Using Cluster Computing to Support Automatic and Dynamic Database Clustering

Sylvain Guinepain, Le Gruenwald

School of Computer Science, The University of Oklahoma
Norman, OK 73019, USA
Sylvain.Guinepain@ou.edu
gGruenwald@ou.edu

***Abstract*— Query response time is the number one metrics when it comes to database performance. Because of data proliferation, efficient access methods and data storage techniques have become increasingly critical to maintain an acceptable query response time. Retrieving data from disk is several orders of magnitude slower than retrieving it from memory, it is easy to see the direct correlation between query response time and the number of disk I/Os. One of the common ways to reduce disk I/Os and therefore improve query response time is database clustering, which is a process that partitions the database vertically (attribute clustering) and/or horizontally (record clustering). A clustering is optimized for a given set of queries. However in dynamic systems the queries change with time, the clustering in place becomes obsolete, and the database needs to be re-clustered dynamically.**

This paper presents an efficient algorithm 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. The paper then discusses how this algorithm can be implemented using the cluster computing paradigm to reduce query response time even further through parallelism and data redundancy.

I. Introduction

We live in the information age and our society's appetite for knowledge and data has created the need for very large database support. Databases in many applications, such as those that process large volumes of sales transactions, medical records and phone calls, can be very large. The usefulness of these databases highly depends on how quickly data can be retrieved. Clever data organization is one of the best ways to improve retrieval speed. By improving the data storage, we can reduce the number of disk I/Os and thereby reduce the query response time.

In this paper we combine database clustering and parallelism to reduce the cost of I/Os. We present our automatic database clustering algorithm, AutoClust, and show how it can benefit from a cluster computing architecture.

Specifically, we describe the attribute clustering algorithm in AutoClust. In attribute clustering, attributes of a relation are divided into groups based on their affinity. Clusters consist of smaller records, therefore, fewer pages from secondary memory are accessed to process transactions that retrieve or update only some attributes from the relation, instead of the entire record [14]. This leads to better query performance.

AutoClust is an automated and dynamic clustering technique. It is well documented that with the ever-growing size and number of databases to monitor, human attention has become a precious resource [5]. In response to this concern, the computing world is relying more and more on automated self-managing systems capable of making intelligent decision on their own. The area of autonomic computing has been getting a lot of attention [1][17][7][18]. For Autoclust to be useful it should be fully automated, which implies that no human intervention is needed during the clustering process. To do this, the algorithm generates attribute clusters automatically based on closed item sets mined from the attributes sets found in the queries running against the database.

Finally in today's world, the advent of the Internet has made cluster computing a powerful and cost-effective way to share and process data. AutoClust can take advantage of this computing paradigm to not only speed up its execution time but also produce an efficient data storage scheme where the query response time of the resulting database is faster than that of AutoClust running on a single node.

The remainder of this paper is organized as follows. In Section II we review the relevant literature in the areas of traditional attribute clustering, data mining clustering and autonomic computing. In Section III we describe our autonomic attribute clustering algorithm, AutoClust. In Section IV we discuss the adaptation of AutoClust to cluster computers. Finally we give our conclusions in Section V.

II. Literature Review

Minimizing the number of disk I/Os has long been a topic of interest with the database community. Efforts to achieve reduced I/Os though data clustering can be retraced back to the early 1970s. The first techniques were relying solely on finding patterns within the data itself.

The clustering problem is a very difficult problem and the number of solutions is equal to the Bell number [4] that follows the following recurrence relation: $b_{n+1} = \sum_{k=0}^{n} b_k \binom{n}{k}$, where n is the total number of attributes we wish to cluster.

The first well-known attribute clustering technique is credited to [13] with his Bond Energy Algorithm (BEA). The purpose of the BEA is to identify and display natural variable groups and clusters that occur in complex data arrays by permuting the rows and columns so as to push the numerically larger array elements together. The resulting matrix is in block

978-1-4244-2640-9/08/$25.00 © 2008 IEEE

Invited Talk
Third international Workshop on
Automatic Performance Tuning (iWAPT 2008)

diagonal form. It is hard however to determine how many clusters there are and what attributes they contain. The interpretation is subjective and therefore requires human input and cannot be considered reliable.

Navathe's Vertical Partitioning (NVP) [14] added a second phase to the BEA algorithm in an effort to reduce the subjectivity of the final interpretation. In the phase, the author performs the BEA algorithm against an affinity matrix containing all pairs of attributes in the database. The BEA is then used to rearrange the rows and columns of the matrix such that the value of the global affinity function is maximized. The rearranged matrix is called the clustered affinity (CA) matrix, which then becomes an input to the second phase of the technique called the Binary Vertical Partitioning (BVP) algorithm. BVP recursively partitions the CA matrix into two halves in order to minimize the number of transactions that access attributes in both the halves. This technique has two drawbacks: 1) the objective function to maximize in phase 2 is subjective and alternative functions could produce different results, and 2) the solution only contains two clusters of attributes.

