SMOPD-C: An Autonomous Vertical Partitioning Technique for Distributed Databases on Cluster Computers

Liangzhe Li
School of Computer Science
University of Oklahoma
Norman, USA
lzli@ou.edu

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, USA
ggruenwald@ou.edu

Abstract

Distributed databases on 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 paper introduces a dynamic algorithm, SMOPD-C, that can autonomously partition a distributed database vertically on cluster computers, determine when a database re-partitioning is needed, and re-partition the database accordingly. The paper then presents comprehensive experiments that were conducted to study the performance of SMOPD-C using the TPC-H benchmark on a cluster computer. The experiment results show that SMOPD-C is capable of performing database re-partitioning dynamically with high accuracy to provide better query cost than the current partitioning configuration.

Keywords: vertical partitioning; cluster computer; query optimizer; physical read mainly queries

1. Introduction

For both centralized databases stored in a single computer and distributed databases stored in cluster computers, self-managing database design is becoming more and more important to many applications. This is because with the structure of data getting more and more complex, the size of data getting bigger and bigger, and the velocity of data is getting faster and faster, it is not feasible to ask users to manage all the settings in a database by themselves. Self-managing database systems are thus needed. Self-managing database algorithms cover multiple database research areas including self-managing database indexing ([2] and [3]), self-managing database caching [4], self-managing database partitioning [5], self-tuning database parameters [6], etc. In this paper we present a novel technique, SMOPD-C, for self-managing database partitioning of distributed databases stored in cluster computers.

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 database partitioning techniques which can considerably improve query response time when physical database design is performed [7]. Today, most database systems support horizontal partitioning [8]. The common horizontal partitioning approaches that are used by most database developers are range partitioning and hash partitioning [12]; 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 distributed databases stored in cluster computers.

Since the first vertical partitioning algorithm [17] was developed in early 1970s, a number of different types of vertical partitioning algorithms have been proposed, such as [11], [13], [18], [19], [20] and [23]. One common feature among these algorithms is that they use a fixed workload to generate the partitioning solution for database tables. If users want to re-partition the database tables, they have to review the query log files to select recent queries which they believe important in order to perform the re-partitioning task. So these algorithms are called static vertical partitioning algorithms. It is time-consuming to review the query log files. If the DBA does not have solid physical database design knowledge or is not careful enough, the workload used to re-partition the database tables might not be accurate and will even generate a worse partitioning solution. To solve this problem, some dynamic vertical partitioning algorithms ([5] and [23]) have been proposed in recent years.

A dynamic vertical partitioning algorithm has the ability to monitor the incoming queries and automatically re-partition the database tables when their partitions do not perform well any more. The algorithms [5] and [23] are dynamic and designed for a centralized database stored in a single computer. However, in recent years, cluster computers have been widely adopted by many organizations to handle their distributed databases. A cluster computer consists of a collection of interconnected stand-alone computers working together as a single, integrated computing resource [15]. A network composed of many cluster computers can provide excellent computing performance for distributed databases with low cost. So how to optimize the physical database design on a cluster computer system becomes extremely important, and database partitioning on a cluster system is one of those important research topics in the physical database design area. Unfortunately, the algorithms presented in [5] and [23] do not give any clue on that. In order to fill the gap, a novel vertical partitioning algorithm called SMOPD-C is presented in this paper. It is a dynamic algorithm providing multiple vertical partitioning solutions and designed for a distributed database system running on a cluster computer. The paper also presents the results of the experiments that are conducted to study the performance of SMOPD-C using the TPC-H benchmark [9].

The rest of the paper is organized as follows. Section 2 discusses the related work. Sections 3 and 4 present SMOPD-C and its experimental results, respectively. Finally Section 5 concludes the paper with future research.

2. Related Work

A good number of algorithms that can vertically partition databases on a single computer have been proposed in the literature. Some of them are static ([11], [13], [16], [17], [18] and [19]) and some of them are dynamic ([1], [5] and [22]). A few algorithms exist to partition distributed databases on cluster computers ([21], [24], [26], [14] and [16]).

In [24], a database schema level partitioning algorithm, called ElasTras, was introduced. 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 tables. 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.

