HaoLap: a Hadoop based OLAP system for big data

Jie Song, Chaopeng Guo, Zhi Wang, Yichan Zhang, Ge Yu, Jean-Marc Pierson

HaoLap: A Hadoop based OLAP system for big data

Jie Song[a,*], Chaopeng Guo[a], Zhi Wang[a], Yichan Zhang[a], Ge Yu[b], Jean-Marc Pierson[c]

[a] *Software College, Northeastern University, Shenyang 110819, China*
[b] *School of Information and Engineering, Northeastern University, Shenyang 110819, China*
[c] *Laboratoire IRIT, Université Paul Sabatier, Toulouse F-31062, France*

Keywords:
Cloud data warehouse
Multidimensional data model
MapReduce

A B S T R A C T

In recent years, facing information explosion, industry and academia have adopted distributed file system and MapReduce programming model to address new challenges the big data has brought. Based on these technologies, this paper presents HaoLap (Hadoop based oLap), an OLAP (OnLine Analytical Processing) system for big data. Drawing on the experience of Multidimensional OLAP (MOLAP), HaoLap adopts the specified multidimensional model to map the dimensions and the measures; the dimension coding and traverse algorithm to achieve the roll up operation on dimension hierarchy; the partition and linearization algorithm to store dimensions and measures; the chunk selection algorithm to optimize OLAP performance; and MapReduce to execute OLAP. The paper illustrates the key techniques of HaoLap including system architecture, dimension definition, dimension coding and traversing, partition, data storage, OLAP and data loading algorithm. We evaluated HaoLap on a real application and compared it with Hive, HadoopDB, HBaseLattice, and Olap4Cloud. The experiment results show that HaoLap boost the efficiency of data loading, and has a great advantage in the OLAP performance of the data set size and query complexity, and meanwhile HaoLap also completely support dimension operations.

1. Introduction

With the development of computer technologies and its widespread usage in fields like Internet, sensors and scientific data analysis, the amount of data has explosively grown and the data volumes are approximately doubling each year (Gray et al., 2005). The scientific fields (e.g., bioinformatics, geophysics, astronomy and meteorology) and industry (e.g., web-data analysis, click-stream analysis and market data analysis) are facing the problem of "data avalanche"(Miller, 2010). There are tremendous challenges in storing and analyzing big data (Shan et al., 2011; Xiaofeng and Xiang, 2013).

On-Line Analytical Processing (Shim et al., 2002) (OLAP) is an approach to answer multidimensional analytical queries swiftly, and provides support for decision-making and intuitive result views for queries. However, the traditional OLAP implementation, namely the ROLAP system based on RDBMS, appears to be inadequate in face of big data environment. New massively parallel data architectures and analytic tools go beyond traditional parallel SQL data warehouses and OLAP engines. Therefore, some databases such as SQL Server and MySQL are able to provide OLAP-like operations, but the performance cannot be satisfactory (Chaudhuri et al., 2011). Generally, OLAP has three types, such as ROLAP (Relational Online Analytical Processing), MOLAP (Multidimensional Online Analytical Processing), and HOLAP (Hybrid Online Analytical Processing) (Chaudhuri et al., 2011). The differences between three types of OLAP can be listed as follows: (1) MOLAP servers directly support the multidimensional view of data through a storage engine that uses the multidimensional array abstraction, while in ROLAP, the multidimensional model and its operations have to be mapped into relationship and SQL queries; (2) in MOLAP, it typically pre-computes large data cubes to speed up query processing, while ROLAP relies on the data storage techniques to speed up relational query processing; (3) MOLAP suffers from poor storage utilization, especially when the data set is sparse; (4) since ROLAP relies more on the database to perform calculations, it has more limitation in the specialized function it can use; (5) HOLAP combines ROLAP and MOLAP by splitting storage of data in a MOLAP and a relational store. ROLAP and MOLAP both have pros and cons. However in big data environment, the drawbacks of MOLAP are nothing as compared to the advantages of quick response would bring and the cost needed to implement would be negligible if we optimize the implement approach of MOLAP.

