An Active System for Dynamic Vertical Partitioning of Relational Databases

Lisbeth Rodríguez, Xiaoou Li, and Pedro Mejía-Alvarez

Department of Computer Science, CINVESTAV-IPN, Mexico D.F., Mexico
lisbethr@computacion.cs.cinvestav.mx,
{lixo,pmalvarez}@cs.cinvestav.mx

Abstract. Vertical partitioning is a well known technique to improve query response time in relational databases. This consists in dividing a table into a set of fragments of attributes according to the queries run against the table. In dynamic systems the queries tend to change with time, so it is needed a dynamic vertical partitioning technique which adapts the fragments according to the changes in query patterns in order to avoid long query response time. In this paper, we propose an active system for dynamic vertical partitioning of relational databases, called DYVEP (DYnamic VErtical Partitioning). DYVEP uses active rules to vertically fragment and refragment a database without intervention of a database administrator (DBA), maintaining an acceptable query response time even when the query patterns in the database suffer changes. Experiments with the TPC-H benchmark demonstrate efficient query response time.

Keywords: Active systems, active rules, dynamic vertical partitioning, relational databases.

1 Introduction

Vertical partitioning has been widely studied in relational databases to improve query response time [1-3]. In vertical partitioning, a table is divided into a set of fragments, each with a subset of attributes of the original table and defined by a vertical partitioning scheme (VPS). Fragments consist of smaller records, therefore, fewer pages from secondary memory are accessed to process queries that retrieve or update only some attributes from the table, instead of the entire record [3].

Vertical partitioning can be static or dynamic [4]. Most works consider a static vertical partitioning based on a priori probabilities of queries accessing database attributes in addition to their frequencies which are available during the analysis stage. It is more effective for a database to dynamically check the goodness of a VPS to determine whenever refragmentation is necessary [5].

Static vertical partitioning works only consider that the queries that operate on the relational database are static and a VPS is optimized for such queries. Nevertheless, applications like multimedia, e-business, decision support, and geographic information systems are accessed by many users simultaneously. Therefore, queries

I. Batyrshin and G. Sidorov (Eds.): MICAI 2011, Part II, LNAI 7095, pp. 273–284, 2011.

tend to change over time, and a refragmentation of the database is needed when query patterns and database scheme have undergone sufficient changes.

Dynamic vertical partitioning techniques automatically trigger the refragmentation process if it is determined that the VPS in place has become inadequate due to a change in query patterns or database scheme. This implies to develop a system which can trigger itself and make decision on their own.

Active systems are able to respond automatically to events that are taking place either inside or outside the system itself. The central part of those systems is a set of active rules which codifies the knowledge of domain experts [6]. Active rules constantly monitor systems and user activities. When an interesting event happens, they respond by executing certain procedures related either to the system or to the environment [7].

The general form of an active rule is the following:

ON *event*
IF *condition*
THEN *action*

An event is something that occurs at a point in time, e.g., a query in database operation. The condition examines the context in which the event has taken place. The action describes the task to be carried out by the rule if the condition is fulfilled once an event has taken place. Several applications, such as smart homes, sensor and active databases integrate active rules for the management of some of their important activities [8].

In this paper, we propose an active system for dynamic vertical partitioning of relational databases, called DYVEP (DYnamic VErtical Partitioning). Active rules allow DYVEP to automatically monitor the database in order to collect statistics about queries, detect changes in query patterns, evaluate the changes and when the changes are greater than a threshold, trigger the refragmentation process.

The rest of the paper is organized as follows: in Section 2 we give an introduction on dynamic vertical partitioning. In Section 3 we present the architecture of DYVEP. Section 4 presents the implementation of DYVEP, and finally Section 5 is our conclusion.

2 Dynamic Vertical Partitioning

2.1 Motivation

Vertical partitioning can be static and dynamic [5]: In the former, attributes are assigned to a fragment only once at creation time, and then their locations are never changed. This approach has the following problems:

1. The DBA has to observe the system for a significant amount of time until probabilities of queries accessing database attributes in addition to their frequencies are discovered before the partitioning operation can take place. This is called an analysis stage.

2. Even then, after the partitioning process is completed, nothing guarantees that the real trends in queries and data have been discovered. Thus the VPS may not be good. In this case, the database users may experience very long query response time [14].
3. In some dynamic applications, queries tend to change over time and a VPS is implemented to optimize the response time for one particular set of queries. Thus, if the queries or their relative frequencies change, the partitioning result may no longer be adequate.
4. With static vertical partitioning methods, refragmentation is a heavy task and only can be performed manually when the system is idle [11].