In [26], the authors proposed an algorithm called FINDER that aims to find the optimal data distribution policy for a set of database tables. 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 = {T1, …,Tt}, this algorithm can find the distribution policy D = {X1, …, Xt} where Xi is a set of attributes (usually they are the primary keys) and Ti is distributed based on Xi. The tuples of a database table will be assigned to different segments according to the hash value of Xi. This algorithm is used to statically and horizontally partition the database tables.

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 [14] is a partitioning algorithm used only on OLTP applications. In [14] 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.

In [16] and [21], a table level partitioning algorithm, called AutoClust, was presented. This algorithm uses query optimizer to generate partitioning solutions. According to the authors' ideas, multiple partitioning solutions are selected from the candidate partitioning solution pool. 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. There are 7 steps in the AutoClust algorithm when it works on cluster computers. 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 by the same set of queries. We want to keep such attributes in the same 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 vertical database partitioning solution. In Step 5, the solutions are submitted to the query optimizer of the database system in order to determine what solutions are better. In Step 6, the suitable solutions are distributed to different computing nodes so that each node carries a solution. Finally, a query routing table, which maps each query type to a unique computing node that has the best partitioning solution for that specific query type, is built so that a query can always be routed to a node with the best partitioning solution (the detail information of how to construct the query routing table can be found in [21]). 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. So this algorithm is a static algorithm, too.

To the best of our knowledge, while there are static and dynamic database vertical portioning algorithms designed for databases on a single computer and some static algorithms for databases on cluster computers. There exists no partitioning algorithm designed to vertically and dynamically partition database tables on cluster computers. In the next section we present our SMOPD-C algorithm, the first vertical partitioning algorithm that can dynamically partition distributed databases stored in a cluster computer.

3. SMOPD-C

In SMOPD-C, we treat all computing nodes as a whole system together. 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. We use the estimated query cost to represent the query response time. For each computing node, if a database table's current partitions' average estimated query cost of the current query set is larger than the average estimated query cost of the old query set that is used to generate the current partitions for this database table, then we say that 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 majority of the computing nodes is not good, then we say the average performance of the entire cluster computer is not good. We measure the average performance of all computing nodes and decide whether re-partitioning is needed or not. If the average performance is bad, the control node will trigger Filtered AutoClust, which is an advanced version of AutoClust [16], to repartition the corresponding database table.

From the description above, we can see there are two major components in SMOPD-C: the current partitions' performance checking component and the database re-partitioning component. The current partitions' performance checking component is deployed on every computing node. The major function of this component is to monitor the performance trend of the current partitions for each node and make the correct re-partitioning decisions. The database re-partition component is deployed on the control node. The major function of this component is to invoke the Filtered AutoClust algorithm to statically partition distributed database tables on a cluster computer using the table set and query set collected by the current partitions' performance checking component.

In the following two subsections we will describe how each component works using an example. In the example we deploy the TPC-H benchmark database [9] on a cluster computer with 8 computing nodes. The TPC-H database benchmark consists of twenty two query types and eight database tables, ORDERS, CUSTOMER, LINEITEM, PART, SUPPLIER, PARTSUPP, REGION and NATION; however, in our research we use only the first six tables as the last two tables are too small to benefit from partitioning. We show how our SMOPD-C algorithm works on the PART database table which has the following schema PART (P_COMMENT, P_PARTKEY, P_BRAND, P_CONTAINER, P_SIZE, P_TYPE, P_NAME, P_MFGR, P_RETAILPRICE). Before we run this algorithm, we need to partition the PART database table once. The query set we use is randomly selected from the TPC-H query types. The partitioning solutions we got are S1 and S2 as listed below.

Candidate solution S1: [{P_COMMENT, P_PARTKEY}, {P_MFGR, P_PARTKEY}, {P_PARTKEY, _RETAILPRICE}, {P_BRAND, P_PARTKEY, P_SIZE,P_TYPE}, {P_CONTAINER, P_PARTKEY}, {P_NAME, P_PARTKEY}].

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}].