Industry and academia have adopted distributed file system (Bolosky et al., 2000), MapReduce (Dean and Ghemawat, 2008) programming model and many other technologies (Song et al., 2011)

* Corresponding author. Tel.: +86 02483689258.
E-mail address: songjie@mail.neu.edu.cn (J. Song).

to address performance challenges. MapReduce is a well-known framework for programming commodity computer clusters to perform large-scale data processing algorithm. Hadoop (Apache, 2013a,b), an open-source MapReduce implementation, is able to process big data sets in a reliable, efficient and scalable way. Based on Hadoop, many cloud data warehouses (e.g., Hive (Thusoo et al., 2009), HBase (Leonardi et al., 2014), and HadoopDB (Abouzeid et al., 2009)) are developed and widely used in various fields. Even though these data warehouses support ROLAP-like functions, the performances are unsatisfactory. The reasons for this situation are: (1) these systems do not provide big data oriented OLAP optimizations; (2) the join operation, which is quite common operation in ROLAP, is very inefficient when big data are involved (Song et al., 2012). In this work we provide evidences in Section 7 to prove that when data amount or query complexity increases the performance of ROLAP-tools decreases. Compared with ROLAP the query performance of MOLAP is faster. Industry and academia develop many OLAP tools based on HBase (e.g., Olap4Cloud, HBaseLattice). However, in order to simplify the design of the tools, they give up many characteristics of MOLAP for example the direct operation of dimension like roll up and drill down. Hence, the tools only support data query and aggregation but does not naturally support dimension hierarchy operations. In general, MOLAP suffers from long data loading process especially on large data volumes and difficult querying models with dimensions with very high cardinality (i.e., millions of members) (Wikipedia, 2014). In big data environment, how to cope with disadvantages of MOLAP meanwhile support dimension hierarchy operations naturally becomes a challenge.

In this paper we present HaoLap (**Ha**doop based **oLap**) an OLAP system for big data. Our contributions in this paper can be listed as follows: (1) drawing on the experience of MOLAP, HaoLap adopts many approaches to optimize OLAP performance, for example the simplified multidimensional model to map dimensions and measures, the dimension coding and traversing algorithms to achieve the roll up operation over dimension hierarchies. (2) We adopt the partition and linearization algorithms to store data and the chunk selection strategy to filter data. (3) We deploy the OLAP algorithm and data loading algorithm on MapReduce. Specifically, HaoLap stores dimensions in metadata and stores measures in HDFS (Shvachko et al., 2010) without introducing duplicated storage.

In general simplified multidimensional model and data loading algorithm make loading process of HaoLap simple and effective. In query process, it make HaoLap could handle high cardinality that we do not have to instantiate the cube in the memory because of the OLAP algorithm and MapReduce framework.

The differences between HaoLap and the other MOLAP tools can be listed as follows: (1) HaoLap adopts simplified multidimensional model to map dimensions and measures that keeps data loading process and OLAP simple and efficient. (2) In OLAP, HaoLap adopts the dimension coding and traversing algorithms proposed in the paper to achieve the roll up operation over dimension hierarchies. (3) HaoLap do not rely on pre-computation and index technologies but sharding and chunk selection to speed up OLAP. (4) In OLAP, HaoLap do not store large multidimensional array but calculation. In general, HaoLap is kind of MOLAP tool which adopts simplified dimension and keep OLAP simple and efficient.

In the paper we design a series of test cases to compare HaoLap with some open-source data-warehouses systems (Hive, HadoopDB, Olap4Cloud, and HBaseLattice) designed for big data environment. The results indicate that HaoLap not only boosts the efficiency of loading data process, but also has a great advantage in the OLAP performance regardless of the data set size and query complexity on the premise that HaoLap completely supports dimension operations.

The rest of this paper is organized as follows. Following the introduction, Section 2 introduces the related work. Section 3 introduces the definitions and Section 4 explains all the algorithms on the proposed data model including dimension related algorithms, partition algorithms, data storage algorithms and chunks selection algorithm. Section 5 described the MapReduce based OLAP and data loading implementation. Section 6 introduces the system architecture of HaoLap, and explains each component of the system. Section 7 evaluates the loading, dicing, rolling up and storage performance of HaoLap and compares that with Hive, HadoopDB, HBaseLattice, and Olap4Cloud. Finally, conclusions and future works are summarized in Section 8.