In contrast, with dynamic vertical partitioning, attributes are being relocated if it is determined that the VPS in place has become inadequate due to a change in query information. We develop DYVEP to improve the performance of relational database systems. Using active rules, DYVEP can monitor queries run against the database in order to accumulate the accurate information to perform the vertical partitioning process, eliminating the cost of the analysis stage. It also automatically reorganizes the fragments according to the changes in query patterns and database scheme, achieving good query performance at all times.

2.2 Related Work

Liu Z. [4] presents an approach for dynamic vertical partitioning to improve query performance in relational databases, this approach is based on the feedback loop used in automatic performance tuning, which consists of observation, prediction and reaction. It observes the change of workload to detect a relatively low workload time, and then it predicts the coming workload based on the characteristics of current workload and implements the new vertical partitions.

Reference [9] integrates both horizontal and vertical partitioning into automated physical database design. The main disadvantage of this work is that they only recommend the creation of vertical fragments but the DBA has to create the fragments. DYVEP has a partitioning reorganizer which creates automatically the fragments on disk.

Autopart [10] is an automated tool that partitions the relations in the original database according to a representative workload. Autopart receives as input a representative workload and designs a new schema using data partitioning, one drawback of this tool is that the DBA has to give the workload to autopart. In contrast, DYVEP collects the SQL statements when they are executed.

Dynamic vertical partitioning is also called dynamic attribute clustering. Guinepain and Gruenwald [1] present an efficient technique for attribute clustering that dynamically and automatically generates attribute clusters based on closed item sets mined from the attributes sets found in the queries running against the database.

Most dynamic clustering techniques [11-13] consist of the following modules: a statistic collector (SC) that accumulates information about the queries run and data returned. The SC is in charge of collecting, filtering, and analyzing the statistics. It is responsible for triggering the Cluster Analyzer (CA). The CA determines the best

possible clustering given the statistics collected. If the new clustering is better than the one in place, then CA triggers the reorganizer that physically reorganizes the data on disk [14]. The database must be monitored to determine when to trigger the CA and the reorganizer.

To the best of our knowledge there are not works related to dynamic vertical partitioning using active rules. Dynamic vertical partitioning can be effectively implemented as an active system because active rules are expressive enough to allow specification of a large class of monitoring tasks and they do not have noticeable impact on performance, particularly when the system is under heavy load. Active rules are amenable to implementation with low CPU and memory overheads [15].

3 Architecture of DYVEP

In order to get good query performance at any time, we propose DYVEP, which is an active system for dynamic vertical partitioning of relational databases. 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, in such case; it triggers the Vertical Partitioning Algorithm (VPA). If the VPS is better that the one in place, then the system reorganizes the scheme. Using active rules, DYVEP can react to the events generated by users or processes, evaluate conditions and if the conditions are true, then execute the actions or procedures defined.

The architecture of DYVEP is shown in Fig. 1. DYVEP is composed of 3 modules: *Statistic Collector*, *Partitioning Processor*, and *Partitioning Reorganizer*.

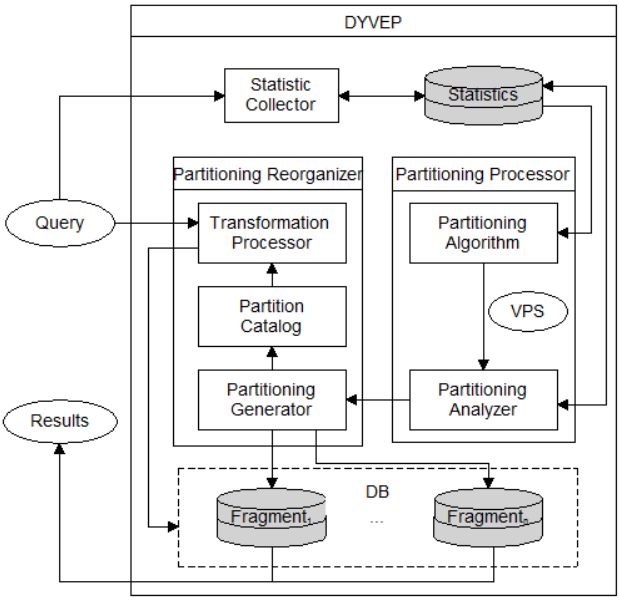

Fig. 1. Architecture of DYVEP

3.1 Statistic Collector

The statistic collector accumulates information about the queries (such as id, description, attributes used, access frequency) and the attributes (name, size). When DYVEP is executed for first time in the database, the statistic collector creates the tables queries (QT), attribute_usage_table (AUT), attributes (AT) and statistics (stat) and a set of active rules in such tables.