Data mining clustering is another tool to group data items together by finding similarities in the data itself. To accomplish this, data mining clustering algorithms use a similarity measure or distance function to determine the distance/similarity between any two data items [9]. The objective is to create groups or clusters where the elements within each group are alike and elements between groups are dissimilar. Elements in the same cluster are alike and elements in different clusters are not alike.

The three techniques described so far suffer the same problem. These techniques group data items based on similarities found in the actual data, not based on attribute usage by queries. Thus two data items could be stored together because they were found to be similar but rarely be accessed together. The problem with such an approach is that it only helps with record clustering and does not help reducing the number of I/Os for queries accessing only few attributes of a relation. Realizing that the next evolution in clustering algorithm was transaction-based clustering.

A transaction-based vertical partitioning technique, the Optimal Binary Partitioning algorithm (OBP), is proposed in [8] where the attributes of a relation are partitioned according to a set of transactions. This concept derives from the fact that transactions carry more semantic meaning than attributes. The technique creates clusters by splitting up the set of attributes recursively. Each cluster of attributes is separated into two groups with each query: the attributes that are part of the query and those that are not. At each sub level of the tree, a new query is used to split the clusters further. In the end the algorithm produces a tree whose leaves contain the split up clusters of attributes. A cost function is then applied to determine the optimal binary partitioning while merging adjacent leaves of the tree. The disadvantages of this technique are that it may have to examine a large number of possible partitions in order to find the optimal binary partitioning and it produces partitions that contain only two clusters. Other more recent clustering algorithms can cluster attributes in more than 2 clusters. This allows for better performance when the relations have many attributes and there are many queries.

A graph theory approach to the clustering problem was proposed in [12] with a clustering technique based on graph connectivity. The similarity data is used to form a similarity graph in which vertices correspond to elements and edges connect elements with similarity values above some threshold. Clusters are highly connected sub-graphs, which are defined as subgraphs whose edge connectivity exceeds half of the number of vertices. This technique does not take into account query frequencies and the resulting solution could contain clusters that favour infrequent queries over more frequent ones.

[1], [2] is Microsoft's data mining based solution to the automatic clustering problem. Using the attribute affinity matrix, the algorithm mines the frequent item sets of attributes and retains the top k ordered by confidence level. Each attribute-set forms a binary partition: attributes in the sets and attributes not in the set. The algorithm then determines which such binary partition is optimal for each individual query. The cost is obtained by creating two sub-tables corresponding to the two clusters and running the query through the query optimizer to obtain its cost. Then a merging step combines the resulting binary clusters two at a time and evaluates the cost of all possible merged partitions. In the end the merged partition with the best cost is selected for the table. The authors clearly state that their goal is "to optimize performance of a database for a given workload". This means that this clustering is static and, given its ties to other database objects such as indices, it is not possible to convert it into a dynamic solution.

Using the query optimizer for automated physical design was also investigated in [15]. The authors introduce the Index Usage Model (INUM), a cost estimation technique that returns the same values that would have been returned by the optimizer, but with 3 orders of magnitude faster. This is then used in the context of index selection.

A report of the last 10 years of progress in the area of self-tuning databases can be found in [7]. Through the eyes of the AutoAdmin project at Microsoft, the authors review the progress in automated database management.

As we have just seen none of the attribute-clustering algorithm reviewed in the literature is autonomic. Some require manual and subjective interpretation of the results; some only produce two clusters of attributes; others use subjective parameters that result in sub-optimal solutions if their values are not chosen carefully. In the next section we describe our proposed attribute-clustering algorithm. It is autonomic, can produce any number of clusters and requires no parameters.

III. AUTOCLUST

In this section, we present the improved version of our attribute clustering algorithm [11] and discuss how it can take advantage of the cluster computing paradigm. Note that unlike

our previous version of the algorithm we do not need to keep track of the result set size returned by each query. This is due to the fact that we changed our cost-model and that the cost of each clustering solution is now estimated by the query optimizer only. In addition to a better cost-model, this adds the advantage that no information is needed about the query besides its SQL formulation.