2. Related work

OLAP was introduced in the work done by Chaudhuri and Dayal (1997). They provided an overview of data warehousing and OLAP technologies, with an emphasis on their new requirements. Considering the past two decades have seen explosive growth, both in the number of products and services offered and in the adoption of these technologies in industry, their other work (Chaudhuri et al., 2011) gave the introductions of OLAP and data warehouse technologies based on the new challenges of massively parallel data architecture.

There exist some optimization approaches of OLAP system, which are related to this paper. The OLAP optimizations can be classified as follows: (1) taking advantage of pre-computation; (2) optimizing data model or storage system to boost the OLAP performance; (3) taking advantage of data structure to optimize OLAP algorithm; (4) taking advantage of implementation of OLAP to speed up query process. The latter three are close to HaoLap and introduced briefly in this section.

Yu et al. (2011) introduce epic, an elastic power-aware data-intensive cloud platform for supporting both data intensive analytical operations and online transactions, a storage system, supporting OLAP and OLTP through index, data partition, was introduced. D'Orazio and Bimonte (2010) tried to store big data in multidimensional array and apply the storage model to Pig (Apache, 2013a,b), which is a data analysis tool and is based on Hadoop. Then, the storage model is proved efficient by experiment. In these studies, the data storage and data model are discussed, while in HaoLap the multidimensional data model is designed. In addition, we adopt the dimension coding method and come up with dimension related algorithms and OLAP algorithm to boost the OLAP performance.

Tian (2008) presented a OLAP approach based on MapReduce parallel framework. First, a file type was designed, which was based on SMS (Short Message Service) data structure. Then, the OLAP operation was implemented using MapReduce framework. Ravat et al. (2007) defined a conceptual model that represents data through a constellation of facts and dimensions and present a query algebra handling multidimensional data as well as multidimensional tables. These studies take advantage of special data structure to improve the OLAP performance while in HaoLap the multidimensional model is adopted to perform OLAP, so the formers have a certain usage limitation.

There are some other studies for optimizing OLAP performance. Jinguo et al. (2008) take advantage of pre-computation to optimize the performance of the OLAP. At the same time, the study proposed the query algorithm based on MapReduce framework. In comparison, HaoLap takes advantage of MapReduce to perform the distributed OLAP algorithm, but we do not adopt pre-computation and closed cube technology in order to reduce the storage in big data environment. Abelló et al. (2011) compare the different approaches to retrieve data cubes from BigTable (Chang et al., 2008). In the study, they focus on retrieving data cubes by taking advantage of BigTable and MapReduce paradigm and implementing the approaches on HBase. In the process of experiments, we draw on experiences from this study.

Lin and Kuo (2004) proposed a greedy-repaired genetic algorithm, called the genetic greedy method, to solve the data selection problem. The data cube selection problem is, given the set of user queries and a storage space constraint, to select a set of materialized cubes from the data cubes to minimize the query cost and/or the maintenance cost. While in HaoLap, we solve the problem by chunk selection algorithm (Section 4.5) which is more effective owing to the simplified multidimensional data model.