Table 1. Estimated query costs of two partitioning solutions on different nodes

Node	Candidate Solution	Est. cost (Query 8)	Est. cost (Query 9)	Est. cost (Query 14)	Est. cost (Query 15)	Est. cost (Query 16)	Est. cost (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

Table 2. Query set used to re-partition database tables

Query No.	Freq. %	Average Physical Read Ratio	Query 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_Q….
3	9.3%	50%	23	SELECT L_ORDERKEY, SUM(L_EXTENDEDPRICE……
4	9.9%	51%	24	SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER……
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_……
10	3.7%	50%	9	SELECT C_CUSTKEY, C_NAME, SUM(L_EXTENDED……
11	9.7%	47%	24	SELECT PS_PARTKEY, SUM(PS_SUPPLYCOST * PS_AV……
13	2.9%	51%	7	SELECT C_COUNT, COUNT(*) AS CUSTDIST FROM ……
15	3.9%	5%	10	SELECT P_BRAND, P_TYPE, P_SIZE, COUNT(DISTINCT ……

The way of deploying the two partitioning solutions on the cluster computer is shown in Table 1. In Table 1 we can see the estimated cost of each query for each computing node.

Once the current partitions are constructed, SMOPD-C can be started. Below are the details of the two components of this algorithm.

3.1. Current Partitions' Performance Checking Component

This component running on each computing node monitors the most recent queries processed by that computing node. This process can be done by reading the system views which contain the query information (for example, in SQL Server [28], the system view is SYS.DM_EXEC_QUERY_STATS; in Oracle [29] the system view is V_$SQLAREA). Then this component will calculate how many queries it needs to collect so that it can ensure that there are enough physical read mainly queries to be analyzed. Physical read mainly queries are those queries which access most data from hard disk rather than main memory [22]. Once the component collects enough physical read mainly queries, it will use this query set to estimate the performance of the current partitions of each database table on each node. If the performance of a database table's partitions on the majority of the computing nodes is not good, this database table will be passed to the database re-partition component on the control node for re-partitioning preparation. The process can be summarized in the following steps using the example mentioned at the beginning of Section 3.

Step 1: estimate the number of queries N that needs to be collected using the formula $N = z_\alpha^2 * \frac{fn(1-fn)}{c_\alpha^2}$ [22], where c_α is the precision, z_a is the function of confidence level α, *fn* is the ratio threshold of number of queries that satisfies physical read ratio threshold of a query. We randomly select 60% of the query types from the TPC-H benchmark query type set with a random frequency for each query type. So we get the query type set as shown in Table 2.

Step 2: collect queries' information (query id, physical read ratio, logical read ratio, count, query context) from the system views of the DBMS on each computing node until N queries are collected.

Step 3: simplify and filter queries, and put the result queries into a set *FQS*. Simplifying queries rewrites each query into a simple format which contains only the original table name followed by the attributes accessed by the query, and filtering queries removes logical mainly read queries and outlier queries from the query set as logical read mainly queries and outlier queries will not be impacted by database partitioning [22].

Step 4: evaluate the performance of the current partitions for each database table on each computing node. In this step the new average estimated query cost for the current partitions on each computing node is calculated and shown in Table 3. We can see that the current partitions' performance is getting worse due to the query set changes on node 1, node 3, node 5, node 7 and node 8.

Step 5: evaluate the average performance of each database table on all nodes and put the database table whose current partitions' average performance on more than half of the nodes is not good any more in a set *T* on the control node.

From Table 3 we can see that for the PART database table, 62.5% of the computing nodes do not perform well using the current partitions and should be re-partitioned. The algorithm of this component is shown in Figure 1.

Table 3. Different cost results of the partitioning solutions on different nodes

Node	Candidate Solution	New Average Cost	Current 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

```
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. Precision- $c_\alpha$
    5. Confidence level- α
Output:
database table which needs re-partitioning and the physical read
mainly query set
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 on each node 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: simplify and filter queries, put the result queries into a set
FQS
Step 4: evaluate the performance of current partitions for each
database table on each node.
5    let $p_i$ to be the performance of the current partitions of a
     database table on node i
6    set $p_i$ to 1 if the performance is still good, otherwise set it to 0
Step 5: evaluate the average performance of each database table
on all nodes
7    if $\frac{\sum_{i=1}^{n} p_i}{n} \geq 50\%$, where n is the total number of computing
     nodes
8      invoke Filtered AutoClust of the database re-partitioning
       component on the control node
9    else
10     goto step 2
11   end if
```

