International Journals of Combined Research & Development
eISSN:2321-225X,pISSN:2321-2241,Vol:5;Issue:6 ,June- 2016

Clustering With An Auto-Indexing for Tuning Databases

KAVITHA S N
Assistant Professor
Department of MCA
New Horizon College
Of Engineering
Bengaluru – 560103
Karnataka

JINCY C MATHEW
Assistant Professor
Department of MCA
New Horizon College
Of Engineering
Bengaluru – 560103
Karnataka

BINOJ M
Assistant Professor
Department of MCA
New Horizon College
Of Engineering
Bengaluru – 560103
Karnataka

Abstract : *In opinion of the wide association of databases and its size, mainly in data warehouses, it is important to systematize the physical design so that the task of the database administrator (DBA) is minimized. A vital part of physical database design is index selection. An auto-index selection tool capable of exploring large amounts of data and suggesting a good set of indexes for a database is the goal of auto-administration. Clustering is a data mining technique with broad request and usefulness in exploratory data analysis. This idea provides a motivation to apply clustering techniques to obtain good indexes for a workload in the database. In this paper we describe a technique for using clustering auto-indexing. The experiments showed that the proposed technique performs better than Microsoft SQL Server Index Selection Tool and can suggest indexes faster than Microsoft's IST.*

Keywords: DBA, Auto Indexing

1. Introduction

Given a relational database system and a workload of queries that signifies a sample of transactions done in a database, the Index Selection Problem (ISP) contains selecting a set of index configurations for each table so that the cost for processing the workload is minimum subject to a edge on the total index space [2]. Since all indexes have a preservation cost during update, insertion and deletion, we cannot indefinitely increase the number of indexes on a database table. The Index Selection Problem has been approached differently by diverse researchers to build Index Selection Problem. We can classify tools that address the Index Selection Problem based on their approach in two ways. The first category is external tools which use linear programming optimization techniques and other cost minimization techniques to resolve the Index Selection Problem [8]. Some external tools have also used data mining techniques to solve the Index Selection Problem [1]. The second category is the tools that consume the query optimizer to give cost estimates for various index structures and suggest a structure with the least cost estimation. [2], [3].

A disadvantage with the first category is that the tool is detached from the optimizer. This means that there could be some indexes suggested by the tool which are not used by the optimizer while handling the workload. The presence of such indexes will be an overhead on the DBMS [2].. A second drawback is that these tools are based on the current knowledge of the strategy used by the optimizer and will become outdated as the optimizer changes.

The second approach has the advantage that all indexes are chosen by the optimizer and will be used by the optimizer while handling the workload [3]. However, this approach requires many optimizer calls because many possible index configurations have to be evaluated by the optimizer. This means higher index suggestion time and longer processing time for other applications using the DBMS when indexes are being suggested.

The idea behind our research is to chain the two approaches so that the major part of the solution to the Index Selection Problem is done externally and also use the optimizer to choose the final set of indexes. In our technique the optimizer is invoked only once for each query in the workload to choose the final set of indexes from a set of externally resolute index configurations. Also most of the existing external tools address only

single column and non-clustered indexes. Our tool has the capacity to suggest a set of single-column and multi-column indexes as well as clustered and non-clustered indexes. We refer to [4] for the definition of these terminologies.

The repose of the paper is organized as follows. In Section 2 we describe our index selection technique. In Section 3 we describe re-indexing. In Sections 4 and 5 we discuss our research and results, respectively. In Section 6 we discuss conclusions and future work.

2. Proposed Index Selection Technique

Our suggested technique is based on the perception that the attributes that occur more frequently and frequently in a group of similar queries are likely to be useful for indexing[2] [4]. Based on this idea we group queries which are similar in terms of their use of attributes. Attributes which are accessed by all the queries in each group are mined as indexes. These indexes can be single-column or multi-column indexes. For multi-column indexes, the order of the columns is firm by assigning weights to attributes based on whether they are used in a search argument, join clause, GROUP BY/ORDER BY clause or aggregate function. A clustered index is also chosen by assigning weights to the attributes liable on whether they occur in range queries, join clause or GROUP BY/ORDER BY clause. A detailed clarification is given in Section 2.1. These indexes which are extracted are then submitted to the query optimizer for final selection for the given workload. The indexes not selected by the optimizer are eliminated. The remaining indexes are the final indexes suggested by our tool.