Some practical data warehouses based on Hadoop have emerged such as Hive, HadoopDB and HBase. Hive, a framework for data warehousing on top of Hadoop, was created to make it possible for analysts with strong SQL skills to run queries on the huge volumes of data stored in HDFS (Thusoo et al., 2009). HadoopDB is an architectural hybrid of MapReduce and DBMS technologies for Analytical workloads (Abouzeid et al., 2009). Olap4Cloud (akolyade, 2013) and HBaseLattice (dlyubimov, 2013), based on HBase (Khetrapal and Ganesh, 2006), are MOLAP-like systems. Olap4Cloud takes advantage of index technologies to provide lightweight OLAP and boost the performance of OLAP, but when data amount increases the cost of scanning index table is intolerant. HBaseLattice adopts index technology and MapReduce to provide low latency OLAP. Both Olap4Cloud and HBaseLattice adopt pre-computation to optimize performance of OLAP, which is not suitable for big data environment. When data volume is large, the loading performance is bad. At the same time, in order to decrease OLAP time, Olap4Cloud and HBaseLattice give up many characteristics of MOLAP for example they replace roll up operation with aggregation operation. Wu et al. (2012) proposed Avatara, the Hadoop-based OLAP implementation designed by and used at LinkedIn. Avatara provides online analytics and supports the high throughput, low latency, and high availability for a high-traffic website environment. The basis technologies of Avatara are pre-computation and sharding. They optimized small queries in a "many, small cubes" scenario. In conclusion, Avatara is suitable for small cube but HaoLap is designed for big data.

There are few works focus on boosting the performance of OLAP by enhancing the computing ability. Riha et al. (2013) introduced adaptive hybrid OLAP architecture. An OLAP system takes advantages of heterogeneous systems with GPUs and CPUs and leverages their different memory subsystems characteristics to minimize response time. Lima et al. (2009) proposed more efficient distributed database design alternatives which combine physical/virtual partitioning with partial replication. In HaoLap, we exploit MapReduce framework to implement OLAP in a parallel processing way.

In summary, at present the lack of effective support for multidimensional data storage model and OLAP analysis needs to be resolved urgently in big data era. At the same time, Hadoop as a cloud-computing framework is most widely used in the big data analysis platform, but the MOLAP tools, based on Hadoop, are still blank. Based on our previous research (Song et al., 2014), in which we propose the distributed MOLAP technique for big data environment, we designed the HaoLap system, which has been proven efficient on the storage and analysis of big data. This research has certain theoretical and practical values.

3. Definitions

In order to implement efficient OLAP in big data environment, HaoLap introduces simplified multidimensional model based on a simplified dimension that is defined in this section. In addition, we give few related definitions that are basis for algorithms illustrated in next section. In HaoLap, the definitions of level, measure, cell and cube are in accordance with the traditional multidimensional model (Chaudhuri et al., 2011), therefore we do not redefine these definitions in the paper.

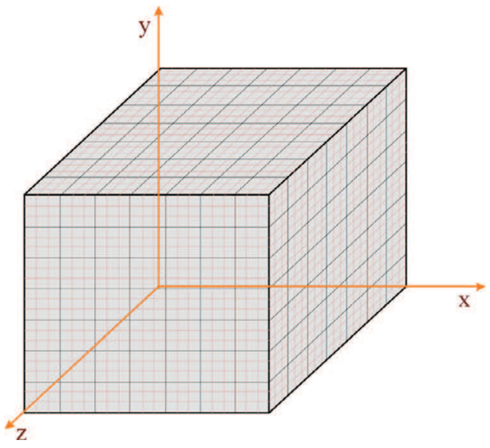


Fig. 1. Example of HaoLap cube.

Definition 1 (Simplified dimension)**.** In a multidimensional data model, a dimension categorizes each item in a data set into non-overlapping regions in order to providing the filtering, grouping and labeling. Dimension in HaoLap is a simplified one named simplified dimension, and it follows common dimension definition and three restrictions below. Let d be a simplified dimension in HaoLap:

(i) d has one and only one dimension hierarchy.
(ii) d has m ($m \geq 1$) dimension levels, denoted as $l_1, l_2, \ldots, l_m$. Let l_i ($i \in [1,m]$) be a dimension level, l_i has only one dimension attribute with n_i dimension values.
(iii) In the hierarchy of dimension values, sibling nodes have same number of descendants.

Based on these assumptions, a definition of HaoLap dimension is given as follow.

A simplified dimension d is composed of:

- A scheme, made of:
  (i) A finite set $L(d)$ of m levels, and these levels are organized as one and only one concept hierarchy;
  (ii) A total order $\succ_d$ on $L(d)$. Roll up is the operation that climbs up the hierarchy of the dimension. If $l_j \succ_d l_i$ ($i, j \in [1,m]$, $i < j$), we say l_j rolls up to l_i.