Figure 1. Algorithm for the current partitions' performance checking component

3.2. Database Re-Partitioning Component

This component is used to re-partition the database tables that are recorded due to bad performance of their current partitions. The Filtered AutoClust algorithm is invoked when the performance of a database table's partitions on the majority of the computing nodes is not good. Filtered AutoClust is an improvement of AutoClust [21], a static vertical partitioning algorithm based on closed item set mining which we have described in Section 2. Filtered AutoClust improves AutoClust's execution time by removing unnecessary closed item sets, removing over-partitioned candidate solutions and distributing the remaining candidate solutions to the query optimizers on the computing nodes so that the query optimizers can estimate the query costs for the candidate solutions in parallel. Filtered AutoClust consists of the following steps:

Step 1: generate the attributes usage matrix.

Step 2: generate the set of attribute set *As* and the corresponding frequency set *Fs* for each query based on the attribute usage matrix.

Step 3: mine the closed item sets from the matrix generated in Step 1.

Step 4: filter the CIS set based on *As* and *Fs* in order to remove unnecessary CIS.

4.1. Remove the attribute set from *As* whose frequency is below the average frequency.

4.2. For each attribute set *Asi* in the new *As* remove its subset in the CIS set to form the new CIS set.

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

Step 5: 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.

Step 6: generate the execution tree where each leaf represents a candidate vertical partitioning solution.

Step 7: 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 8: 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 9: 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 10: 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.

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

Figure 2. The Filtered AutoClust algorithm on cluster computers

The pseudo code of the Filtered AutoClust algorithm is shown in Figure 2.

Once we re-partition the PART database table, we get the following partitioning solutions S1_new and S2_new as the 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}]

These two new solutions will be implemented on 8 computing nodes in a round robin order, i.e. S1_new will be implemented on node 1, node 3, node 5 and node 7; S2_new will be implemented on node 2, node 4, node 6 and node 8. The query routing table will be reconstructed according to the new partitioning solutions.

4. Experimental Performance Studies

In this section we present our experiment results comparing the performance of the new partitions generated by SMOPD-C and that of the current partitions. Our test was conducted on the cluster computer OSCER BOOMER [25] at the University of Oklahoma. We use the TPC-H benchmark [9] database tables and queries for our test, and all the TPC-H database tables are fully replicated on the Oracle database system on the cluster computer. We set the five parameters in SMOPD-C, *a* (confidence level), c_a (precision), *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.

4.1. Impact of Number of Computing Nodes

In this experiment we study the impact of number of computing nodes. We use the PART database table as the test database table. We first measure the final estimated query cost for the current partitions based on the new query set and then measure the average estimated query cost for the new partitions based on the new query set. If the average estimated query cost of the new partitions is less than the average estimated query cost of the current 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 unnecessary run. The test results presented in Figure 3 show that when the number of computing nodes increases, the average estimated query costs of both the current partitions and the new partitions decrease for the PART database table. This 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. For the same number of computing nodes, the average estimated query cost of the new partitions is always less than the average estimated query cost of the current 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 current partitions.

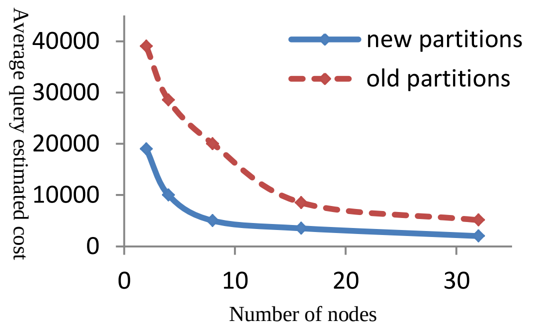

Figure 3. Impacts of the number of computing nodes for the PART database table

4.2. Impact of Different Table Sizes

In this experiment we study the impact of the database table size by measuring the performance for the six largest tables in the TPC-H benchmark. We set the number of computing nodes to 8. From Figure 4 we can see that there is no performance improvement for the ORDERS database table. This is because the performance of more than half of the computing nodes is still good even the query set has been changed for the ORDERS database table, so re-partitioning is not triggered for this database table; the new partitioning solution of the PART database table gives the best performance improvement comparing with its current partitioning solution; the average estimated query cost improvement for an individual table is from 8% to 74%; and the average improvement for all database tables together is 38%. From the test results we can see that our algorithm works well on cluster computers.

