# A Parallel Algorithm for Closed Cube Computation

Jinguo You, Jianqing Xi, Pingjian Zhang, Hu Chen
*School of Computer Science and Engineering, South China University of Technology, Guang Zhou, 510641, China*
*jgyou@126.com, csjqxi@scut.edu.cn, pjzhang@scut.edu.cn*

## *ABSTRACT

*Closed cubing is a very efficient algorithm for data cube compression proposed recently in the literature. It losslessly condenses a group of cells into one cell if these cells have the same aggregate value and preserve roll-up/drill-down semantics. Despite its importance, parallel closed cubing solutions for huge data sets are not well studied so far to the best of the authors' knowledge. This paper presents a parallel closed cube construction and query algorithm over low cost PC clusters using the MapReduce framework. In addition, we proved that with the number of data blocks increases, the closed cubes' storage size decreases gradually. Thus users can specify the number of data blocks to balance the performance between cubes storage and query time. Experimental study demonstrates that our algorithm is efficient and scalable.*

**Keywords**: OLAP, parallel computation, closed cube, MapReduce, Hadoop.

## 1. INTRODUCTION

It's well recognized that data cubing often produces huge outputs and consumes large amount of time. For a given raw data set with *d* attributes (dimensions), the full cubing over it results in $2^d$ cuboids (group-bys) and even more cells. Correspondingly, it costs more query answering time. This performance is much worse for data cubing over massive data.

To address this issue, many sequential or parallel approaches have been studied. In [5], Beyer and Ramakrishnan proposed the iceberg cube, which contains only cells with aggregate values greater than a minimum threshold (minimum support). To construct it, they developed BUC algorithm which, not like previous ones, proceeds bottom-up by starting from the cuboid with "all" to others. This facilitates the Apriori pruning where a cell and its all descendants are not computed if the cell does not satisfy minimum support. Lakshmanan, Pei, and Han proposed the quotient cube where a group of cells are losslessly condensed into one cell if these cells have the same aggregate value and preserve roll-up/drill-down semantics [6]. Later on, Lakshmanan et al. developed a BUC-like procedure QC-DFS to find upper bound cells [7]. Dong Xin et al. introduced a more intuitive concept of closed cube that is identical to quotient cube [1]. Raymond T. Ng et al. studied parallel iceberg cubing which is not the subject of the paper [8]. Ying Chen, Frank Dehne parallelized pipesort algorithm which seldom considered the compression of the data cube [2, 9].

Recently, Jeffrey Dean and Sanjay Ghemawat in Google Corporation proposed a parallel framework, MapReduce, which automatically parallelizes and executes programs as users specified the appropriate map and reduce tasks, while other common work such as data partitioning, execution scheduling, failure tolerance etc., are handled by the MapReduce system. Since closed cube is a very efficient sequential algorithm and MapReduce simplify the real world tasks parallelism, how about the combination of their advantages?

In this paper, we present a novel parallel solution for the construction and query answering of closed cube using the MapReduce framework. Our approach can obtain more compression ratio than directly generating a closed cube over the overall data set. Further, the MapReduce framework makes the implementation usable and practical. To the best of our knowledge, this is the first paper about parallelizing closed cube computation by MapReduce framework.

The paper's main contributions are summarized below:

(*i*) We prove that as the number of the closed cubes increases, the total size of all the closed cubes goes down sharply. Meanwhile, the query answering time tends to rise slightly. Since each data block generates a closed cube, users can specify the number of data blocks to control the balance between storage and time for closed cubes.

(*ii*) We employ MapReduce framework to parallelize the process of construction and query of closed cubes. It's a simple but efficient approach to tackle large data sets since our system has features of much compression ratio, parallel tasks automatically process and failure tolerance etc.

## 2. PRELIMINARY

* The research is sponsored by the Science & Technology Project of Guangdong Province(NO.2006B11301001) and the Science & Technology Project of Guangzhou City(NO. 2006Z3-D3081)

978-0-7695-3131-1/08 $25.00 © 2008 IEEE
DOI 10.1109/ICIS.2008.63

Let $r$ be a base table over the database schema $R$. Attributes of $R$ are divided into two groups: *(i)* $D$ the set of dimensions and *(ii)* $M$ the set of measures. $ALL$ (or *) represents a set in $D$, i.e., the set over which all the aggregate values were computed. A cell with one measure is denoted by $(a_1, a_2, ..., a_n : m)$ where $a_i$ is a value or * in dimension $D_i$ and $m$ is a measure.