After initialization, when a query (q_i) is run against the database, the statistic collector verifies if the query is not stored in QT; in that case it assigns an id to the query, stores its description, and sets its frequency to 1 in QT. If the query is already stored in QT, only its frequency is increased by 1. This is defined by the following active rule:

Rule 1
ON $q_i \in Q$
IF $q_i \notin$ QT
THEN insert QT (*id, query, freq*) values (*id_ q_i, query_ q_i, 1*)
ELSE update QT set *freq*=old.freq+1 where *id*=id_ q_i

In order to know if the query is already stored in QT, the statistic collector has to analyze the queries. Two queries are considered equal if they use the same attributes, for example if we have the queries:

q_1: SELECT *A, B* FROM T
q_2: SELECT SUM (*B*) FROM T WHERE *A*=Value

If q_1 is already stored in QT and q_2 is run against the database, the statistic collector analyzes q_2 in order to know the attributes used by the query, and compares q_2 with the queries already stored in QT, since q_1 uses the same attributes then its frequency is increased by 1.

The statistic collector also registers the changes in the information of queries and attributes over time and compares the current changes (*currentChange*) with the previous changes (*previousChange*) in order to determine if they are enough to trigger the VPA. For example, when a query is inserted or deleted in QT after initialization, the changes in queries are calculated. If the changes are greater than a threshold, then VPA is triggered.

The changes in queries are calculated as the number of inserted or deleted queries after a refragmentation divided by the total number of queries before refragmentation. For example, if QT had 8 queries before the last refragmentation and one query is inserted after refragmentation, then the change in queries is equal to 1/8*100=12.5%. If the value of the threshold is 10%, then VPA will be triggered.

The threshold is updated after each refragmentation and it is defined as previousChange plus *currentChange* divided by two.

The following rules are implemented in the statistic collector:

Rule 2
ON insert or delete QT
THEN update stat set *currentNQ*=*currentNQ*+1

Rule 3
ON update stat.*currentNQ*
IF *currentNQ*>0 and *previousNQ*>0
THEN update stat set *currentChange*=*currentNQ*/*previousNQ**100

Rule 4
ON update stat.*currentChange*
IF *currentChange*>threshold
THEN call VPA

3.2 Partitioning Processor

The partitioning processor has two components: the partitioning algorithm and the partitioning analyzer. The partitioning algorithm determines the best VPS given the collected statistics, which is presented in Algorithm 1.

The partitioning analyzer detects if the new VPS is better than the one in place, then the partitioning analyzer triggers the partitioning generator in the partitioning reorganizer module. This is defined using an active rule:

Rule 5
ON *new VPS*
IF *new_VPS_cost*<*old_VPS_cost*
THEN call *partitioning_generator*

```
Algorithm 1. Vertical Partitioning Algorithm

input: QT: Query Table
output: Optimal vertical partitioning scheme (VPS)
begin
{Step 1: Generating AUT}
   getAUT(QT, AUT)
   {generate the AUT from QT}
{Step 2: Getting the optimal VPS}
   getVPS(AUT, VPS)
   {get the optimal VPS using the AUT of step 1}
end. {VPA}
```

3.3 Partitioning Reorganizer

The partitioning reorganizer physically reorganizes the fragments on disk. It has three components: a partitioning generator, a partition catalog and a transformation processor. The partitioning generator creates the new VPS, deletes the old scheme and registers the changes in the partitioning catalog. The partitioning catalog contains the location of the fragments and the attributes of each fragment. The transformation processor transforms the queries so that they can execute correctly in the partitioned domain. This transformation involves replacing attribute accesses in the original

query definition with appropriate path expressions. The transformation processor uses the partitioning catalog to determine the new attribute location.

When a query is submitted to the database DYVEP triggers the transformation processor, which changes the definition of the query according to the information located in the partitioning catalog. The transformation processor sends the new query to the database; the database then executes the query and provides the results.