The attribute clustering is done in four steps, which are described in this section. In Step 1, we build a frequency-weighted attribute usage matrix. In Step 2, we mine the closed item sets (CIS) of attributes. A closed item set is a maximal item set contained in the same transactions. This information tells us what attributes have high affinity, i.e., are often accessed together. In Step 3, the closed item sets mined in Step 2 are augmented and filtered in such a way that the original tuples can be reconstructed through a natural join after the partitioning has taken place. We restrict the set of CIS considered to:

- CIS containing attributes from the same relation and
- CIS containing attributes from several relations as long as the cardinality between all the relations containing these attributes is 1 to 1.

Every CIS that does not contain the primary key (PK) of the relation will be augmented with the primary key attributes. Note that if the CIS contains attributes from multiple relations with 1-to-1 relationships, then it suffices to augment the CIS with the primary key attribute(s) of the first relation if necessary. This new set is called the augmented closed item sets (ACIS).

In Step 4, we use a branch and bound type algorithm that examines all clustering solutions of attributes such that a solution contains at least one cluster that is an ACIS. The solution with the lowest is the one selected as our next vertical clustering of attributes. The query response time is not an accurate measure of the cost of a query since it varies with the system load, buffering, and indexing. The cost of running a query will therefore be given by the estimated I/O cost returned by the query optimizer.

A. Step 1: Build the Frequency-Weighted Attribute Usage Matrix

AutoClust mines the closed item sets found in the queries. The affinity, or co-access, between attributes is stored in an attribute affinity matrix. For instance the following query would produce the first row of the affinity matrix in Table I.

SQL QUERY: SELECT a, c, d FROM T1
WHERE a BETWEEN 1 AND 10 AND d=c

The attribute usage matrix contains a row for each query considered by our technique and a column for each attribute in the database. If a query requests a particular attribute, the intersection of the query row and the attribute column will contain a "1", and "0" otherwise. Building this matrix is trivial and only requires scanning each query run once. The matrix can be updated at will by adding and removing rows based on current database usage.

TABLE I
EXAMPLE OF AN ATTRIBUTES USAGE MATRIX

Queries	Attributes						Query frequency (%)
	A	B	C	D	E	F	
q_1	1	0	1	1	0	0	10
q_2	1	1	1	0	1	0	20
q_3	0	1	0	0	1	0	30
q_4	0	1	1	0	1	0	40

The attributes in this example are defined in Table II.

TABLE II
EXAMPLE OF AN ATTRIBUTES INFORMATION TABLE

Attribute Name	Attribute Database Table	Data Type
A (PK)	Table1	Short
B	Table1	Short
C	Table1	Long
D	Table1	Byte
E	Table1	Short
F	Table1	Text

Note that attribute A is denoted as the primary to the entire relation. For the sake of simplicity, all attributes in this example belong to the same relation. However relations having a 1-to-1 relationship could be processed the same way. For example two relations R_1 = (A, B, C, D) and R_2 = (A, E, F), where there is a 1-to-1 relationship between the primary key attributes R_1.A and R_2.A, is equivalent to a single relation R = (A, B, C, D, E, F).

Next, we multiply each row by of the attribute usage matrix by its corresponding query frequency to obtain the frequency weighted attribute usage matrix (Table III):

TABLE III
FREQUENCY WEIGHTED ATTRIBUTE USAGE MATRIX

Queries	Attributes					
	A	B	C	D	E	F
q_1	10	0	10	10	0	0
q_2	20	20	20	0	20	0
q_3	0	30	0	0	30	0
q_4	0	40	40	0	40	0

The interpretation of this matrix is as follows. 30% (10%+20%) of the queries run access the attribute 'A', 90% (20% + 30% + 40%) of the queries run access the attribute 'B', and so on. Also, and more importantly for the rest of the technique, 10% of the queries run access attributes 'A', 'C', and 'D' (from q_1). 60% of the queries run access attributes 'B' and 'C' (from q_2 and q_4). This percentage can be calculated for any attribute set and it will be used by our technique.

B. Step 2: Mining the Closed Item Sets

Attributes that are frequently queried together should be stored together. If we consider database attributes as items and queries as transactions, the problem of identifying attributes frequently queried together is similar to the data mining

association rules problem of finding frequent (also called large) item sets, which is described below.

1) Frequent Item Sets [9]: A frequent item set is an item set, which is present in a number of transactions greater than a support threshold, s. For example, from Table IV, we see that {B, C} is accessed by 60% of the queries run. Therefore the item set {B, C} has a support of 60%. The item set {A, C, D} has a support of 10%. If we set the support level threshold at 20%, {B, C} would be a frequent (or large) item set but {A, C, D} would not be. A subset of the set of frequent item sets is the set of frequent closed item sets defined as follows.

