Self-Managing Online Partitioner for Databases (SMOPD) – A Vertical Database Partitioning System with a Fully Automatic Online Approach

Liangzhe Li
School of Computer Science
University of Oklahoma
Norman, OK 73019, USA
lzli@ou.edu

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK 73019, USA
ggruenwald@ou.edu

ABSTRACT

A key factor of measuring database performance is query response time, which is dominated by I/O time. Database partitioning is among techniques that can help users reduce the I/O time significantly. However, how to efficiently partition tables in a database is not an easy problem, especially when we want to have this partitioning task done automatically by the system itself. This paper introduces an algorithm called Self-Managing Online Partitioner for Databases (SMOPD) in vertical partitioning based on closed item sets mining from a query set and system statistic information mined from system statistic views. This algorithm can dynamically monitor the database performance using user-configured parameters and automatically detect the performance trend so that it can decide when to perform a re-partitioning action without feedback from DBAs. This algorithm can free DBAs from the heavy tasks of keeping monitoring the system and struggling against the large statistic tables. The paper also presents the experimental results evaluating the performance of the algorithm using the TPC-H benchmark.

1. Introduction

In recent years researchers have been paying more and more attention to the development of self-managing database algorithms which includes self-managing database indexing [22] [19], self-managing database caching [18], self-managing database partitioning [1], self-tuning database parameters [17], 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. The disadvantage of a semi-automatic algorithm is that it relies on the experience of the DBA. An experienced DBA means more cost to an organization and is hard to be hired in market.

IDEAS '13, October 09 - 11 2013, Barcelona, Spain

http://dx.doi.org/10.1145/2513591.2513649

That is why more and more researchers are trying to develop fully automatic algorithms for self-managing DBMS [1] [23] [24] [25]. These algorithms should have the ability to know when the database performance is going down and how to tune the database on one or more aspects to get better performance. The algorithm we introduce in this paper is an algorithm that has a fully automatic online approach and focuses on the vertical database partitioning and re-partitioning problem.

Vertical database partitioning algorithms can be roughly classified into two categories: optimizer-independent and optimizer-integrated. 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. Most previous research on database partitioning design ([7] [8] [9] [11] [14]) belongs to the first category. Our SMOPD belongs to the second category, which largely uses the query optimizer as a black-box to perform the optimization. The query optimizer performs the *what-if* calls [2] to select a better execution plan which can be used to find out the best possible partitioning solution. Those algorithms listed above use a fixed workload to decide the partitioning results. However, in many cases, the workload keeps changing; 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.

In this paper we introduce SMOPD, which uses an online approach to monitor the queries running on the database and collects the statistic information from the system statistic views so that it can detect the performance trend and make the re-partitioning decision automatically. The remainder of this paper is organized as follows. In Section 2 we review the related work on database vertical partitioning. In Section 3 we introduce SMOPD. In Section 4 we present the experiment analysis results of SMOPD using the TPC-H benchmark. Finally we provide our conclusions and future work in Section 5.

2. Related work

The first well-known attributes clustering algorithm is called Bond Energy Algorithm [7]. This algorithm represents two different kinds of variables, row variable and column variable, with a two-dimension array. The relationship between each row variable and each column variable is represented by a numeric value. The tighter the relationship is, the bigger the numeric value will be. The algorithm divides the two dimension array

into several fragments by permutating rows and columns so that the similar values can be put together in the same block. Each fragment then represents a clustering partition. This algorithm needs people's judgment to tell the similarity of those numeric values and it has a high chance to create some overlaps among the fragment results.

Another important attributes clustering algorithm was introduced after the Bond Energy Algorithm, which is Navathe's Vertical Partitioning algorithm [8]. It was the first time that a vertical partitioning algorithm considers query frequency as a key factor that may impact the partitioning results; but 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.

Later, after Navathe's Vertical Partitioning algorithm, a new algorithm called Optimal Binary Partitioning algorithm was proposed [9]. This algorithm uses the branch and bound method [10] to construct a binary tree in which each node represents a query. The left branch of the node contains the attributes queried by this node and the right branch contains 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 does reduce time complexity compared to 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.

Some researchers use graph theory to do attribute clustering. One of such examples was introduced in [11] where vertices of a graph are used to represent the attributes and weighted edges are used to represent how often the two attributes connected by the edge appear in the same query. This algorithm traverses the graph and divides the graph into several sub fragments. In each fragment, which represents an attributes cluster, the edges have a similar weight. This algorithm may lead to inefficient partition results when some queries in the query set are used more often than other queries because it considers frequent queries to be the same as infrequent queries.