4 Implementation

We have implemented DYVEP using triggers inside the open source PostgreSQL object-relational database system running on a single processor 2.67-GHz Intel (R) Core(TM) i7CPU with 4 GB of main memory and 698-GB hard drive.

4.1 Benchmark

As an example, we use the TPC-H benchmark [16], which is an ad-hoc, decision support benchmark widely used today in evaluating the performance of relational database systems. We use the partsupp table of TPC-H 1 GB; partsupp has 800,000 tuples and 5 attributes.

In most of today's commercial database systems, there is not native DDL support for defining vertical partitions of a table [9]. Therefore, it can be implemented as a relational table, a relational view, an index or a materialized view. If the partition is implemented as a relational table, it may cause a problem of optimal choice of partition for a query. For example, suppose we have table

partsupp
(*ps_partkey,*
ps_suppkey,
ps_availqty,
ps_supplycost,
ps_comment),

Partitions of partsupp::

partsupp_1(*ps_partkey, ps_psavailqty, ps_suppkey, ps_supplycost*)
partsupp_2(*ps_partkey, ps_comment*)
Where *ps_partkey* is the primary key. Considering a query:

SELECT *ps_partkey*, *ps_comment* FROM partsupp

The query of selection of partsupp cannot be transformed to selection from partsupp_2 by query optimizer automatically. If the partition is implemented as a materialized view, the query processor in the database management system can detect the optimal materialized view for a query and be able to rewrite the query to access the optimal materialized view. If the partitions are implemented as indexes over the relational tables, the query processor is able to detect that horizontal traversal of an index is equivalent to a full scan of a partition. Therefore implementing the partitions

either as a materialized view or index allows the changes of the partition as transparent to the applications [4].

4.2 Illustration

DYVEP is implemented as an SQL script, the DBA who wants to partition a table executes only once DYVEP.sql in the database which contains the table to be partitioned. DYVEP will detect that it is the first execution and will create the tables, functions and triggers to implement the dynamic vertical partitioning.

Step 1. The first step of DYVEP is to create an initial vertical partitioning, to generate this, the Statistic collector of DYVEP analyzes the queries stored in the statement log and copies the queries run against the table to be partitioned in the table queries (QT). To implement the **Rule 1** on this table, we create a trigger called *insert_queries*.
Step 2. When all the queries has been copied for the statistic collector, then it triggers the vertical partitioning algorithm, DYVEP can use any algorithm that uses as input the attribute_usage_table (AUT), as an example, the vertical partitioning algorithm implemented in DYVEP is the Navathe's algorithm [2], we selected this algorithm because is a classical vertical partitioning algorithm.
Step 3. The partitioning algorithm first will get the AUT from the QT, the AUT has two triggers for each attribute of the table to be fragmented, one trigger for insert and delete and one for update, in this case we have the triggers *inde_ps_partkey*, *update_ps_partkey*, etc., these triggers provide the ability to update the attribute_affinity_table (AAT) when the frequency or the attributes used by the query suffer changes in the AUT, an example of rule definition for the attribute *ps_partkey* is

Rule 6
ON update AUT
IF new.*ps_partkey*=true
THEN update AAT set *ps_partkey*=*ps_partkey*+new.*frequency* where attribute=*ps_partkey*

Step 4. When the AAT is updated, a procedure called BEA is triggered, a rule definition for this is:

Rule 7
ON update AAT
THEN call BEA

BEA is the Bond Energy Algorithm [17], which is a general procedure for permuting rows and columns of a square matrix in order to obtain a semiblock diagonal form. The algorithm is typically applied to partition a set of interacting variables into subsets which interact minimally. The application of the procedure BEA to the AAT generates the clustered affinity table (CAT),

Step 5. Once CAT has been generated, a procedure called partition is triggered which receives as input the CAT and gets the vertical partitioning scheme (VPS).

Step 6. When the initial VPS is obtained, the partitioning algorithm triggers the partitioning generator which materializes the VPS, i.e., creates the fragments on disk. The active rule for this is:

Rule 8
ON NEW VPS
IF *VPS_status*=initial
THEN call *partitioning_generator*

Step 7. The partitioning generator implements the fragments as materialized views, so the query processor of PostgreSQL can detect the optimized materialized view for a query and is able to rewrite the query to access the optimal materialized view instead of the complete table. This provides fragmentation transparency to the database.

