Parallelization of Matrix Clustering Algorithms

Galaktionov V. A., viacheslav.galaktionov@gmail.com

Abstract

Matrix clustering is one of the oldest approaches to the vertical partitioning problem. Its main idea is to represent workload as a Boolean matrix and permute its rows and columns to obtain a block-diagonal form. After that it is easy to distinguish fragments into which the table should be divided.

Several of such algorithms are considered. All of them are based on a branch and bound paradigm. The more advanced ones are computationally demanding which can be addressed using a simple parallelization technique.

In this paper we demonstrate one such technique implemented using TBB library. By allowing the algorithm to traverse the tree in parallel we significantly decrease computation time while preserving the quality of the end result. We also report preliminary experimental results.

Introduction

Vertical partitioning problem is one of the oldest problems in the database domain [13]. Its most general formulation is the following: for a given relation and a given workload find a set of vertical partitions which would minimize the cost of workload processing. The workload is a set of queries and their relative frequencies.

There are a lot of different formulations of this problem, many of them feature additional information and constraints. For example, the solution should fit into the specified storage bound or the solution should contain exactly K partitions.

Just like the other physical design problems all of the non-trivial formulations for vertical partitioning problem were proved to be NP-hard [11, 14, 2]. Thus, all of the existing approaches are heuristic ones. There are dozens of approaches [7], one may classify them into two groups: affinity-based [11] and cost-based [1].

Despite the formidable age of this problem it still interests the research community. Recently, several technologies have heightened the interest of researchers: in-memory databases, big data and cloud applications.

In this paper we consider a particular subclass of affinity-based approaches called matrix clustering approaches. We design a parallelization scheme for branch and bound matrix clustering approach.

Vertical Partitioning: an Example

Suppose we are given a relation $R(a, b, c, d, e)$ that is represented in our database as a table with 5 columns. Having used just this one table for some time, we've been able to gather some statistics as to what queries were used and how often each one was executed. Using that we can partition the table into several to improve performance.

More specifically, suppose there were only 3 queries that used subrelations (a, c), (b, d) and (e) respectively. Obviously, it's better to partition the table accordingly since that would mean less data to be read into main memory.

If several queries used the same column, there would be several ways to handle that:

- Place the only copy of that attribute in a subrelation that is accessed more often.
- Duplicate it across all subrelations that use it.
- Dedicate one or more subrelations for such properties.

The choice has to be made by the system's administrator.

Matrix Clustering Approach

Matrix clustering approach is the earliest approach to this problem, it is known since the 70s. The first and the most well-known algorithm is BEA (Bond Energy Algorithm) [8]. It was followed later by a number of successors [4, 5, 6, 12].

The general idea of this approach is the following:

- having obtained a workload, treat it as an Attribute Usage Matrix (AUM). The matrix is constructed as follows: a value at position (i, j) is 1, if transaction (query) i accesses attribute j and 0 otherwise.
- Then, cluster this matrix by permuting its rows and columns to obtain block-diagonal matrix form. The resulting blocks would be the partitions.

This approach was largely superseded in 90s by the cost-based methods[1]. These methods provided more complex models which allow to drastically improve the quality of the obtained solutions. These models feature variables representing access costs, query plan operator specifics and many other details. Despite the loss of popularity, matrix clustering still has some appeal[9, 10] in applications where the response time is critical: dynamic partitioning and big data applications.

We have considered some of the more recent matrix clustering algorithms developed by Cheng et al. [4, 5, 6]. These are based on the branch and bound paradigm, whose parallelization has been well studied. This paper focuses on [6] since it is the most computationally demanding one and the most promising in terms of parallelization results.

The initial matrix is put in a root of a tree that will contain all potential solutions. Child nodes are generated by removing individual accesses from a column that prevents further clustering. This tree is traversed in a depth-first order to find a solution that would have the smallest number of removed accesses.

All nodes are characterized by cohesion of their submatrices. Cohesion can be defined as a ratio of the number of “1” entries to the number of “0” entries. This ratio is provided as an argument. A node with a smaller cohesion cannot represent a final solution and requires further clustering.

Two other algorithms operate in a similar fashion but have a different branching strategy: whole columns are removed instead of individual accesses. They also use the number of rows in each submatrix instead of cohesion.

Parallelization

There are two classic approaches to parallelization of branch and bound algorithms [3]:

- Node-based strategies that aim to improve the performance of the processing of each node of the solution tree.

- Tree-based ones that are used to improve the speed at which the tree is explored.

All considered vertical partitioning algorithms deal with a tree consisting of a great number of nodes which are computationally easy to process. For that reason the tree-based approach is chosen. The idea is to allow multiple threads to traverse the tree of solutions simultaneously.

Every generated node is put in a queue that can be accessed by any other idle thread. Designing the system this way ensures that no worker would be idling. The proposed parallelization technique does not affect the quality of the final result: the resulting clustering may not be the same as if we used the single-threaded version but would have the same number of removed accesses.

To develop a program that implements parallel versions of studied algorithms the Threading Building Blocks (TBB)[1] library was used. This choice is justified by the fact that TBB provides a task-based scheduler which features load balancing.