A new idea of using the query optimizer of a database system for automating the physical database design was proposed in [13]. A query optimizer can estimate how well a query performs without really running the query and such ability gives researchers a new direction when they are working on physical database design.

A more recent attribute clustering algorithm was introduced in [15], which uses the idea of performing clustering based on an attributes affinity matrix from [16]. This algorithm starts with a vertex V that satisfies the least degree of reflexivity and then finds a vertex with the max degree of symmetry among V's neighbors. Once such a neighbor is found, both V and its neighbor are put in a subset. The neighbor would become the new V. The process would continue to search 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 [11] since infrequent queries are treated the same as frequent queries.

The algorithms we have discussed so far are all static. The assumption is that the future workload must be very similar to the one used for generating the partitioning results. To the best of our knowledge, the only true online vertical database partitioning algorithm is AutoStore developed in 2011 [1]. This system has the ability to automatically collect queries and partition the data at checkpoint time intervals. When enough queries are collected, the system will update the old attributes affinity matrix and do permutation on this matrix to make the matrix have the best quality (the quality can be calculated using BEA [7]). 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 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. 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. The second problem, which is a very serious one, is that this algorithm will run re-partitioning every time the CheckpointSize queries are collected no matter what performance trend it has at that time. As we know re-partitioning is very expensive and should not be run too often. Our SMOPD introduced in the next section can solve both of the problems.

We have discussed many existing vertical database partitioning algorithms but only [1] is dynamic. Some other researchers introduced some algorithms that only work for specific cases, such as a recent partitioning system called Schism, which was developed by Curinoet al. [21]. The main idea of their work is similar to the one in [11]. They use a graph to represent a database and query set where tuples are represented by nodes and queries are represented by edges. The system tries to minimize the weights of edges, i.e., the system tries to minimize the multi-sited queries. A limitation of their work is that the system is designed only for short-running OLTP queries. This means the queries only access very few tuples and attributes. For large and complex queries, their system might lose accuracy, that is the result provided might be even worse since complex queries (like OLAP queries) may have a lot of nested joins and access many sites.

3. Auto Online Database Partitioner

In this section, we present our SMOPD system, a special vertical database partitioning system 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 the 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 result candidates is time-consuming.

In SMOPD there are three major components: query collector, statistics analyzer and database cluster, which we introduce in the following sections. As SMOPD uses our AutoClust algorithm [3] [4] to perform vertical partitioning for a database table once it decides that the current partitioning solution for that table is no longer good, we first briefly describe how AutoClust works before presenting the functionality of each component of our system. Interested readers are referred to [3] [4] for details of AutoClust.

3.1. AutoClust

In this section, we summarize the key ideas of AutoClust [3] [4]. There are five steps in our 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) [5] of attributes are mined. An item set is called closed if it has no superset having the same support which is the fraction of transactions in a data set where the item set appears as a subset [5]. 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 we will remove duplicate ACIS. 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 cost estimation and the solution with the best estimated cost is chosen as the final attribute clustering (or vertical database partitioning) solution. This algorithm is used in the database cluster component of our SMOPD system. Now we introduce all the components of our system one by one.

3.2. Query Collector

We know that information retrieval from the main memory (logical I/Os) is much faster than information retrieval from a hard disk (physical I/Os). An attribute clustering technique can help the database system improve the physical I/O performance significantly but not the logical I/O performance. Here we introduce the first configuration parameter, r, we use for our system 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 either 1's 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 system 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.

From the discussion above we can see that the query distribution of a workload could be regarded as a binomial distribution if we use either 1 or 0 to represent a query. According to [12], 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 possibility 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 system, and z_α is a function of α which is the fourth configuration parameter of the system 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. The outliner queries are those that occur rarely. 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. 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. In our experiments, for each run we randomly select 60% of the TPC-H queries, which are 13 different types of queries out of 21 query types in the TPC-H benchmark. The average query frequency (*Avg_Freq*) is $\frac{100\%}{13} = 7.7\%$. We define f_t as 10% of the averagequery 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 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.

The second way of defining f_t is to have the system automatically calculate its value using some specific outlier detection algorithm. In our system, for instance, we can use the classic Grubbs' outlier test (GOT) algorithm [6]. 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 a sample is an outlier or not is calculated according to the following formula:

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

Where $\bar{x}$ is the average query frequency, x_{min} is the minimal query frequency and *s* is the standard deviation of the 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 [6], then it means an outlier query is detected. This configuration method of f_t is good to use when

the DBA has no idea of how the future workload will look like and what the frequency distribution of each query will be. 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.

