%. We can see that our algorithm works well on cluster computers.

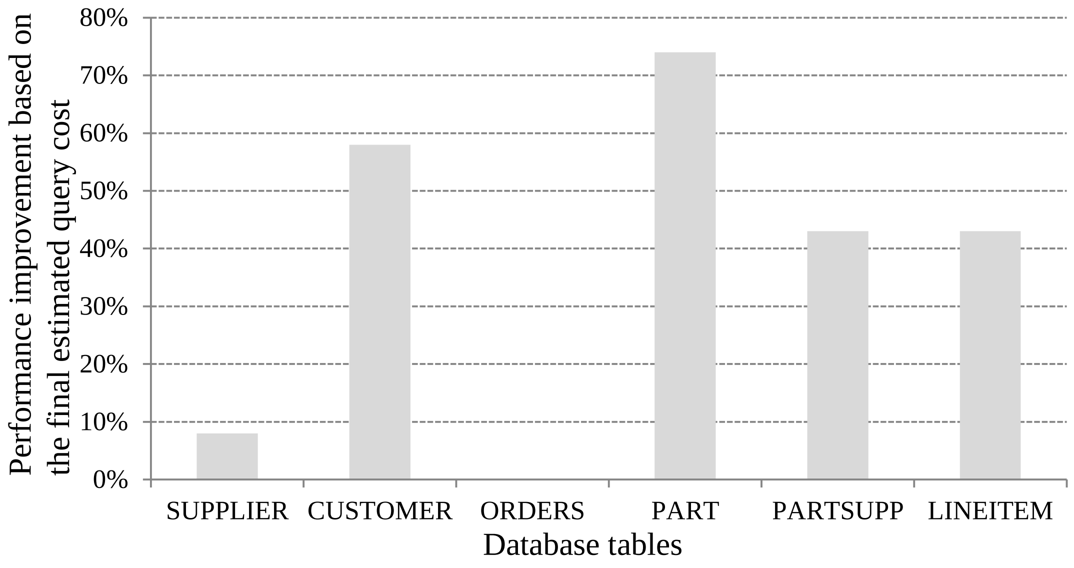

Figure 16. Performance Improvement Based on Final Estimated Query Cost of the New Partitioning Solutions over the Old Partitioning Solutions for Different TPC-H Database Tables

Chapter 6 Conclusions and Future Work

In this research we have proposed a dynamic vertical database partitioning algorithm, SMOPD-C, which makes use of two novel algorithms, Filtered AutoClust and SMOPD, developed in this thesis, Filtered AutoClust is designed for statically clustering the database attributes using a fixed query workload on both single and cluster computers. It mines closed item sets in order to cluster attributes that are frequently accessed together in queries into the same partition, then it implements the partitions on computing nodes, and routes each incoming query to the computing node containing the partition that gives the best estimated query cost for the query execution. The second algorithm, SMOPD, is designed for dynamically clustering the database attributes on a single computer without using any fixed query workload. It reads the statistic information from the system views to collect physical read mainly queries from those queries that have been processed since the last partitioning to build a new query set. Then it reevaluates the average estimated query cost for each database table using the new query set to determine the current performance trend. If the current performance trend is degrading, then Filtered AutoClust is called to re-partition this database table. SMOPD-C uses both SMOPD and Filtered AutoClust to vertically and dynamically partition the database tables stored in a cluster computer. In SMOPD-C, each computing node runs SMOPD to monitor the performance of queries accessing its database tables. When SMOPD-C determines that for a particular database table, the query performance is decreasing on more than half of the computing nodes in the cluster computer, it invokes Filtered AutoClust to re-partition this database table.

We have performed extensive experiments in order to study the performance of Filtered AutoClust, SMOPD and SMOPD-C using TPC-H benchmark as well as a synthetic database and synthetic queries. We have compared our Filtered AutoClust algorithm with other existing vertical partitioning algorithms, No Partition (i.e. the baseline case when the database tables are not partitioned), AutoClust [19] and Eltayeb [5], in order to measure both the partitioning solutions' performance in terms of average estimated query cost and the algorithm's running time. The new partitioning solutions generated by our algorithm show a great performance improvement compared to the existing work algorithms. We have studied our SMOPD and SMOPD-C algorithms by comparing the performance of the current partitioning solution with the performance of the new partitioning solution from different angles. The results show that SMOPD and SMOPD-C are capable of performing database re-partitioning dynamically with high accuracy to provide better average estimated query cost than the current partitioning configuration. A summary of our performance evaluation results is presented in the following sections.

6.1 Summary of the Performance Evaluation Results

6.1.1 Summary of the Performance Results for Filtered AutoClust

Filtered AutoClust is a semi-automatic or static vertical database partitioning algorithm based on closed item sets mining. It requires one user-defined parameter: query frequency threshold (f_i). In the thesis we used the average query frequency as the default value. By changing this value people can decide which queries are less important and should be filtered out. Below we summarize the results we have obtained so far for Filtered AutoClust.