Given two cells $u$, $v$, we denote $u \geq v$ if $\forall A \in D$, $u[A] \neq ALL \rightarrow u[A] = v[A]$. When $u \geq v$, we say $u$ generalizes $v$ and $v$ specializes u. A cell $c$ is closed if there doesn't exist another cell $c'$ such that $c' < c$ and $c'$ has the same measure aggregate value as $c$. A closed cube is a data cube consisting of only closed cells.

As described in [1, 4], measures that can be computed incrementally fall into two categories:

Distributive Measure: A measure is called distributive if the whole data set can be computed solely based on the measures of the subsets of that data set. Sum(), Count(), Max(), Min() are all distributive.

Algebraic Measure: A measure is called algebraic if the whole data set can be computed based on a bounded number of measures of the subsets of that data set. Avg() is a common example of algebraic measures.

**Definition 1**: A local closed cube is a closed cube that is constructed from a data block. Correspondingly, an overall closed cube is a closed cube constructed over the whole base table.

Obviously, the result derived by applying the distributive / algebraic measure to n aggregate values queried from n local closed cubes respectively is the same as that derived by directly querying from the overall closed cube.

**Definition 2**: The size of a closed cube is measured by the number of all the closed cells in the closed cube.

The closed cells can be checked by tuple-based approach. Namely, it scans the tuples in the current partition (i.e. subset of the base table). If the dimension has the same value v, then v is the value in the dimension of closed cells.

**Theorem 1**: Given a base table with $T$ tuples which are divided $k$ data blocks ($1 \leq k \leq T$). Each data block generates a local closed cube. Let $S'$ the total size of all the local closed cubes and $S$ the size of the overall closed cube over the base table, we can infer that $T \leq S' \leq S$. Further, with the $k$ increasing, the $S'$ decreases.

Rationale: A closed cell is generated from a subset of the base table. Any subset of a data block is also the same subset of the base table. Accordingly, all the closed cells derived from all the data blocks are also contained in the overall closed cube derived from the base table. With $k$ increasing, the data block contains less tuples which result in less closed cells. Thus the $S'$ decreases until it reaches $T$.

E.g. Given a database schema sales(Store, Product, Season, Sale). The first three attributes are dimensions and the last attribute is a measure. Suppose that the base table is {(s1 p1 s 6), (s1 p2 f 12), (s2 p2 s 9)}. The base table is divided into 2 data blocks: {(s1 p1 s 6), (s1 p2 f 12)}, {(s2 p2 s 9)}. We can conclude that $T$=3, $S$=7, $S'$=4. It satisfies that $T \leq S' \leq S$.

# 3. PARALLEL ALGORITHM

## 3.1 Overview

The PC clusters consists of one or two namenodes and multiple datanodes. Namenodes process data partitioning, tasks schedule etc. across datanodes, while datanodes accommodate data blocks, execute map and reduce tasks.

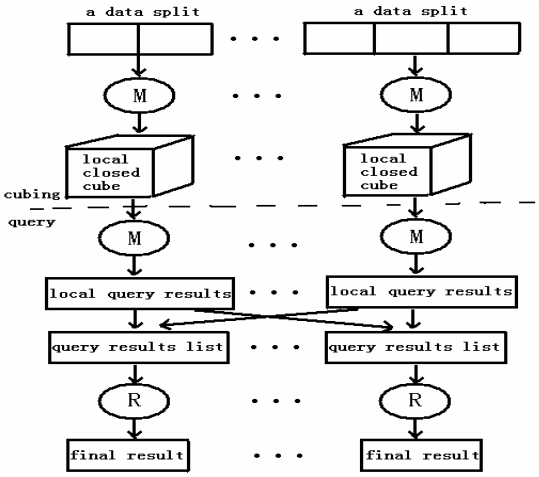


**Figure 1: Algorithm Execution Flowchart**

Figure 1 shows the flowchart of our algorithm. A map task is denoted by M and a reduce task by R in the figure. All map tasks and reduce tasks are implemented by our map and reduce procedures. There are two phases: construction phase and query phase.

During the construction phase, the massive data sets are first split into multiple blocks of almost the same size which are then dispatched to datanodes by MapReduce Framework. Also the map and reduce tasks are automatically distributed across datanodes. As soon as a data block reaches a datanode, the map task on this datanode begins to compute a local closed cube for this block. The reduce task does nothing except simply output the local closed cube.

A point query is a cell whose measures are unknown. When the query phase starts, the OLAP queries are sent to every local closed cube. Then map task scans its local closed cube and returns a local query result set. These local query results gathered from local closed cubes are partitioned and grouped by keys (i.e., the cells). The reduce task combines the measures to one final result for every group.