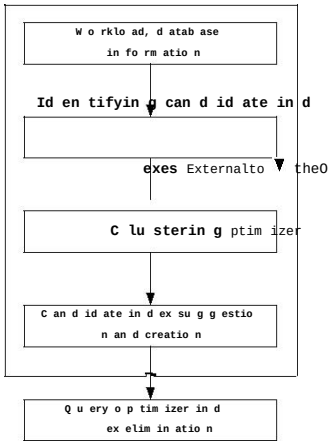

Figure 1

The phases of the proposed auto-indexing technique

Figure 1. shows the many phases of our method. In Section 2.1 we describe the identifying candidate indexes phase where we citation candidate indexable attributes, classify the ordering of multi-column indexes and identify clustered and non-clustered indexes. In Section 2.2 we describe the clustering phase where queries based on attributes are grouped together. In Section 2.3 we discuss the candidate index proposal and creation phase, and in Section 2.4 we deliberate the query optimizer index elimination phase.

2.1 Identifying candidate indexes

During this stage a workload of queries is in use as input, indexable attributes are removed and a query-attribute matrix [1] is created. While extracting indexable attributes we also consider columns in collective functions such as MIN, MAX, SUM, AVG and COUNT as indexable attributes because non clustered indexes can also be created on columns existing in aggregate functions [4].

In a query-attribute matrix the occurrence of an indexable attribute in a query is showed by a 1 and absence by a 0 [1]. An example of a query-attribute matrix is shown in Figure 2. Let columns A and B belong to a table named T1 having 20 rows and C, D and E belong to a table named T2 having 15 rows in Figure 2.

Queries	Indexable attributes				
	T1.A	T1.B	T2.C	T2.D	T2.E
Q1	1	0	1	1	0
Q2	0	1	1	0	1
Q3	1	1	1	0	1
Q4	0	1	0	0	1
Q5	1	1	1	0	1

Figure 2. Query-attribute matrix [1]

A query-frequency matrix is created during this phase to extract aspirant indexable attributes if their frequencies (*Freq*) satisfy equation (1).

$$Freq > threshold1 \text{ OR } Freq * T > threshold2 \quad (1)$$

In equation (1) *Freq* is the frequency of each indexable attribute in the workload and *T* is relative to the size of the table in rows to which the column goes. *threshold1* removes the attributes that do not occur very regularly in the workload and *threshold2* eliminates the attributes that do not belong to large tables except they

occur very frequently. Both the threshold values are mechanically added by the tool and can also be supplied by the user of the tool.

An example of a query-frequency matrix is shown in Figure 3. In a query-frequency matrix the 0's and 1's of query-attribute matrix are replaced by the frequency of the attributes happening in the query. Let *threshold1* be 5 and *threshold2* be 100 then the candidate indexable attributes are T1.B, T2.C and T2.E in Figure 3. The attributes A and E are removed from the query-attribute zmatrix. It is worth mentioning here that the frequency of a query in the workload is automatically taken care of by our technique. If a query appears many times in a workload, its corresponding attributes will occur many times in the query frequency matrix. As a result the chance of these attributes being selected up as candidate indexes increases.