1. Filtered AutoClust can provide the same suitable partitioning solutions as AutoClust according to our experiments. These partitioning solutions have better performance than Eltayeb and No Partition based on the estimated query cost.
2. Filtered AutoClust has much better running time than AutoClust. Database performance can be greatly benefited by deploying this algorithm on cluster computers. When the number of computing nodes increases the running time of Filtered AutoClust decreases.
3. The running time of Filtered AutoClust is affected significantly by the number of query types. From the experiments we performed using a synthetic database, the running time of Filtered AutoClust is getting bigger when there are more query types.

6.1.2 Summary of the Performance Results for SMOPD

SMOPD is a dynamic vertical database partitioning algorithm. It requires five user-defined parameters: 1) physical read ratio threshold of a query (r) (by changing this value, people can decide whether a query is a physical read mainly query or a logical read mainly query); 2) the ratio threshold of number of queries that satisfies r (f_n) (by changing this value, people can decide how many queries have to be collected in order to do re-partitioning checking; 3) query frequency threshold (f_t) (by changing this value people can decide what queries are outlier queries); 4) confidence interval (a) and 5) confidence level (c_a) (by changing these two values a and c_a, people can decide how many queries have to be collected so that there are enough physical read mainly queries for re-partitioning. Below we summarize the results we have obtained so far for SMOPD.

1. SMOPD is able to dynamically monitor the performance trend for the current partitions of each database table and judge whether or not re-partitioning is needed for the database table.
2. SMOPD is capable of performing database re-partitioning with high accuracy to provide better estimated query cost than the current partitioning solutions.

6.1.3 Summary of the Performance Results for SMOPD-C

SMOPD-C makes use of Filtered AutoClust and SMOPD to perform dynamic vertical database partitioning for databases stored in cluster computers; so the parameters used by Filtered AutoClust and SMOPD are used by SMOPD-C. Below we summarize the results we have obtained so far for SMOPD-C.

1. When SMOPD-C determines that re-partitioning is needed for a database table on a cluster computer, its new partitioning solutions generated always yield better estimated query cost than the current partitioning solutions on the same cluster computer.
2. When the number of computing nodes increases, the final estimated query cost of the new partitioning solution generated by SMOPD-C decreases. If the number of computing nodes is big enough, the performance of the old partitioning solutions is close to the performance of the new partitioning solutions.

6.2 Future Work

The SMOPD-C algorithm fills the gap in that no existing algorithms can dynamically partition database tables on a cluster computer. However, it has its own disadvantage as we described in Chapter 4. When a re-partitioning task is performed, all computing nodes have to be involved including those nodes that still work well. This is a computing resource waste and we do need a better way to perform the re-partitioning task. Also in this thesis, we tested our algorithms, SMOPD and SMOPD-C, using the TPC-H benchmark, but not real datasets and queries.

In the future research, we would like to develop a new algorithm which can dynamically monitor each computing node separately so that the re-partitioning process only happens on the computing nodes with bad performance. In this way, the algorithm can avoid unnecessary re-partitioning work. Also we would like to use real data and queries to test our algorithms.

References

[1] Codd, E.F., A Relational Model of Data for Large Shared Data Banks, Communications of the ACM Volume 13 Issue 6, June 1970, pages 377–387.

[2] Schnaitter, K., and Polyzotis, N., Semi-Automatic Index Tuning: Keeping DBAs in the Loop, Proceedings of Very Large Data Bases (PVLDB), 5(5):478–489, 2012.

[3] Schnaitter, K., Abiteboul, S., Milo, T., and Polyzotis, N., On-line Index Selection for Shifting Workloads. In International Workshop on Self-Managing Database Systems, pages 459–468, 2007.

[4] Rodd, S. F., and Kulkrani, U. P., Adaptive Tuning Algorithm for Performance tuning of Database Management System, International Journal of Computer Science and Information Security, Vol. 8, No. 1, April 2010.

[5] Jindal, A., and Dittrich, J., Relax and Let the Database do the Partitioning Online. In Business Intelligence for Real Time Enterprise (BIRTE), September 2011.

[6] Duan S., Thummala V., and Babu S., Tuning Database Configuration Parameters with Ituned, Proceedings of Very Large Data Bases (PVLDB), vol. 2, pp. 1246–1257, August 2009.

[7] Agrawal, S., Narasayya, V., and Yang, B., Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design, Special Interest Group on Mamagement od Data (SIGMOD), June 2004.

[8] Rodriguez, L. and Li, X., A Dynamic Vertical Partitioning Approach for Distributed Database System, Systems, Man, and Cybernetics (SMC), IEEE International Conference 2011.

[9] http://www.tpc.org.

[10] Pasquier, N., Bastidem, Y., Taouil, R. and Lakhal, L., Efficient Mining of Association Rules Using Closed Item set Lattices, Information Systems, Vol. 24, No. 1, 1999.