A screenshot of DYVEP is given in Fig. 2. A scheme called DYVEP is created in the database. In such scheme, all the tables (queries, attribute_usage_table, attribute_affinity_table, clustered_affinity_table) from the DYVEP system are located, the triggers *inde_attributename*, *update_attributename* are generated automatically by DYVEP according to the view attributes, therefore the number of triggers in our system will depend on the number of attributes of the table to fragment.

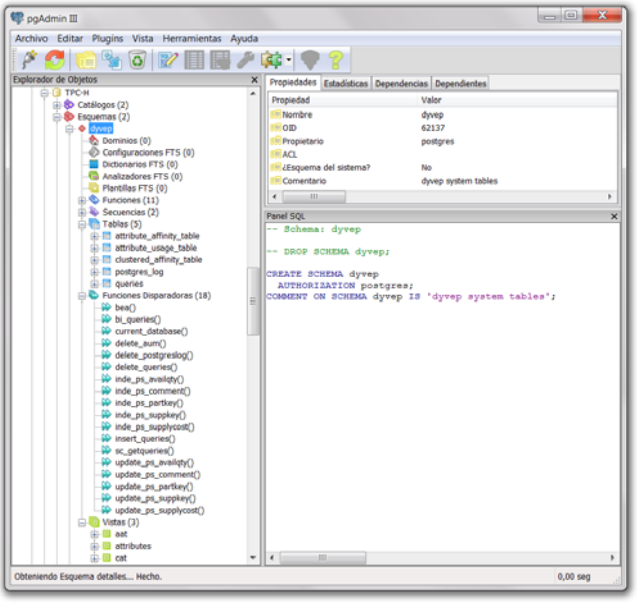

Fig. 2. Screenshot of DYVEP in PostgreSQL

4.3 Comparisons

Having the following queries

q_1: SELECT SUM(*ps_availqty*) FROM partsupp WHERE *ps_partkey*=Value
q_2: SELECT *ps_suppkey*, *ps_availqty* FROM partsupp

q_3: SELECT *ps_suppkey*, *ps_supplycost* FROM partsupp WHERE *ps_partkey*=Value
q_4: SELECT *ps_comment*, *ps_partkey* FROM partsupp

DYVEP got the attribute usage table of Fig. 3. The VPS obtained by DYVEP according to the attribute usage table was

partsupp_1 (ps_*partkey*, *ps_psavailqty*, *ps_suppkey*, *ps_supplycost*)
partsupp_2 (*ps_partkey*, *ps_comment*)

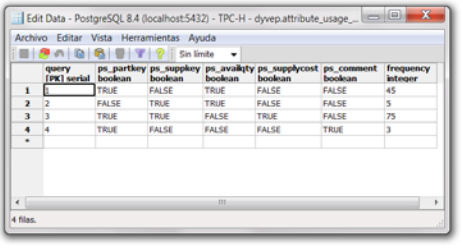

Fig. 3. Attribute Usage Table

In Table 1 we can see the execution time of these queries in TPC-H not partitioned (NP) vs. vertically partitioned using DYVEP. As we can see, the execution time of the queries in TPC-H vertically partitioned using DYVEP is lower than in a TPC-H not partitioned, therefore DYVEP can generate schemes that can significantly improve query execution, even without the use of any indexes.

Table 1. Comparison of query execution time

TPC_H	q_1	q_2	q_3	q_4
NP	47 ms	16770 ms	38 ms	108623 ms
DYVEP	15 ms	16208 ms	16 ms	105623 ms

5 Conclusion and Future Work

A system architecture for performing dynamic vertical partitioning of relational databases has been designed, which can adaptively modify the VPS of a relational database using active rules within efficient query response time. The main advantages of DYVEP over other approaches are:

1. Static vertical partitioning strategies [2] take into account an a priori analysis stage of the database in order to collect the necessary information to perform the vertical partitioning process, also in some automated vertical partitioning tools [9, 10] it is necessary that the DBA gives as input the workload. In contrast, DYVEP implements an active-rule based statistic collector which accumulates

information about attributes, queries and fragments without the explicit intervention of the DBA.