Queries	Indexable attributes				
	T1.A	T1.B	T2.C	T2.D	T2.E
Q1	2	0	1	3	0
Q2	0	1	2	0	1
Q3	1	1	3	0	1
Q4	0	2	0	0	3
Q5	1	4	2	0	2
Freq	4	10	9	3	8
*Freq * T*	80	200	135	45	120

Figure 3. Query-frequency matrix

Ordering the columns constructed on our instinct that those columns occurring in a WHERE clause should be given higher priority to be chosen as an index than those columns which occur in GROUP BY or ORDER BY clauses and the least importance should be given to columns occurring in aggregate functions. According to the priorities, weights of 3, 2 and 1 are given to the columns occurring in a WHERE clause, GROUP BY or ORDER BY clauses and aggregate functions, respectively[5]. The total weight of an attribute in the workload is found and the attributes are ordered in downward order of weight from left to right in the query-attribute matrix. The attributes stirring with high frequency query will have more weight and eventually will be placed on the left.

The selection of clustered indexes is also done during this phase. Though clustered indexes cause an overhead they are also helpful to certain queries. We choose to create single-column clustered indexes in order to reduce overhead. Since clustered indexes should be created on columns happening frequently in range queries we assign more weight for range queries and the same weight for join clause and GROUP BY or ORDER BY clauses. The total weight of attributes in the workload is found and a higher rank is assigned to an attribute with a higher weight. We make the clustered indexes as selective as possible by also considering the rank of attributes according to selectivity. The higher the discrimination of an index, the higher is the possibility of the index being chosen by the optimizer to execute a query [4]. Clustered indexes are automatically created on primary key columns and for a primary key column selectivity is equal to 1. Therefore columns with selectivity equal to 1 are not considered for clustered index however, they are considered for non-clustered index. The sum of the rank with discernment and rank with weights is calculated. The column with the highest sum for each table is chosen as the clustered index for that table. If two or more columns have the same sum then the column with a higher weight rank is chosen. This is because attributes occurring in range queries and having duplicate values should be given more importance than columns with higher selectivity.

2.2 Clustering

Output of the identifying candidate index phase is the query attribute-matrix containing the ordered candidate indexable attributes. This query-attribute matrix is the input for the clustering phase. Our area is to group queries in a workload based on common attributes occurring in the query using the query attribute matrix in Figure 2. During the clustering phase queries that are similar based on common and frequently occurring attributes are clustered together. A possible clustering result from Figure 2 is [Q1], [Q2, and Q4] and [Q3].

2.3 Candidate index suggestion

During this phase, those candidate indexable attributes which are mutual to all the queries clustered together during the clustering phase are suggested as indexes. For example from Figure 2, the suggested index configuration will be the index [T2.C, T2.D] for cluster [Q1], indexes [T1.B] and [T2.C] for cluster [Q3, Q5] and index [T1.B] for cluster [Q2, Q4]. Note that attributes T1.A and T2.E are not candidate indexable attributes. The order in the query-attribute matrix is maintained while creating multi-column indexes. These suggested indexes are then existing to the optimizer for final selection.

2.4 Query optimizer index elimination

Our idea behind this phase is the presence of an optimizer capable of choosing from a set of virtual or theoretical indexes that outputs its choice and cost estimate for each query [7]. The optimizer uses its statistics and cost estimations to choose indexes for each query. In the absence of an optimizer which is capable of choosing from a virtual set of indexes in SQL Server we actually create indexes suggested from Section 3.3. Then we invoke the optimizer to find out the indexes estimated to be used to execute the workload[8]. Those indexes not being picked up by the optimizer are released because the presence of these unused indexes will cause an overhead of space and conservation in the database. The remaining indexes in the database are the final indexes suggested by our technique.

3. Re-Indexing

hich are part of new but not part of existing set are created, those which are part of existing set and not in new set are dropped and those which interconnect remain[9]. The process of dropping and creating indexes in the system follows similar methodology as Oracle's Automated Index-Rebuild System [12] which can be done either online or offline.

4. Experiments

We have conducted experiments on Microsoft SQL Server 2000 [4] using the decision support TPC-R benchmark [9]. We have created TPC-R's 1 GB database and have used 22 read-only queries from the benchmark to create a workload of 240 query cases[11]. The 22 read-only benchmark queries are exponentially distributed in the workload.