3.3. Statistics Analyzer

Once the query set *QS* is passed to the second component, Statistics Analyzer, this component will separate the QS into two sub sets, *S0* and *S1*. *S1*contains all the physical read mainly queries and *S0* contains all the logical read mainly queries. *S1* 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.

Once the simplification work is done, an attribute usage matrix is built for each original database table as done in AutoClust [3] [4]. 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 cost for each type of query and, then a new estimated cost is generated by calculating the average estimated cost for all types of queries in the attribute usage matrix. These two steps are discussed in detail in our papers on AutoClust [3] [4]. If a database table's new estimated cost which is generated based on the new query set is bigger

Input:
1. Database tables *T*
2. Queries running on DB
3. Current estimate cost set for each 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* that need to be read
$N = z_\alpha^2 * \frac{fn(1-fn)}{c_\alpha^2}$
Step 2: collect queries
Initialize two empty set *s0*, *s1* and an integer parameter $k = 0$;
While $\frac{s1.szie}{k*N} < fn$ 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 *s0*;
      Remove q_i from *QS*;
    End if
    Else
      Put q_i in *s1*;
      Remove q_i from *QS*;
    End else
  End for
k++;
End do
Step 3: simplify and filter queries in *s1*
Simplify queries in *s1* 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' name in *FQS*[i].query and replace them with original tables' names and put each original 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 attributes 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 AutoClust using *T* and *FQS*

Figure 1. The SMOPD algorithm

than its 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 table set *T*. The Statistics Analyzer examines all the database tables included in the set S1 so that it can find out 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.

3.4. Database Cluster

The third component, Database Cluster, runs our AutoClust algorithm [3] [4]. This component uses *T* and *S1* as input to generate multiple candidate database partitioning solutions and uses the query optimizer to select the best solution for each database table.

The whole algorithm of SMOPD is shown in Figure 1. 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 in 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 *S1* (lines 7-19). In order to make the Database Cluster component understand the queries in *S1,* the Statistics Analyzer creates a set *FQS* (Filtered Query Set) containing the simplified format of each query in *S1* and replaces the sub partition database tables with the original database tables (lines 20-21, 26-28). 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 22-25, 29). Now *FQS* contains all the queries that will heavily impact the partitioning solution, so the Statistics Analyzer calculates their new estimated cost and compares this cost with the old estimated cost read from C_{old} in the configuration file (lines 30-37). If the new estimated 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 tables must be re-partitioned; so it starts to run AutoClust on each table in *T* to find a better partitioning solution to replace the current one (line 38).

4. Experiment result analysis

We conduct experiments to test the performance of SMOPD using the TPC-H benchmark [20]. Since the database tables, NATION and REGION, are very small (5-25 rows), we eliminate them from our experiments. We use the average query estimated cost as the metric to evaluate the performance. We analyze our system's performance by comparing the query estimated cost of the new database partitioning solution with the query estimated cost of the old database partitioning solution. Below we present our experiment model and results.

Table 1. Configuration file parameters

Name	Type	Value Range	Default Value
r	Dynamic	10%-30%	20%
f_n	Dynamic	20%-50%	40%
f_t	Static	N/A	10%*Avg_Freq
c_α	Static	N/A	1%
α	Static	N/A	95%

4.1. Experiment Model

Before we conduct experiments we need to set up all the parameters in a configuration file which will be read every time when SMOPD is running. Table 1 shows all parameters with their values. For each dynamic parameter, when we study its impacts on the system performance, we vary its value within a range and keep other dynamic parameters at their default values.

We test our system using a computer with a processor of Intel Core 2 Quad Q8400, RAM of 3 GB and hard disk of 300 GB, running on Windows 7. The database system is Oracle 11g EX Edition. The program is coded in Java.

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 AutoClust once to get the first partitioning solution.

AutoClust is based on the closed item sets mining. The output of the closed item sets is determined by the attribute usage table. If there is no change in the attributes usage table then the output of the closed item sets will keep unchanged. So we have to use a new attribute usage table for each original TPC-H database table. 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 for each query selected (this ensures that we have a change in query frequency).