2) Closed Item sets [16]: A closed item set carries more information for it is a maximal item set contained in the same transactions. It meets the following two conditions:

All members of the closed item set X appear in the same transactions. There exists no item set X' such that:

- X' is a proper superset of X and
- Every transaction containing X also contains X'.

Mathematically, the problem is described as follows [10]:

Let D = (O, I, R) be a data mining context, O a set of transactions, I a set of items, and R a binary relation between transactions and items. For $O \subseteq O$ and $I \subseteq I$, we define:

- $f(O) = \{ i \in I \mid \forall o \in O, (o,i) \in R \}$ and
- $g(I) = \{ o \in O \mid \forall i \in I, (o,i) \in R \}$.

f(O) associates with O, items common to all transactions $o \in O$, and g(I) associates with I, transactions related to all items $i \in I$. The operators h = f∘g and h' = g∘f are the Galois closure operators (Pasquier, 1999).

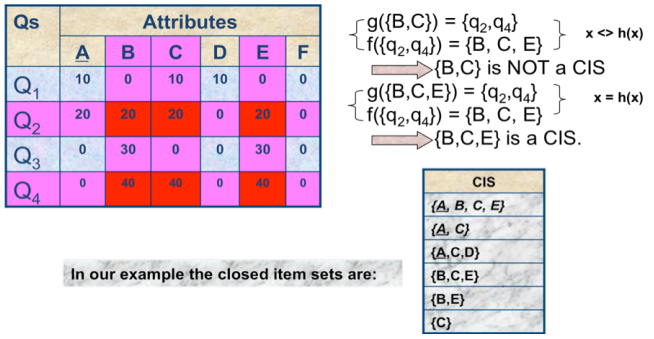

Fig. 1. Determining a closed item set

In Fig. 1, we see that:

- $g(\{B, C\}) = \{q_2, q_4\}$ and
- $f(g(\{B, C\}) = f(\{q_2, q_4\}) = \{B, C, E\}$

Since $g(\{B, C\})$ is not equal to $f(g(\{B, C\}), \{B, C\}$ is not a closed item set.

Now we consider the item set {B, C, E}.

- $g(\{B, C, E\}) = \{q_2, q_4\}$ and
- $f(g(\{B, C, E\}) = f(\{q_2, q_4\}) = \{B, C, E\}$

Since $g(\{B, C, E\})$ is equal to $f(g(\{B, C, E\}), \{B, C, E\}$ is a closed item set. By definition, this means that when attributes 'B' and 'C' are queried, attribute 'E' is always queried along with them.

This also means that it suffices to consider the closed item sets as clusters of attributes and there is no need to consider all their subsets that are frequent item sets. This greatly reduces the complexity of the clustering problem. Many algorithms for mining closed item sets exist. [16] and CHARM [20].

Using the data in Table III, the list of all items I = {A, B, C, D, E, F} and the closed item sets (CIS) and their respective support are given in Table IV.

TABLE IV
CLOSED ITEM SETS AND THEIR SUPPORT

Closed Item Set	Support (in %)
{A, B, C, E}	20
{A, C}	30
{A, C, D}	10
{B, C, E}	60
{B, E}	90
{C}	70

C. Step 3 Filtering the Closed Item Sets

Since all attributes in this example come from the same table it suffices to augment, if needed, each CIS with the primary key of the relation, i.e. attribute 'A' as shown in Fig 2.

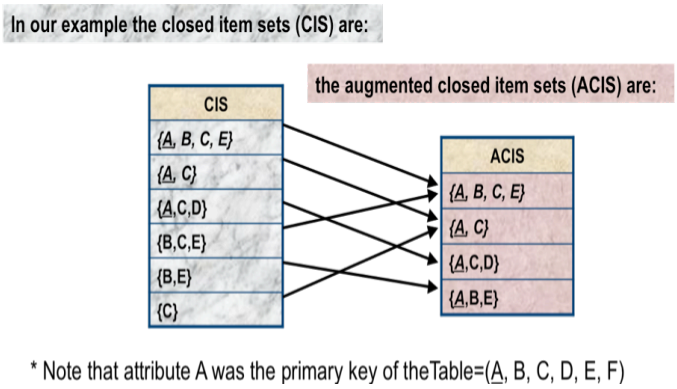

Fig. 2. Augmenting the closed item sets with the primary key ACIS ={{A, B, C, E}, {A, C}, {A, C, D}, {A, B, E}, {A, C}}.

D. Step 4: Determine the Best Clustering of Attributes Based on Closed Item Sets

We wish to partition the database vertically by clustering some attributes together. A clustering solution is therefore a partition of the set of attributes I = {A, B, C, D, E, F}. For example {{A, C, E}, {A, B, D}, {A, F}} is a clustering solution containing 3 clusters. Each cluster contains a copy of the relation's primary key for natural join purposes.