Our experiments use the *k*-Means [5], [6] clustering algorithms. The *k*-Means clustering algorithm accepts a parameter *k* from the user which is the final number of clusters for a group of observations. It is a well-established clustering algorithm and has been used successfully in many applications. KEROUAC is a categorical data clustering algorithm and the final number of clusters is mechanically found in this algorithm. KEROUAC also accepts a parameter from the user known as the granularity factor which determines the degree of dissimilarity among clusters. Both these clustering algorithms have low computational costs and are advantageous to us to reduce the index suggestion time.

We compare the performance of our technique with the baseline case where no indexes are created as well as with the Frequent Item sets Mining technique [1]. This technique uses the Close algorithm [11] to extract maximal set of items (attributes) that are common to a set of transactions and their support. Those item sets satisfying a minimum support are advised as indexes. The measure for comparison that we use is the average query response time in minutes. We also compare our technique with Microsoft SQL Server's IST using its thorough tuning feature and no limitations on workload size and available disk space.

5. Experiment results

All our experiments are conducted on the system Intel Pentium 4-M, CPU 2.0GHz, 512 MB RAM. The results of the experiments conducted with *k*-Means and KEROUAC are shown in Figures 4 and 5, respectively. In Figure 4 the number of clusters *k* is depicted on the *x*-axis, and the average query response time on the *y*-axis. In Figure 5 the value of is showed on the *x*-axis and the average query response time on the *y*-axis. In [1] it is reported that using the 22 read-only queries of TPC-R benchmark, the performance improvement of the frequent item sets mining technique when compared with the case of no indexes is from 15% to 25%. In Figures 4 and 5 the performance improvement of 25% for [1] is shown with a straight line. Both the figures also show the average query response time when no indexes are present and when SQL Server's recommended indexes are present in the database.

While performing experiments we observed that the average query response time varies with the choice of threshold values and that the choice of the threshold values should be such that a extensive number of indexable attributes are eliminated but not many. We experimented with different values for *threshold1* and *threshold2*. The performance is best when *threshold1* is kept close to 50% of the size of the workload. In Figures 4 and 5 *threshold1* is equal to 50% of our workload size. In our experiments the value of *threshold2* was varied for a low (20), medium (60), and high (100), and the best performance was achieved with *threshold2* equal to medium (60). Our tool chooses *threshold2* such that indexes are considered on tables with a relatively large number of rows (about 120,000). However, the DBA can also set these threshold values.

Our results show that the auto-indexing tool is sensitive to the parameters of the clustering algorithms such as *k* for *k*-Means and for KEROUAC. The similarity

of the queries in a cluster increases with the values of these parameters and better results are achieved. After increasing these parameters up to a certain value, the results do not improve further. This is expected because at this point, the same queries are clustered together, and increasing the value of the parameters cannot improve the result of clustering further. On the other hand, the lower the value of these parameters, the lower is the comparison of queries in each cluster, and therefore, the poorer are the results and performance. Clearly, the choice of these parameters is very important. Our tests indicate that if the value of these parameters is close to the number of distinct queries in the workload, a good presentation is achieved. The performance of both the clustering algorithms is the same in the best case and in the worst case[13]. As for the average case, the performance critically depends on the choice of the parameter of the clustering algorithm. Our tool automatically computes the value of this parameter so that it works within the best performance range.

While calculating the performance of our technique we have considered the parameter values within the best performance range because our tool computes the clustering limits and thresholds such that it can operate in the best performance range. When compared with the case of no indexes, the performance improvement using *k*-Means clustering is 78.89% and that using KEROUAC is 79.95%. When comparing with the Frequent Item sets Mining technique for index selection in [1], the performance improvement using *k*-Means is 71.43% and that using KEROUAC is 73.26%.

When compared with Microsoft IST the performance improvement with KEROUAC is 21.5% and that with k-means was 16.2%. For a workload of 240 queries the index suggestion time by Microsoft IST was about 8 minutes whereas our tool suggested it in less than 2 minutes.