## 3.2 Data Partitioning

Theorem 1 shows that with the data block number k increasing, the total size of all local closed cubes tends to shrink gradually until it arrives at the minimum value $T$. This results in much more cells unclosed, which implies that more time may be spent in scanning the local closed cubes.

Since the number of data blocks heavily affects the closed cubes' storage and query answering time, the number can be configured by users to control the ratio of storage and time. For instance, if users want to get much data compression and acceptable query answering time, the number of data blocks can be set greater.

## 3.3 Parallel Construction Algorithm

For the input data (e.g. text files or tables), MapReduce framework partitions them into blocks. Every block's content as a whole is an input value and assigned a unique key by the MapReduce framework. The map procedures accept these pairs and compute a local closed cube per input pair by calling QC-DFS algorithm. The closed cells are collected into a local variable closedCells (line 2 in the DFS procedure). Finally the closedCells identified by a blockid are output (line 3 in the map procedue). Note that each local closed cube containing the closed cells is stored in a file.

```
Algorithm1 Parallel ClosedCubing
local variable closedCells;
map(InputKey blockid, InputValue blockdata)
1. cl = (ALL, …, ALL);
2. call DFS(cl, blockdata, 0);
3. emit(blockid, closedCells);

DFS(cl, data, d)
1. Compute the closed cell cc(including measures) of cl;
2. closedCells += cc;
3. if cc is examined previously, then return;
4. for(i=d; i<dimsNum; i++)
5.   if cc[i] == ALL
6.    card=Partition(data, i); //partition data in ith dimension
7.    for(j=0; j<card-1;j++)
8.     let subdata=data's jth partition;
9.     cc[i]=subdata[0][i];
10.    DFS(cc, subdata, i);
11. return;
```

**Figure 2: Parallel Construction Algorithm**

Note that *k* map tasks produce *k* data block and then *k* local closed cubes. These local closed cubes are distributed across datanodes without being merged into an overall closed cube. In this way, the total size of local closed cubes is less than that of ovrall closed cube.

## 3.4 Parallel Query Algorithm

Parallel query includes a map procedure and a reduce procedure. Each input file data as whole is regarded as the input of the map procedure. The point queries are stored in a file which can be accessed by all map tasks (line 1 in map procedure).

Then the map function search every queried cell in its local closed cube and emit a (cell, msr) intermediate key/value pair. These intermediate pairs are partitioned by keys, i.e. the cells, so the measures are grouped by cells. Finally, the measures for a cell are reduced to one measure by applying aggregate function (e.g. sum).

```
Algorithm2 Parallel QueryClosedCube
map(InputKey blockid, InputValue closedcells)
1. get queried cells from a file;
2. for each cell in queried cells
3.  msr = query(cell, closedcells);
4.  emit(cell, msr);

reduce(InputKey cell, InputValue msrlist)
1. result = 0;
2. for each msr in msrlist
3.   result += msr;
4. emit(cell, result);
```

**Figure 3: Parallel Query Algorithm**

The query is processed in parallel across nodes. The bandwidth for the queries and the results is so small that there are little communication costs cross processors. This make the response time almost equal to that on a local closed cube. Thus, the response time increase little even for large data sets.

# 4. EXPERIMENTS

We have implemented the parallel construction and query algorithm using standard C++ and the Hadoop middleware. Hadoop is a software platform that allows one to easily write and run parallel or distributed applications that process vast amounts of data. It incorporates features similar to those of the Google File System and of MapReduce [10].

The experiments were conducted on an 18 node cluster with 3.0 GHz Intel Pentium 4 CPU, 1GB RAM and 30 GB 7200 RPM IDE hard disk space available per node. Every node was running Linux with kernel 2.6.21and gcc 4.1.2 and Hadoop 0.15.0. All nodes were on the same 100Mbit/sec Ethernet network. Among the nodes, one node served as the namenode and other 17 nodes the datanodes.

We generated a large number of synthetic data sets with Zipf factor all set to 0. The other parameters of data sets are: n - number of tuples, *d* - number of dimensions, $|D_0|$, $|D_1|$ . . . $|D_{d-1}|$ - cardinality in each dimension, p – number of processors.

We mainly explored the CPU time and corresponding speedup, since they are essential evaluation factors for parallel application.

## 4.1 The Effect of Data Partitioning

We verified theorem 1 on 11 node cluster. The test data set parameters are as following: n = 20M, d = 5. According to users' configuration for the number of data blocks, the system partitions the test data set into data blocks. Each data block generates a local closed cube. When the number of data blocks increases, the number of all local closed cells decreases gradually in Figure 4 (a).