  A dimension can be treated as an total order set $L(d)$ and the relation $\succ_d$.
- An instance, made of:
  (i) A function m_d, associating a set of dimension values with each level;
  (ii) A roll up function ${\rho_d}^{lj \to li} = m_d(l_j) \to m_d(l_i)$ for each pair of levels $l_j \succ_d l_i$. $\forall v^j \in m_d(l_j)$, $\exists v^i \in m_d(l_i)$ satisfying ${\rho_d}^{lj \to li}(v^j) = v^i$, and $\forall v^i \in m_d(l_i)$, $\exists v^j \in m_d(l_j)$ satisfying ${\rho_d}^{lj \to li}(v^j) = v^i$.
  (iii) $\forall {v_x}^i, {v_y}^i \in m_d(l_i)$, $|\{{v_x}^{i+1} | {\rho_d}^{li+1 \to li}({v_x}^{i+1}) = {v_x}^i\}| = |\{{v_y}^{i+1} | {\rho_d}^{li+1 \to li}({v_y}^{i+1}) = {v_y}^i\}|$.

Definition 2 (Chunk)**.** A chunk is a logic partition of a cube. A cube is divided into chunks according to the cube partition algorithm.

Fig. 1 illustrates a cube that includes three dimensions (x, y and z). In Fig. 1, the smaller cuboid represents cell and the bigger cuboid, which consist of few smaller cuboids, represents chunk. In practical, chunks may contain *null* cells which does not contain any measure value.

Table 1
Symbols used in definition illustrations.

Symbol	Description
d_i	The *i*th ($i \in [1,n]$) dimension of the cube
l_j	The *j*th ($j \in [1,m]$) level of the dimension
$L(d)$	A finite set consisting of all levels of the dimension *d*
$m_d(l_i)$	A set of all dimension values at the level l_i
$\|l_i\|$	The level size of l_i—the number of descendants of a node that at $l_{i\text{-}1}$
$\|d\|$	The dimension size of the dimension *d*
$\|cube\|/\|chunk\|$	The size of the cube or chunk
$\|cube\|^n/\|chunk\|^n$	The capacity of the cube or chunk

Definition 3 (*Level size*)**.** Level size is the number of descendants of a node in the dimension hierarchy tree. In HaoLap, let *d* be a dimension, consisting of *m* levels, l_i ($i \in [1,m-1]$) be a dimension level and the level size of l_i be $|l_i|$, then, $\forall v^i \in m_d(l_i)$, $|l_{i+1}| = |\{v^{i+1}|\rho_d{}^{li+1\to li}(v^{i+1}) = v^i\}|$.

Definition 4 (*Dimension size*)**.** Dimension size is the number of dimension values of bottom level. Let *d* be a dimension, which consist of *m* levels denoted as $l_1, l_2 \ldots l_m$, and $|d|$ be the size of the dimension, therefore $|d| = |m_d(l_m)| = \prod_{i=1}^{m}|l_i|$.

Definition 5 (*Cube size and cube capacity*)**.** Cube size is a tuple that consists of the size of dimensions, which compose the cube. Cube capacity is the number of cells in a cube. Let *cube* consists of *n* dimensions, denoted as d_i ($i \in [1,n]$), therefore cube size is denoted as $|cube| = \langle |d_1|,|d_2| \ \ldots \ |d_n|\rangle$ and cube capacity is denoted as $|cube|^n = \prod_{i=1}^{n}|d_i|$.

Definition 6 (*Chunk size and chunk capacity*)**.** Chunk size is a tuple, which consists of the number of dimension values of bottom level of dimensions, which compose the chunk. Chunk capacity is number of cells in the chunk. Let a chunk consists of *n* dimensions, denoted as d_i ($i \in [1,n]$) that is partitioned into p_i segments, therefore the chunk size is denoted as $|chunk| = \langle \lambda_1,\lambda_2,\ldots,\lambda_n\rangle$ ($\lambda_i = \lceil |d_i|/p_i \rceil$), and chunk capacity is denoted as $|chunk|^n = \prod_{i=1}^{n}\lambda_i$.