We have seen previously in Section 2 that the clustering problem has been shown to be NP-HARD. In our case, however, the search space is greatly reduced since we only consider the clustering solution containing at least one cluster that is a closed item set. The attributes not present in any set in ACIS will each be clustered in its own blocks along with a copy of the primary key.

A solution produced by our algorithm (leaves of the tree in Fig. 5) is called a candidate clustering solution (CCS) and is a complete partition of the set of items I. A complete partition of the set of items is a partition that contains every item in I. In other words, the union of all sets contained in the partition is equal to I. For a candidate clustering solution to be valid it

must contain at least one cluster that is a modified closed item set.

The algorithm given in Fig. 3 and Fig. 4 essentially starts with an empty solution and adds clusters of attributes that are ACIS to the solution until the solution forms a complete partition of the set of attributes. When the solution is complete, as shown in the leaves of the execution tree in Figure 5, its cost is measured by running all the queries in Q through the query optimizer, which estimates the I/O cost. Note that the queries are not actually run. All possible combinations of ACIS as clusters will be considered.

```
BEGIN MAIN
    // Inputs:
        // I = set of all items (attributes)
        // ACIS = Augmented Closed Item Sets.
        // PK = the set containing all the primary keys
    // Output:
        // The vertical clustering solution with the lowest cost

    // Find all attributes present in any set in ACIS
    F = All attributes present in any set in ACIS

    // Find all attributes NOT present in any set in ACIS
    NF = I – F

    // Remove all the primary keys from F since all ACIS
    // are already augmented
    F = F – PK

    // Find all candidate clustering solutions (CCS)
    //Attributes not in any ACIS are clustered separately
    FOR all attributes attr in NF DO
        // {{a,b},{c}} ∪ {d} = {{a,b},{c},{d}}
        // In our example NF={F}, hence {AF} is a cluster
        // since PK = {A} is the sole primary key.
        CCS = CCS ∪ ( PK ∪ attr )
    END FOR

    // call recursively completeCandidateSolution
    completeCandidateSolution( CCS, F, ACIS, PK)

END MAIN
```

Fig. 3. Algorithm to determine the best clustering solution

The algorithm in Fig. 3, using I = {A, B, C, D, E, F}, computes F = {A, B, C, D, E}, the set of all items present in at least one ACIS and NF = {F} the set of all items not present in any ACIS. The primary key attribute is then removed from F leaving us with the set F = {B, C, D, E}.

Next, we need to address the case of attributes that are not present in any ACIS. Since these attributes are not used in any queries, they should not be stored with any other attributes. These attributes will be stored together in a separate cluster. In our example attribute F is the only attribute not present in any ACIS and our first cluster of attribute is therefore {A, F} as depicted in Fig. 5.

Next, our algorithm calls the recursive subroutine in Fig. 4. This subroutine will scan solutions and add an ACIS to our solution at each step until the solution becomes a complete partition of I. For instance, let us examine the rightmost branch of the execution tree in Fig. 5. With the current solution being CCS = {{A, F}}, the algorithm is not looking for an ACIS that contains attribute 'B' to add to the CCS. There are 3 solutions: {A, B}, {A, B, C, E}, and {A, B, E}. Let us focus on the latter. Since there is no overlap between CCS and {A, B, E} except for the primary key which is required, {A, B, E} is then appended to CCS. Our new CCS is then CCS = {{A, F}, {A, B, E}}.

```
completeCandidateSolution( CCS, F, ACIS, PK )

    // Role:
        // Incrementally builds a clustering solution of the set of
        // attributes by adding a set of ACIS or a 1-item set to the
        // current incomplete solution

    // Output:
        // A partition of the set of attributes, complete or not.

    att = first attribute in F

    FOR each set S in (ACIS U (att U PK)) containing att DO
        IF  ( CCS U S ) contains all the attributes in I
            // then the solution is complete
        THEN
            CCS = CCS U S
            // Run every query in Q and multiply their
            // cost (EstimateIO) by their frequency and
            // compute the aggregate cost of the
            // clustering.
            // If the cost is the best save it as current best solution.
        ELSE
            // The solution is incomplete
            // We need to prune F and ACIS, then call recursively.
            remove from F all elements contained in S.
            remove from ACIS all sets containing elements in S.

            completeCandidateSolution( CCS U S, F, ACIS, PK )

        END IF

    END FOR

END completeCandidateSolution( CCS, F, ACIS, PK )
```

Fig. 4. Algorithm (cont'd)

Next, our algorithm will look for ACIS that contain attribute 'C' and do not contain any attributes already in CCS. There are two choices, {A, C} and {A, C, D}, that will create two sub-branches in our execution tree: CCS_1 = {{A, F}, {A, B, E}, {A, C}} and CCS_2 = {{A, F}, {A, B, E}, A, C, D}}.