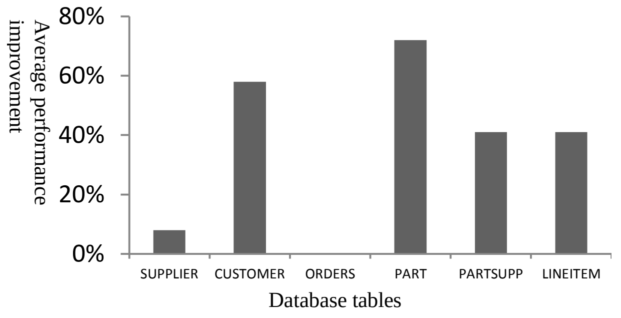

Figure 4. Impacts of different database table sizes on 8 computing nodes

5. Conclusions

In this paper, we presented an efficient algorithm, SMOPD-C, which can dynamically partition a distributed database vertically on cluster computers. SMOPD-C keeps monitoring the most recent queries processed by each computing node and reading the related query information (query id, query context, physical read ratio, logical read ratio, and query count) from the system views until enough physical read queries are collected. Then SMOPD-C uses the query set collected, passes it to the query optimizer to calculate the current partitions' average query estimated cost, and compares this cost with the old cost based on the old query set which was used to generate the current partitions. If the new cost is bigger than the old cost on the majority of the computing nodes, then SMOPD-C will invoke the Filtered AutoClust algorithm that applies the attribute-usage matrix and closed item set mining concepts on the new query set to re-partition the corresponding database table, producing one or more new vertical partitioning solutions. SMOPD-C will then implement the new partitioning solutions on different computing nodes and repeat the process to continue monitoring the most recent queries to make the next re-partitioning decisions. The experimental results using the TPC-H benchmark and the Oracle database system on a cluster computer show that SMOPD-C algorithm can make suitable decisions for re-partitioning action at the correct time, and can accurately generate one or more better re-partitioning solutions based on the most recent query set.

For future work, we plan to extend our research in several directions. In the re-partitioning process, there are two major overheads in our current algorithm. The first overhead is the computation work in order to find out all closed item sets for a particular database table based on a specific query set. The second overhead is that some useless temporary partitions will be physically created in order to evaluate the estimated query cost for a particular partitioning solution. Those two overheads can consume a lot of time when the algorithm is running. However, since our algorithm is running on cluster computers, those two overheads can be reduced by parallelizing the computation work across many computing nodes. This task will be included in our future work. In addition, we will extend SMOPD-C to relational clouds [27]. In a relational cloud, the working environment is a resources sharing and multi-tenants environment. The partitioning and re-partitioning actions for one tenant may impact the virtual machine-hardware assignments and the query performance of other tenants. SMOPD-C will need to be modified to address this issue in order to satisfy the service level agreements (SLAs) for all tenants.

6. Acknowledgement

This work is supported in part by the National Science Foundation grant number CCF-0954310.

7. References

[1] Rodriguez, L., Li, X., Cuevas-Rasgado, D. A., Garcia-Lamont, F., DYVEP: An Active Database System with Vertical Partitioning Functionality, Networking, Sensing and Control (ICNSC), 10th IEEE International Conference, 2013.

[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 Management of 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] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] Garcia-Alvarado, C., Raghavan, V., Narayanan, S. and Waas, F.M., Automatic Data Placement in MPP Databases, Data Engineering Workshops (ICDEW) ofIEEE7th International Conference on Self Managing Database Systems (SMDB), 2012.

[27] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, Relational Cloud: A Database as a Service for the Cloud, in Conference on Innovative Data Systems Research (CIDR), pp. 235–240, 2011.

[28] www.microsoft.com/sql-server.

[29] www.oracle.com/Database.