Figure 4 (b) shows the total size of all local closed cubes also goes down. Although the total storage volume is compressed greatly with the number of data blocks increasing, the CPU time of construction and query of closed cube tends to increase as shown in Figure5. Users can make the balance between data cube storage and query answering time in our system by configuring the number data blocks before data partitioning.

## 4.2 Parallel Construction Performance

Figure 6 compares CPU time of closed cube construction w.r.t processors between 20M tuples and 60M tuples. We observed that with processors (nodes) increasing, the CPU time of construction of closed cube varies greatly when the number of tuples is large. But the CPU time of construction of closed cube falls down slowly for the small data set because the workload per node is not full. For a data set of 60M tuples (file size 1.37G), the speedup is almost linear. All local closed cubes are constructed in less than 5 minutes and output 2.98G size of files over 17 nodes. Figure 7 (b) shows that with the number of dimensions increasing, the CPU time of the construction of closed cubes rises exponentially. But the time is still acceptable. For example, it just costs 273 seconds to compute the data set with 12 dimensions and 20M tuples.

## 4.3 Parallel Query Performance

In practical business application, users simultaneously send multiple point queries. So we randomly generated 1,000 cells to be queried. Figure 8 shows that the query answering time for local closed cubes constructed from large data set decreases quickly with the processors increasing. For 60M tuples, the query answering time is only 203 seconds for 1,000 point queries over 17 nodes. Each point query answering time is 0.2 seconds in average. Also the time approaches linear speedup when the number of nodes increases from 5 to 17.

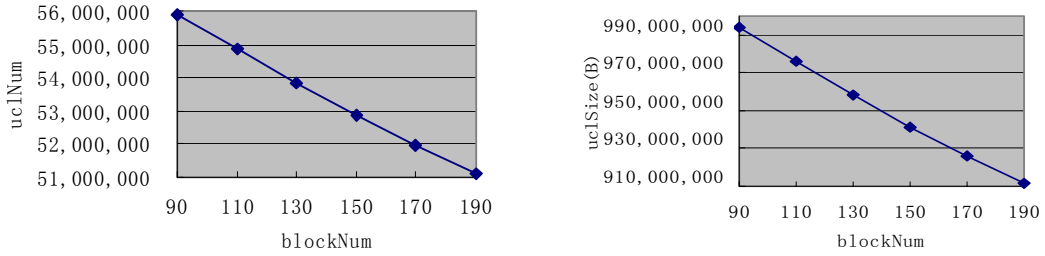


**Figure 4:** (a) the number of local closed cells w.r.t the number of data blocks; (b) the total size of local closed cubes w.r.t the number of data blocks

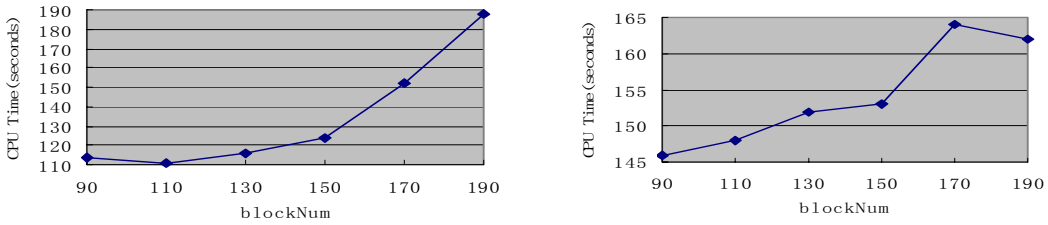


**Figure 5:** (a) CPU time of closed cube query w.r.t the number of data blocks; (b) CPU time of closed cube construction w.r.t the number of data blocks

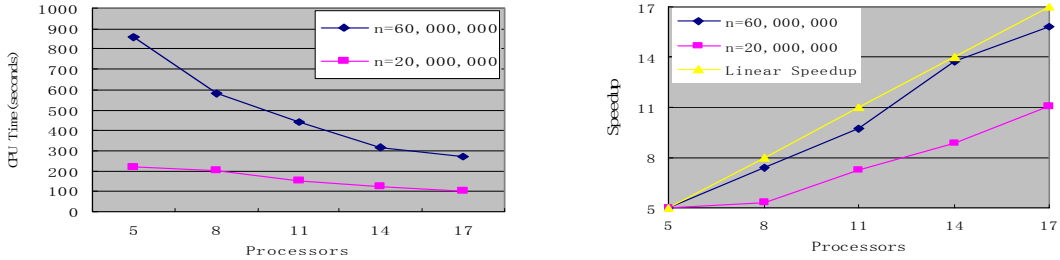


**Figure 6:** (a) CPU time of closed cube construction w.r.t processors, where $d$=5, $|Di|$=2000, 1000, 800, 300, 80; (b) Corresponding speedup