Note that CCS_2 is a complete partition of I and, therefore, is highlighted in Fig. 5. CCS_1, on the other hand, is still incomplete and is missing attribute 'D'. There are no ACIS that contain attribute 'D' and have no overlap with the attributes already in CCS_1. Therefore, the attribute 'D' is simply augmented with the primary key and appended to

CCS_1, which gives us CCS_1 = {{A, F}, {A, B, E}, {A, C}, {A, D}}.

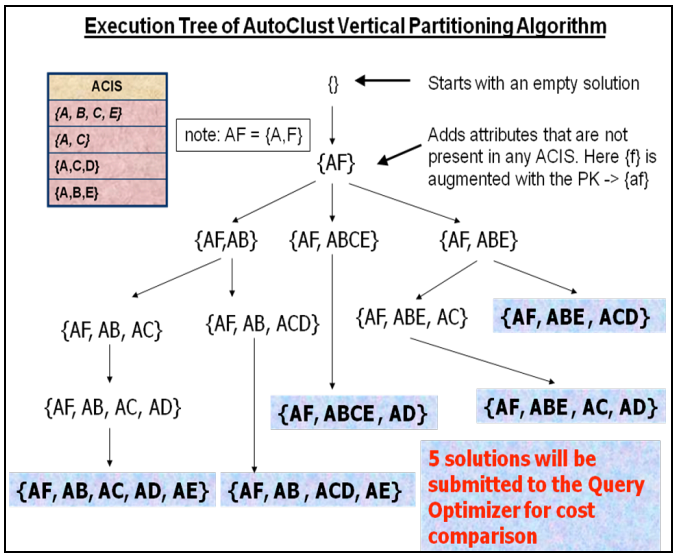

Fig. 5. Trees containing the candidate solutions

In our example, the algorithm produces the following 5 candidate clustering solutions, which are the leaves of the execution tree in Fig. 5.

- S_1 = {{A, F}, {A, B}, {A, C}, {A, D}, {A, E}}
- S_2 = {{A, F}, {A, B}, {A, C, D}, {A, E}}
- S_3 = {{A, F}, {A, B, C, E}, {A, D}}
- S_4 = {{A, F}, {A, B, E}, {A, C}, {A, D}}
- S_5 = {{A, F}, {A, B, E}, {A, C, D}}

Each clustering solution is then implemented in our sample database, which consists of 1,000,000 randomly generated records. We then estimate the I/O costs for running each query using each clustering solution. Note that the queries are not actually executed, but only estimated by the query Optimizer tool of SQL Server.

The aggregate cost in Table 5 was computed by weighing the cost of each query by its frequency. For instance, the aggregate cost of S_1 is (3.857 * 10) + (5.601* 20) + (3.121 * 30) + (5.602* 40) = 468.27 using the query frequencies given in Table 1.

TABLE V
ESTIMATED I/O COST OF CANDIDATE CLUSTERING SOLUTIONS

Solution	Estimated Individual Query Cost (I/O)				Aggregate Cost
	q_1	q_2	q_3	q_4	
S_1	3.857	5.601	3.121	5.601	468.27
S_2	2.658	5.779	3.121	5.779	466.95
S_3	4.403	3.026	3.026	3.026	316.38
S_4	3.857	4.406	1.926	4.406	360.79
S_5	2.657	4.584	1.926	4.584	359.47

E. *Analysing the results*

Our first observation is that clustering S_3 is the best overall clustering solution with an aggregate cost of 316.38 for the set of all queries. It is also worth noting that not all queries perform the best using clustering S_3. In fact only q_2 and q_4 perform the best using S_3. Query q_1 performs the best using S_2 and S_5 while q_3 performs the best using S_4 and S_5. This is a strong argument in favour of dynamic clustering. It is easy to see that a slight change in the query frequencies would tilt the balance in favour of another clustering solution.

Our tests results in [11] showed that the time required to mine closed item sets is so small that there is no need to set up frequency threshold. As a result the quality of the solution produced is the best possible. We ran simulations with a simpler model that did not require primary key duplication and where the cost was an estimated number of disk blocks accessed and found out that our technique returned a solution 57% better than OBP on average while the execution time was 500 times faster on the TPC-R benchmark data [19]. We also noted that using a simpler model, our algorithm systematically returned the same solution as a brute force algorithm that found the optimum solution. We are currently in the process of comparing AutoClust against Microsoft's solution [1] through simulation. Preliminary results show that AutoClust finds a solution, which is at least as good, and it finds it faster. An added bonus of AutoClust is that it scales up very easily and adapts very advantageously to the cluster computing paradigm, which we will discuss in the next section.