[11] Abuelyaman, E., S., An Optimized Scheme for Vertical Partitioning of a Distributed Database, International Journal of Computer Science and Network Security (IJCSNS), Vol.8, No.1, 2008.

[12] Horowitz, E. and Sahni, S., Fundamentals of Computer Algorithms, Rockville, MD: Computer Science Press, 1978.

[13] Ghandeharizadeh S. and DeWitt D. J., Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In Very Large Data Bases (VLDB), pages 481–492, 1990.

[14] Navathe, S., Ceri, S., Wierhold, G. and Dou, J., Vertical Partitioning Algorithms for Database Design, ACM Transactions on Database Systems, Vol. 9, No. 4, December 1984.

[15] Rao, J., Zhang, C., Megiddo, N., and Lohman, G. M., Automating Physical Database Design in a Parallel Database. In Special Interest Group on Mamagement od Data (SIGMOD), page 558-569, 2002.

[16] Baker, M. Cluster Computing at a Glance, Chapter 1, High Performance Cluster Computing: Architectures and Systems, Vol. 1, Prentice Hall, 1st edition, Editor Buyya, R., May 1999.

[17] Kurmann C., Rauch F. and Stricker T. M., Cost/Performance Tradeoffs in Network Interconnects for Clusters of Commodity PCs. ETH Z ̈urich, 2003.

[18] Wu, S., Li, F., Mehrotra, S., and Ooi, B. C., Query Optimization for Massively Parallel Data Processing. In Symposium on Cloud Computing (SOCC), 2011.

[19] Guinepain, S. and Gruenwald, L., Using Cluster Computing to support Automatic and Dynamic Database Clustering, International Workshop on Automatic Performance Tuning (IWAPT), 2008.

[20] DeWitt, D. and Gray J., Parallel Database Systems: The Future of High Performance Database Systems, Communications of ACM, Volume 35 Issue 6, June 1992.

[21] McCormick, W. T. Schweitzer P.J., and White T.W., Problem Decomposition and Data Reorganization by A Clustering Technique, Operation Research, Vol. 20, No. 5, September 1972.

[22] Wesley W. Chu and I. Ieong, A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems, IEEE Transactions on Software Engineering, Vol. 19, No. 8, August 1993.

[23] Navathe, S. and Ra M., Vertical Partitioning for Database Design: A Graph Algorithm, ACM Special Interest Group on Mamagement od Data (SIGMOD) International Conference on Management of Data, 1989.

[24] Papadomanolakis, S., Dash, D. and Ailamaki, A., Efficient Use of the Query Optimizer for Automated Physical Design, Proceedings of the 33rd International Conference Very Large Data Bases (VLDB), September 2007.

[25] Li, L., and Gruenwald, L., Autonomous Database Partitioning Using Data Mining on Single Computers and Cluster Computers, International Database Engineering & Applications Symposium (IDEAS), August 2012.

[26] Li, L., and Gruenwald, L., Self-Managing Online Partitioner for Databases (SMOPD) – A Vertical Database Partitioning System with a Fully Automatic Online Approach, International Database Engineering & Applications Symposium (IDEAS), October 2013.

[27] Amossen R, Vertical Partitioning of Relational OLTP Databases using Integer Programming, Data Engineering Workshops (ICDEW) of IEEE 5th International Conference on Self Managing Database Systems (SMDB), 2010.

[28] Akal, F., Bohm, K., and Schek, H. –J. OLAP Query Evaluation in a database Cluster: A performance Study on Intra-query Parallelism, The 6th East European Conference on Advances in Database and Information Systems. London, UK, pp. 218-231, 2002.

[29] Das, S., Agrawal, D., and Abbadi, A. E. ElasTraS: An Elastic Transactional Data Store in the Cloud. In USENIX Hot Cloud, June 2009.

[30] Pukdesree S., Lacharoj V., and Sirisang P., Performance Evaluation of Distributed Database on PC Cluster Computers, World Congress on Engineering and Computer Science (WCECS) 2010, October , 2010.

[31] Graefe G., The Cascades Framework for Query Optimization. IEEE Data Eng. Bull., 18(3): 19-29, 1995.

[32] Graefe G. and McKenna W. J., The Volcano Optimizer Generator: Extensibility and Efficient Search. In Data Engineering Workshops (ICDEW), 1993.

[33] http://oscer.ou.edu

[34] Berthuet R., Cours de Statistiques, CUST, Clermont-Ferrand, France, 1994.

[35] Frank, E. G., Procedures for Detecting Outlying Observations in Samples, Technometrics, Vol. 11, No. 1, pp. 1-21, February 1969.

[36] Buyya, R., High Performance Cluster Computing: Architectures and Systems, Vol. 1, Chapter 1, Prentice Hall, 1st edition, May 1999.

[37] Garcia-Alvarado, C., Raghavan, V., Narayanan, S. and Waas, F.M., Automatic Data Placement in MPP Databases, Data Engineering Workshops (ICDEW) of IEEE 7th International Conference on Self Managing Database Systems (SMDB), 2012.