Definition 7 (*OLAP query*)**.** In HaoLap, OLAP query is a quadruple, denoted as <*Target*, *Range*, *Aggregation*, *Result*>. *Target* is the cube to be queried or analyzed. *Range* gives the dimension range of the *Target*. *Aggregation* means the aggregation function, such as *mean*(), *sum*(), *maximum*() and *minimum*(). *Result* is the cube of query results.

In every OLAP query, the input and output are cubes. *Result* is the newly created cube queried and aggregated from the *Target*. Obviously, the dimension size of *Target* and *Result* may be different.

There are few symbols involved in the experiment analysis which are introduced in above illustrations. We list the symbols in Table 1.

4. Algorithms

In this section, a series of algorithms in HaoLap are discussed. As far as ROLAP is concerned, generally, the star model and snowflake model are adopted in the relational data warehouse (Chaudhuri and Dayal, 1997), which stores dimensions and measures into relational tables and uses foreign keys to refer them. Specifically, in the face of big data the performance of ROLAP is unacceptable for it involves numerous costly join operations. On the contrary, MOLAP has fast response time through saving the cost of join operations especially for big datasets. MOLAP system offers robust performance, but it needs additional storage to maintain the mappings between dimensions and measures (Taleb et al., 2013). Generally, MOLAP acquires the relationship through dimension coding or direct addressing in which the heavy index of dimensions is required but in the face of big data, the storage cost and complexity of maintaining dimension index are unacceptable. In this section, dimension coding algorithm, dimension traversing algorithm, cube partition algorithm, data storage algorithm, OLAP query algorithm and data loading algorithm are introduced.

In order to avoid the extra storage cost of MOLAP, HaoLap adopts simplified data model and sophisticated algorithms. HaoLap exploits the integer encoding method to avoid sparsity of multidimensional array. Meanwhile, we propose dimension traverse algorithm based on the simplified dimension model and the encoding method to mapping dimension values and measures in an efficient way in which we achieve the mapping by calculation rather than index approach. In addition, we propose cube partition algorithm and exploit the linearization algorithm of multidimensional array as data storage algorithm to accommodate to the big data environment. HaoLap introduces chunk partition algorithm to reduce the input data size of OLAP and meanwhile boost the performance of OLAP. The combination of the simplified dimensional model and the algorithms keeps the OLAP simple and effective.

4.1. Dimension coding algorithm

The dimension coding methods mainly include binary encoding and integer encoding (Goil and Choudhary, 1998). Binary encoding and integer encoding both have pros and cons. Binary encoding makes it easy to achieve information of levels, but it leads to sparsity of multidimensional array, while integer encoding do not have the sparsity problem but it cannot achieve level information directly. If we code the *Time* dimension (*Year-Month-Day*) from 2010-01-01 to 2011-12-31, taking 2010-12-31 and 2011-01-01 for example, when we use integer encoding the code of the former is 364 and the latter is 365, while when we use binary encoding, the situation is quite different in which the former is 010110101101100 and the latter is 100000000000000 and the difference of them is 4756. The discontinuity of the binary code leads to sparsity of multidimensional array.

In order to avoid sparsity problem, in HaoLap we adopt integer encoding method to code dimensions and use the traversing algorithm to calculate the mapping of the code and the dimension level. Let *l* be a dimension level of *d*, $\forall x \in [1,|m_d(l)|]$, $v_x \in m_d(l)$, and $code(v_x)$ be the code of v_x. Then we can get $code(v_x) = x$. The coding algorithm can be presented as Algorithm 1.