IV. AUTOCLUST AND CLUSTER COMPUTING

Autoclust can be adapted to and greatly benefit from cluster computing where computers are linked together through a network and work together as a single integrated system to improve performance and/or availability [3]. Using cluster computing in the context of a database application, we can choose between two different architectures. The first architecture would simply be a distributed database where the data are spread across the different nodes of the network with no data replication. The second architecture would have the database replicated on each node of the network.

A. *Cluster Computing with No Replication*

In the case of a distributed database, the database would be split such that each node would contain entire relations of the database. Hence if two tables have a 1-to-1 relationship then they can be stored on the same node and other tables can be stored on other nodes with no consequence. AutoClust clusters database attributes on a per relation basis (as mentioned in Section III). This means that each node can run its own totally independent instance of the AutoClust algorithm. Since the execution is independent and totally parallel, the execution time is greatly improved. Moreover, nodes can be clustered and re-clustered separately and independently at the discretion of supervisor (automated or human).

For example, imagine the following database relations:

- T_1 = (K_1, A, B, C, D)
- T_2 = (K_2, E, F)
- T_3 = (K_3, G, H)
- T_4 = (K_4, I, J)
- T_5 = (K_5, K, L, M)
- T_6 = (K_6, O, P, Q)

Let us assume further that there is a 1-to-1 relationship between the following groups of attributes:

- K_1, K_2, and K_3
- K_4 and K_5

It follows that the entire database can be split up in 3 nodes where each node would only contain tables that are linked by a 1-to-1 relationship. Hence, we would have 3 nodes containing the following data:

- Node 1: (K_1, A, B, C, D, E, F, G, H)
- Node 2: (K_4, I, J, K, L, M)
- Node 3: (K_6, O, P, Q)

The AutoClust attribute-clustering algorithm can be run independently, simultaneously or asynchronously, on each separate node. Each node will implement the best available clustering for the queries considered. The best possible clustering is what is the best for the query set as a whole but certainly not for each query individually.

The improved performance in this case is solely due to the parallel execution of the clustering process and queries.

B. Cluster Computing with Replication

In the case of a high availability clusters containing redundant nodes, the benefits are far greater. The trick is to store the same data on different nodes but in a different way. The version of AutoClust detailed in Section 3 identifies a group of possible candidate clustering solutions and test them against each other through the query optimizer to finally retain only the best one. In the case of redundant nodes we can cluster *n* nodes containing the same data using different clustering structures. The primary node would be clustered using the best possible clustering, the secondary node would use the second best clustering, and so on.

The advantage of this scheme is that it implements a different clustering solution on each node and each query can be run on a different node based on expected query response time and/or availability. In Step III.D AutoClust determines exactly which clustering is the best for each query. This clustering is not necessarily the chosen one at the end because the chosen clustering in the single node version of AutoClust is the one that works best for the sum of all queries. The advantage in having redundant nodes clustered differently is that individual queries can be sent to the node where the clustering will provide them with the best query response time.

Let us use the simulation results from Table V as an example. It is clear that S_3 is the best clustering solution for the current database workload; however S_3 does not provide the lowest estimated I/Os for all queries in the workload.

In fact the following Table VI reflects the best possible choice of clustering for each query.

Now, imagine that our cluster computing architecture contains three nodes. All three nodes contain the same database, but the database on each node is clustered differently. According to Table V, the best three clustering solutions in terms of aggregate costs are in the order of S_3, S_5, and S_4.

TABLE VI
BEST CLUSTERING CHOICES PER QUERY

Choices	Queries			
	q_1	q_2	q_3	q_4
Best Choice	S_5	S_3	S_4	S_3
2nd Choice	S_2	S_4	S_5	S_4
3rd Choice	S_1	S_5	S_3	S_5
4th Choice	S_4	S_1	S_1	S_1
5th Choice	S_3	S_2	S_2	S_2

According to Table VI, the best choice for q_1 is S_5, the best choice for q_2 and q_4 is S_3, and the best choice for q_3 is S_4. So by implementing the best three clustering solutions on the three nodes, we can ensure that each query has a chance to be run against the cluster that would provide it with the lowest expected estimated I/O cost. Using Tables IV and V, and assuming that we have 3 nodes N_1, N_2, and N_3 that implement 3 clustering solutions S_3, S_5, and S_4, respectively. Then each query would be sent to the best possible available node according to the routing table in Table VII.