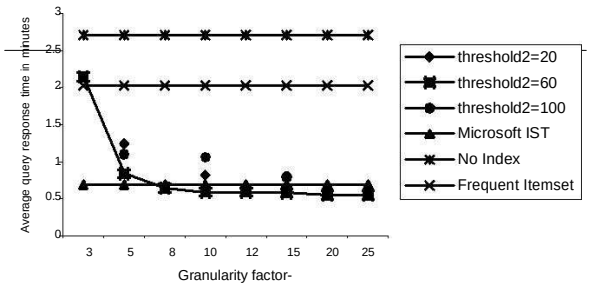

Figure 5. Results with KEROUAC clustering.

6. Conclusions and future work

Our technique is modest and requires very little info on the part of the DBA. For example, two parameters that our tool would essential are the size of the tables in the database and *threshold2*. The size of the tables can be easily recovered from any DBMS, and the DBA can provide the value of the thresholds within the recommended best ranges or can accept the value which is provided by the tool. This technique will help reduce the functions and difficulty of a DBA of a large database to choose a good set of indexes for a workload of queries. Also this technique has the pro that it can be used with any database having an optimizer gifted of outputting its choice of indexes for a given workload. By using clustering algorithms we are able to directly extract single-column and multi-column indexes instead of the iterative procedure followed by [3] which takes longer time to suggest indexes. The Frequent item sets mining technique [11] does not use the optimizer and suffers from the disadvantages of outside tools. As for the performance of the indexes, our results show that we obtain better performance than [11] and IST.

Our trials show encouraging results. However, we plan to test the requirement of our technique on different clustering algorithms, different sizes of workload, various threshold values, by assigning unlike weights and with unlike frequency distributions of the workload. So far we have used read-only queries. We plan to further carry out our experiments with UPDATE, INSERT and DELETE queries in the workload[14]. We would also like to compare our technique with ORACLE and DB2.

7. References

[1] K. Aouiche, J. Darmont and L. Gruenwald, “Frequent itemsets mining for database auto-administration”, *Seventh International Database Engineering and Applications Symposium (IDEAS’03), July 16-18, 2003.*

[2] S. Finkelstein, M. Schkolnick and P. Tiberio, “Physical Database Design for Relational Databases”, *ACM Transactions on Database Systems (TODS), Volume 13, issue 1 (March 1988), Pages 91 – 128, 1988*

[3] S. Chaudhari and V. Narasayya, “An efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server”, *Proceedings of the 23rd Very Large Data Base Conference, 1997.*

[4] R. Rankins, P. Bertucci and P. Jenson, “Microsoft SQL Server 2000”, *SAMS Publishing, Second Edition, UNLEASHED, Chapter 34.*

[5] M. Adenberg, “Cluster Analysis for Applications”, *Academic Press, , 1973.*

[6] P. Jouve and N. Nicoloyannis, “KEROUAC: an Algorithm for Clustering Categorical Data Sets with Practical

Advantages"

[11] Mike Hordila,"Setting up an Automated Index Rebuilding System".http://www.oracle.com/oramag/webcolumns/2001/auto_index.html.

[8] A. Capara, M. Fischetti and D. Maio, "Exact and Approximate Algorithms for the Index Selection Problem in Physical Database Design", *IEEE Transactions on Knowledge and Data Engineering, 7(6):955-967, December 1995*

[9] "TPC Benchmark R", *(Decision Support) Standard Specification, Revision 2.1.0, Transactions Processing Performance Council (TPC), 1993 – 2002*

[10]N. Pasquier, Y.Bastide, R.Taouil, and L.Lakhal, "Efficient Mining of association Rules using closed itemset lattices." *InformationSystems:25-46,1999.*

[12] G. Valentin, M. Zuliani, D. Zilio and G. Lohman, "DB2 Advisor: An optimizer Smart Enough to Recommend its Own Indexes", Int. Conf. on Data Engineering, March 2002.