Algorithm 1 (*Dimension coding algorithm*)**.**

```
Input: Dimension d: a target dimension
Function Dimension coding
1.  For $i = 0$ to $|L(d)|$;
2.      For $j = 0$ to $|m_d(l_i)|$;
3.          Dimension value of $v_j{}^i \in m_d(l_i)$
4.          $v_j{}^i.code = j$;
5.      End for
6.  End for
```

Practically, many dimensions are numerical or continuous (e.g., height, depth, latitude, orders number). Numerical attribute could be modeled to a dimension by partition its value range, and each sub-range is a dimension value on the corresponding level, therefore numerical dimensions could easily conform the definition 1. There also exist a few non-numerical dimensions, such as *Time*, *City*, and *Department*. However, in order to satisfy Definition 1, some empty values are added to complement inconsistent number of descendants of sibling nodes. Taking Fig. 2 for example, at the *month* level, every *month* has the different number of *day*. In order to satisfy the Definition 1, we assume that every month has 31 days.

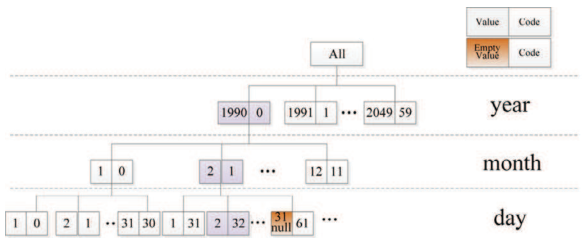


Fig. 2. Example of dimension coding and hierarchy of dimension values.

4.2. *Dimension traverse algorithm*

In HaoLap, the hierarchy of dimension values can be treated as a special rooted tree called hierarchy tree. In hierarchy tree, the *ALL* is the root node, denoted as level 0. The value of each dimension level can be treated as node of each hierarchy level, and sibling nodes have same amount of descendants.

OLAP involves the traversing of dimension, such as roll up or drill down. Coding mechanism is helpful for maintaining the relationship and we can implement the roll up on dimension tree by code calculation. If $code(v^{i+1})$ is given, $code(v^i)$ could be calculated in condition of v^{i+1} roll up to v^i. The drill down from v^i to v^{i+1} is treated as the roll up from v^{i+2} to v^{i+1}.

Let $\langle v^1, v^2, \ldots, v^i, v^{i+1}, \ldots, v^m\rangle$ be a top-down path of dimension tree, $order(v^i)$ be the number of previous sibling nodes of v^i (nodes with same parent node and in left-to-right order). Taking Fig. 2 for example, in path $\langle 1990^1, 2^2, 2^3\rangle$, $code(2^3) = 32$, $order(2^3) = 1$, $code(2^2) = 1$, $order(2^2) = 1$, $code(1990^1) = 0$, and $order\ (1990^1) = 0$. The relationship between $code(v^i)$ and $order(v^i)$ can be illustrated as Eq. (1).

$$code(v^i) = (\cdots((0 + order(v^1)) \times |l_2| + order(v^2)) \times |l_3| + order(v^3)\cdots) \times |l_m| + order(v^i) \quad (1)$$

Given $code(v^i)$, $order(v^1)$ to $order(v^i)$ could be calculated by Eq. (2).

$$\begin{aligned} &temp_i = code(v^i) \\ &order(v^i) = temp_i\%|l_i|\ temp_{i-1} = \left\lfloor \frac{temp_i}{|l_i|} \right\rfloor \\ &order(v^{i-1}) = temp_{i-1}\%|l_{i-1}|\ temp_{i-2} = \left\lfloor \frac{temp_{i-1}}{|l_{i-1}|} \right\rfloor \\ &\ldots \\ &order(v^1) = temp_1\%|l_1| \end{aligned} \quad (2)$$

Based on Eqs. (1) and (2), $code(v^{i-1})$ to $code(v^1)$ could be calculated if $code(v^i)$ is given, so that each node in the order path could be determined. For example, in Fig. 2, at *day* level $code(2^3) = 32$, the path is $\langle 1990^1, 2^2, 2^3\rangle$, according to Eq. (2), $code(2^3) = 32$, $|l_3| = 31$, $|l_2| = 12$ and $|l_1| = 50$, then $order(2^2) = 1$ and $order(1990^1) = 0$, thus $code(2^2) = 1$ and $code(1990^1) = 0$ are calculated according to Eq. (1).