TABLE VII
QUERIES ROUTING TABLE

Choices	Queries			
	q_1	q_2	q_3	q_4
Best Choice	S_5	S_3	S_4	S_3
2nd Choice	S_4	S_4	S_5	S_4
3rd Choice	S_3	S_5	S_3	S_5

Using Table VII, the component in charge of executing queries, would redirect each query to its "Best Choice" if available; otherwise, the second choice would be used.

The advantages of this architecture are many. Different queries can be run in parallel against different nodes. Each query has a chance to get rerouted towards the node that would execute it in the most efficient manner.

Another side advantage of redundant nodes when it comes to clustering is that nodes can be clustered and re-clustered asynchronously, and while a node is being re-clustered, the data are still online from another node. This is important since re-clustering data involves moving data around on disk and this can be a time-consuming task. We could almost make the case that cluster computing makes database clustering possible.

V. CONCLUSION

We presented an autonomic attribute clustering algorithm that is based on data mining techniques. The idea is to form clusters of attributes that correspond to closed item sets of attributes found in the queries. Preliminary tests results indicate that this algorithm returns an excellent quality solution in record time. We also showed how AutoClust can be adapted to fit a cluster computing environment with or without redundant nodes. In the case of redundant nodes we showed that a different clustering scheme could be implement on each node, thereby providing an optimal query response time for virtually any query to run against the database. Redundancy also allows data availability during re-clustering.

ACKNOWLEDGMENT

The Authors would like to thank Ludovic Landry and Serge Tchitembo for implementing the AutoClust algorithm and providing us with some of the simulation results used in this paper.

REFERENCES

[1] Sanjay Agrawal, V. Narasayya, B. Yang, *Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design*, SIGMOD, June 2004.

[2] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, Manoj Syamala, *Database Tuning Advisor for Microsoft SQL Server 2005: Demo*, SIGMOD 2005, June 2005.

[3] Mark Baker, Rajkumar Buyya, *Cluster Computing at a Glance, Chapter 1*, High-Performance Cluster Computing, Architectures and Systems, Vol. 1, Prentice-Hall, 1999.

[4] http://mathforum.org/advanced/robertd/bell.html. Accessed 8/15/2008.

[5] Phil Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Frankiln, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman, *The Asilomar Report on Database Research*, ACM SIGMOD Record,,Vol. 27 , Issue 4 , Dec. 1998.

[6] Surajit Chaudhuri and V. Narasayya, *AutoAdmin What-if" Index Analysis Utility*, SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 1998.

[7] Surajit Chaudhuri and Vivek Narasayya, *Self-Tuning Database Systems: A Decade of Progress*, VLDB 2007, Proceedings of the 33rd International Conference Very Large Databases, September 2007.

[8] 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.

[9] Margaret H. Dunham, *Data Mining: Introduction and Advanced Topics*, Prentice Hall, 2003.

[10] Nicolas Durand and B. Cremilleux, *Extraction of a Subset of Concepts from Frequent Closed Itemset Lattice: A New Approach of Meaningful Clusters Discovery*, International Workshop on Advances in Formal Concept Analysis for Knowledge Discovery in Databases, July 2002.

[11] Sylvain Guinepain and L. Gruenwald, *Automatic Database Clustering Using Data Mining*, **DEXA '06:** Proceedings of the 17th International Conference on Database and Expert Systems Applications, September 2006.

[12] Erez Hartux, and Ron Shamir, *A Clustering Algorithm Based on Graph Connectivity*, Information Processing Letters, Vol. 76, No. 4-6, 2000.

[13] McCormick, W. T. Schweitzer P. J., and White T. W., *Problem decomposition and data reorganization by a clustering technique*, Oper. Res. 20, 5, September 1972.

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

[15] Stratos Papadomanolakis, Debabrata Dash, Anastasia Ailamaki, *Efficient Use of the Query Optimizer for Automated Physical Design*, VLDB 2007, Proceedings of the 33rd International Conference Very Large Databases, September 2007.

[16] Nicolas Pasquier, Y. Bastidem, R. Taouil, and L. Lakhal, *Efficient Mining of Association Rules Using Closed Itemset Lattices*, Information Systems, Vol. 24, No. 1, 1999.

[17] 2nd International Workshop on Self-Adaptive and Autonomic Computing Systems. DEXA 2004.

[18] 3rd International Workshop on Self-Managing Database Systems, ICDE 2008.

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

[20] Mohammed J. Zaki and C. Hsiao, *CHARM: An Efficient Algorithm for Closed Itemset Mining*, SIAM International Conference on Data Mining, April 2002.