2. When the information of the queries changes in the static vertical partitioning strategies, then the fragment configuration will remain in the same way and will not implement the best solution. In DYVEP the fragment configuration will change dynamically according to the changes in the information of the queries in order to find the best solution and not affect the performance of the database.
3. The vertical partitioning process in the static approaches is performed outside of the database and when the solution is found the vertical fragments are materialized. In DYVEP all the vertical partitioning process is implemented inside the database using rules, the attribute usage matrix (AUM) used by most of the vertical partitioning algorithms is implemented as a database table (AUT) in order to use rules to change the fragment configuration automatically.
4. Some automated vertical partitioning tools only recommend the optimal vertical partitioning configuration but they leave the creation of the fragments to the DBA [9], DYVEP has an active rule-based partitioning reorganizer that automatically creates the fragments on disk when is triggered by the partitioning analyzer.

In the future, we want to extend our results to multimedia database system. Multimedia database systems are highly dynamic, so the advantages of DYVEP would be seen much clearly, especially on reducing the query response time.

References

1. Guinepain, S., Gruenwald, L.: Using Cluster Computing to support Automatic and Dynamic Database Clustering. In: Third International Workshop on Automatic Performance Tuning (IWAPT), pp. 394–401 (2008)
2. Navathe, S., Ceri, S., Wiederhold, G., Dou, J.: Vertical Partitioning Algorithms for Database Design. ACM Trans. Database Syst. 9(4), 680–710 (1984)
3. Guinepain, S., Gruenwald, L.: Automatic Database Clustering Using Data Mining. In: 17th Int. Conf. on Database and Expert Systems Applications, DEXA 2006 (2006)
4. Liu, Z.: Adaptive Reorganization of Database Structures through Dynamic Vertical Partitioning of Relational Table., MCompSc thesis, School of Information Technology and Computer Science, University of Wollongong (2007)
5. Sleit, A., AlMobaideen, W., Al-Areqi, S., Yahya, A.: A Dynamic Object Fragmentation and Replication Algorithm in Distributed Database Systems. American Journal of Applied Sciences 4(8), 613–618 (2007)
6. Chavarría-Baéz, L., Li, X.: Structural Error Verification in Active Rule Based-Systems using Petri Nets. In: Gelbukh, A., Reyes-García, C.A. (eds.) Fifth Mexican International Conference on Artificial Intelligence (MICAI 2006), pp. 12–21. IEEE Computer Science (2006)
7. Chavarría-Baéz, L., Li, X.: ECAPNVer: A Software Tool to Verify Active Rule Bases. In: 22nd International Conference on Tools with Artificial Intelligence (ICTAI), pp. 138–141 (2010)

8. Chavarría-Baéz, L., Li, X.: Termination Analysis of Active Rules - A Petri Net Based Approach. In: IEEE International Conference on Systems, Man and Cybernetics, San Antonio, Texas, USA, pp. 2205–2210 (2009)
9. Agrawal, S., Narasayya, V., Yang, B.: Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In: Proc. of the 2004 ACM SIGMOD Int. Conf. on Management of Data, pp. 359–370 (2004)
10. Papadomanolakis, E., Ailamaki, A.: AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. CMU Technical Report, CMU-CS-03-159 (2003)
11. Darmont, J., Fromantin, C., Régnier, S., Gruenwald, L., Schneider, M.: Dynamic Clustering in Object-Oriented Databases: An Advocacy for Simplicity. In: Dittrich, K.R., Oliva, M., Rodriguez, M.E. (eds.) ECOOP-WS 2000. LNCS, vol. 1944, pp. 71–85. Springer, Heidelberg (2001)
12. Gay, J.Y., Gruenwald, L.: A Clustering Technique for Object Oriented Databases. In: Tjoa, A.M. (ed.) DEXA 1997. LNCS, vol. 1308, pp. 81–90. Springer, Heidelberg (1997)
13. McIver Jr., W.J., King, R.: Self-Adaptive, on-Line Reclustering of Complex Object Data. In: Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data (1994)
14. Guinepain, S., Gruenwald, L.: Research Issues in Automatic Database Clustering. SIGMOD Record 34(1), 33–38 (2005)
15. Chaudhuri, S., Konig, A.C., Narasayya, V.: SQLCM: a Continuous Monitoring Framework for Relational Database Engines. In: Proc. of the 20th Int. Conf. on Data Engineering, ICDE (2004)
16. Transaction Processing Performance Council TPC-H benchmark, `http://www.tpc.org/tpch`
17. McCormick, W.T., Schweitzer, P.J., White, T.W.: Problem Decomposition and Data Reorganization by a Clustering Technique. Operations Research 20(5), 973–1009 (1972)