Another reason to choose TBB is it's ease of use. An already existing sequential implementation was parallelized with minimal effort. The parallel version was obtained by replacing the explicit stack used in depth-first traversal with TBB constructs[2]. This allowed us to keep the node inspection code unchanged.

Experiments

The experiments were conducted on a Lenovo Y580 laptop with the following specifications and software: Intel® Core™ i7-3630QM (4 physical cores, hyper-threading enabled) and 8GB (DDR3) RAM, Gentoo Linux (4.1.12 kernel version), GCC 4.9.3, TBB 4.3.20150611.

Each experiment was run 25 times, recording the time taken and a total number of nodes processed. Results are available in Table 1. The columns contain the number of threads used and minimal, average and maximal values of the number of nodes and the time taken in seconds.

The first row contains the measurements obtained after running a sequential version of the program. All subsequent rows represent the results of running a parallel version with different numbers of threads.

Having run the preliminary experiments, we have found the following: even the single-threaded versions of these algorithms are very sensitive to the initial order of columns. Different orders result in different traversal patterns. In its turn, these patterns vary greatly in terms of the number of visited nodes. In some cases their number may differ by an order of magnitude. Thus, we had to adopt a special methodology during the testing.

Matrix A9 from [6] was used as input data in our experiments. The initial 8x10 matrix had its rows shuffled randomly for each experiment. By

[1]`https://www.threadingbuildingblocks.org/`

[2]`https://www.threadingbuildingblocks.org/docs/help/reference/task_scheduler.htm`

doing so we've tried to emphasize the effect of parallelization. The target cohesion was specified to be 0.8.

#	n_{min}	t_{min}	n_{avg}	t_{avg}	n_{max}	t_{max}
seq	1180629	8.23	1585200	10.82	2388680	15.81
1	1024938	7.60	1633061	11.95	2718172	19.59
2	1034575	4.14	1695604	6.77	2693086	10.63
3	1042674	2.83	1793059	4.85	3242883	8.68
4	1013402	2.12	1748715	3.65	3667788	7.50
5	1060971	2.09	1731390	3.37	3785182	7.24
6	1021108	1.91	1578797	2.89	3930198	7.08
7	1021154	1.79	1666498	2.88	4072284	6.95
8	1022199	1.71	1689559	2.78	4452644	7.22

Table 1: Parallelization effects

It is easy to see that parallelization significantly increases the average performance. The amount of work done, however, is not so monotonic. The number of nodes processed varies greatly among runs.

Conclusions

Using a simple programming technique we've developed a parallel version of an already existing algorithm for matrix clustering. The exact effects are yet to be carefully studied.

Although a performance improvement was certainly achieved, due to the nondeterministic nature of parallel tree traversal it is hard to predict how fast the program will finish. Another reason for this behavior is the algrorithm dependency on the exact structure of the input matrix. By permuting rows or columns we can drastically change the amount of work required to decompose it. That is a feature of the clustering algorithm itself and not of the parallelization technique.

References

[1] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In SIGMOD '04, pages 359–370, 2004.

[2] P. M. G. Apers. Data allocation in distributed database systems. *ACM Trans. Database Syst.*, 13:263–304, 1988.

[3] D. A. Bader, W. E. Hart, and C. A. Phillips. *Tutorials on Emerging Methodologies and Applications in Operations Research: Presented at Informs 2004, Denver, CO*, chapter Parallel Algorithm Design for Branch and Bound, pages 5–1–5–44. Springer New York, NY, 2005.

[4] C. Cheng. Algorithms for vertical partitioning in database physical design. *Omega*, 22(3):291–303, 1994.

[5] C.-H. Cheng. A branch and bound clustering algorithm. *Systems, Man and Cybernetics, IEEE Transactions on*, 25(5):895–898, 1995.

[6] C.-H. Cheng and J. Motwani. An examination of cluster identification-based algorithms for vertical partitions. *Int. J. Bus. Inf. Syst.*, 4(6):622–638, 2009.

[7] G. Chernishev. A survey of DBMS physical design approaches. *SPIIRAS Proceedings*, 24:222–276, 2013.

[8] J. A. Hoffer and D. G. Severance. The use of cluster analysis in physical data base design. In VLDB '75, pages 69–86, 1975.

[9] A. Jindal and J. Dittrich. Relax and let the database do the partitioning online. *Enabling Real-Time Business Intelligence*, volume 126 of *LNBIP*, pages 65–80. Springer Berlin Heidelberg, 2012.

[10] L. Li and L. Gruenwald. Self-managing online partitioner for databases (SMOPD): A vertical database partitioning system with a fully automatic online approach. In IDEAS '13, pages 168–173, 2013.

[11] X. Lin, M. Orlowska, and Y. Zhang. A graph based cluster approach for vertical partitioning in database design. *Data & Knowledge Engineering*, 11(2):151–169, 1993.

[12] S. Navathe et al. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9:680–710, 1984.

[13] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems (2nd Ed.)*. Prentice-Hall, Inc., USA, 1999.

[14] D. Sacca and G. Wiederhold. Database partitioning in a cluster of processors. *ACM Trans. Database Syst.*, 10:29–56, 1985.