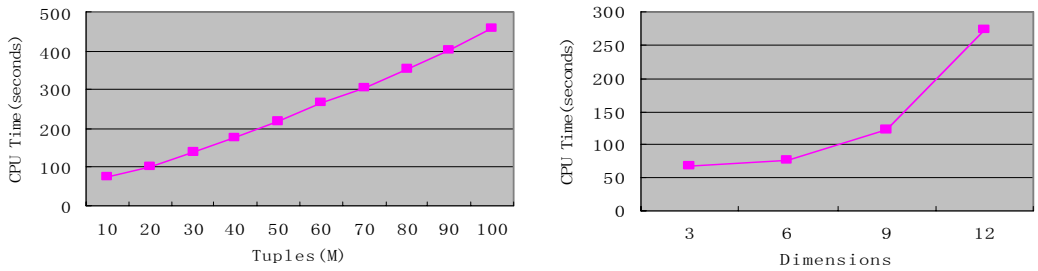


**Figure 7:** (a) construction time w.r.t tuples where $d$=5, $p$=17 (b) construction time w.r.t dimensions where $n$=20,000,000, $p$=17

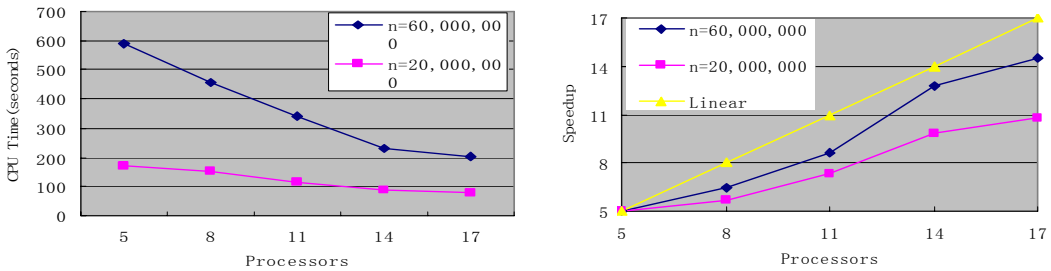


**Figure 8:** (a) CPU time of closed cube query w.r.t processors, where 1000 point queries are submitted to each local closed cube. (b) Corresponding speedup

## 5. Conclusions

For massive data, new data cubing approaches need to be explored. We have developed a simple but efficient solution that incorporates the features of the closed cube and the MapReduce. Our parallel algorithm takes care of the compression of the data cube size, without the much cost of query answering time. In addition, the fault-tolerance and the workload balance in the MapReduce framework makes our solution more applicable. Moreover, we give a demonstration that the computation tasks in data warehouse can be well processed using the MapReduce framework. We are currently developing our parallel data warehouse prototype system using this approach.

## REFERENCES


1. Dong Xin, Z.S., Jiawei Han, Hongyan Liu, *C-Cubing: Efficient Computation of Closed Cubes by Aggregation-Based Checking.* ICDE, 2006.
2. Frank Dehne, T.E., Andrew Rau-Chaplin, *The cgmCUBE project: Optimizing parallel data cube generation for ROLAP.* Distributed and Parallel Databases, 2006.
3. Jeffrey Dean, S.G., *MapReduce: Simplified Data Processing on Large Clusters.* Operating Systems Design and Implementation, 2004.
4. Jim Gray, S.C., Adam Bosworth, Andrew Layman, Hamid Pirahesh etc., *Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals.* Data Mining and Knowledge Discovery, 1997.
5. Kevin Beyer, R.R., *Bottom-Up Computation of Sparse and Iceberg CUBEs.* SIGMOD, 1999.
6. Laks V.S. Lakshmanan, J.P., Jiawei Han, *Quotient Cubes: How to Summarize the Semantics of a Data Cube.* VLDB, 2002.
7. Laks V.S. Lakshmanan, J.P., Yan Zhao, *QCTrees: An Efficient Summary Structure for Semantic OLAP.* SIGMOD, 2003.
8. Raymond T. Ng, A.W., Yu Yin, *Iceberg-cube Computation with PC Clusters.* SIGMOD, 2001.
9. Ying Chen, F.D., Todd Eavis, Andrew Rau-Chaplin, *Parallel ROLAP Data Cube Construction On Shared-Nothing Multiprocessors.* Distributed and Parallel Databases, 2004.
10. http://lucene.apache.org/hadoop/