4.2. Impacts of f_n

According to Table 1 f_n changes from 20% to 50%.the experiment results show that the percentage of unnecessary runs (i.e. SMOPD cannot find a better partitioning solution) is small, which happens only 1 out of 18 runs. The system was able to correctly identify all cases when no repartitioning is needed. Overall our system offers good performance improvement through its automatic decision on re-partitioning. The average performance improvement by our system over all values of f_n is shown in Figure 2.

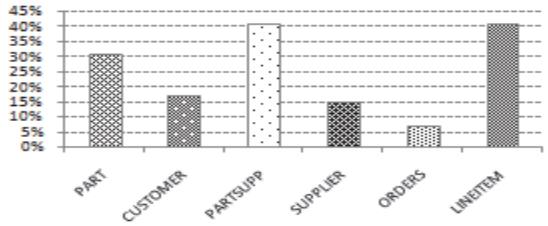

Figure 2. Average performance improvement based on estimated query cost over all values of f_n

4.3. Impacts of *r*

The parameter *r* changes from 10% to 30% according to Table 1.The experiment results show that our system performed 18 cost comparisons and 10 re-partitioning actions. Out of 10 re-partitioning actions, 8 actions are correct, and the new solutions provided by our system are always the best ones based on the current query sets. For the 8 cases where SMOPD decides that re-partitioning is not needed, 7 cases are correct. The average performance improvement of our system over all values of *r* tested for each TPC-H benchmark database table is shown in Figure 3.

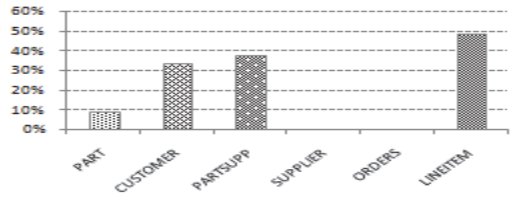

Figure 3. Average performance improvement based on estimated query cost over all values of *r*

5. Conclusions

In this paper we presented SMOPD a fully automatic vertical database partitioning system. Its goal is to locate those database tables the average response time for a query set running on which is getting bigger (measured by the estimated cost of the query optimizer) according to the current partitioning solution and propose the best new partitioning solution from all solutions it generates. The whole process is fully automated with an online approach. From the experiments using the TPC-H benchmark, we can see that our system can automatically split database tables into two groups: good performance group and bad performance group. For each database table in the bad performance group, the system tries to re-partition the database tables using the queries it has collected. If the system determines that a better database partitioning solution exists, then it rebuilds the partition for this database table automatically; otherwise, it keeps using the current database partitioning solution. Our algorithm currently works for one single machine only. Our future work is to extend our system to cluster computers so that each computing node in the cluster computers could automatically change the partitions for the necessary database tables based on the query set executed on that node, and during the re-partitioning process other computing nodes should be able to work normally.

6. Acknowledgement

This work is partially supported by National Science Foundation grant No. 0954310.

References

[1] Jindal, A., and Dittrich, J., Relax and Let the Database do the Partitioning Online. In BIRTE, 2011.

[2] Rao, J., Zhang, C., Megiddo, N., and Lohman, G. M., Automating Physical Database Design in a Parallel Database. In SIGMOD, page 558-569, 2002.

[3] Guinepain, S. and Gruenwald, L., Using Cluster Computing to Support Automatic and Dynamic Database Clustering, IWAPT, 2008.

[4] Li, L., and Gruenwald, L., Autonomous Database Partitioning Using Data Mining on Single Computers and Cluster Computers, IDEAS12, August 2012.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] Navathe, S. and Ra M., Vertical Partitioning for Database Design: A Graph Algorithm, ACM SIGMOD International Conference on Management of Data, 1989.

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

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

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

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

[16] 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.

[17] Duan S., Thummala V., and Babu S., Tuning Database Configuration Parameters with Ituned, Proc. VLDB Endow., vol. 2, pp. 1246–1257, August 2009.

[18]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.

[19] 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.

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

[21] Curino C., et al., Schism: a Workload-Driven Approach to Database Replication and Partitioning. In VLDB, 2010.

[22] Schnaitter, K., and Polyzotis, N., Semi-Automatic Index Tuning: Keeping DBAs in The Loop. PVLDB, 5(5):478–489, 2012.

[23] Agrawal S., Chu E., and Narasayya V., Automatic Physical Design Tuning: Work-load as a Sequence. In SIGMOD, 2006.

[24] Agrawal, S., et al. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In SIGMOD, 2004.

[25] Bruno,N., and Chaudhuri, S., Constrained Physical Design Tuning. PVLDB, 2008.