4.3. *Cube partition algorithm*

The purposes of dividing cube into chunks are filtering input data of MapReduce, which can improve the query performance, and a chunk could be the storage unit of a cube (discussed in Section 4.4). In this section, we introduce the chunk partition algorithm.

The performance of OLAP is related to $|chunk|^n$. When $|chunk|^n$ is smaller, the parallelism could be better, and scanned cells for a certain query range could be much less, but the scheduling cost would be larger. Therefore, $|chunk|^n$ is critical and should be carefully determined. Inspired by Sarawagi and Stonebraker (1994), we propose an approach to determine the chunk size based on all query conditions and each condition's probability. However, the query conditions analyzed are random sample because it is impossible to use all query conditions up. In addition, $|chunk|^n$ is not only determined by the query conditions but also the features of MapReduce, such as file addressing time, data processing time and so on. Table 2 illustrates definitions of symbols. Among the symbols, only λ_i is a variable, while T and N_a are calculation results.

Table 2
Definition of related symbols.

Symbol	Description
T	The average execution time of OLAP
n	The number of dimensions
λ_i	The chunk size at d_i
N_a	The average number of chunks which are selected by queries
v	The number of map tasks processed per second
t_0	The file addressing time
t_1	The scheduling time of the task
q	The number of query conditions
a_{ij}	The number of dimension values of d_i which are matched query condition j
ξ_j	The occurrence probability of condition j

N_a can be calculated by aggregating a_{ij} and ξ_j

$$N_a = \sum_{j=1}^{q}\left(\prod_{i=1}^{n}\left\lceil\frac{a_{ij}}{\lambda_i}\right\rceil \xi_j\right) \quad (3)$$

When we consider some effects of MapReduce, we can get the average time of OLAP.

$$T = \left[\sum_{j=1}^{q}\left(\prod_{i=1}^{n}\left\lceil\frac{a_{ij}}{\lambda_i}\right\rceil \xi_j\right)\right]\left(t_0 + t_1 + \frac{\prod_{i=1}^{n}\lambda_i}{v}\right) \quad (4)$$

The chunk size can be calculated by Eq. (4), if the value of λ_i that minimize T is achieved, and meanwhile the chunk capacity can be calculated too. The details of partition algorithm are abbreviated in this paper. As we can see from the definitions, the sum of chunk capacity of total chunks may be larger than the cube capacity, because of the '$\lceil\rceil$' operation, thus some *null* cells can be found in a few boundary chunks.

4.4. *Data storage algorithms*

The storage cost of MOLAP could be huge because OLAP is fasted by the multidimensional array in the memory therefore such storage approach is impossible in face of big data. The "multidimensional array" of HaoLap is calculated not stored. Therefore, the storage of a HaoLap dimension only contains the size of each level because the codes of dimension values in same level are continuous and the sibling nodes have same number of descendants. Let d be a dimension with m dimension levels, denoted as $\{l_i|i \in [1,m]\}$, the storage of d is a set, denoted as $\{<l_i,|l_i|>|i \in [1,m]\}$. In practice, l_i is simplified as the level identity, and XML files are used to persist the $L(d)$ permanently in *Metadata Server*.

Logically, both "cells of the cube" and "chunks of the cube" can be associated as values of the multidimensional array. Physically, a chunk is the storage unit of a cube, stored as a MapFile (Lai and ZhongZhi, 2010) in HDFS. Both chunk files and cells are linearized for persistence, and reverse-linearized for queries, which are similar to the linearization and reverse-linearization of multidimensional array.

Let a multidimensional array consist of n dimensions and the size of array are denoted as $\langle A_1, A_2, \ldots, A_n\rangle$. If the coordinate of a value in the array is denoted as $(X_1, X_2, \ldots, X_n)$, the linearization and reverse-linearization equations are illustrated as Eqs. (5) and (6), respectively specifically.

$$index(X) = (\cdots((X_n \times A_n + X_{n-1}) \times A_{n-1} + \cdots + X_3) \\ \times A_3 + X_2) \times A_2 + X_1 \tag{5}$$

$$\begin{aligned} & temp_1 = index \\ & X_1 = temp_1 \% A_1 \; temp_2 = \left\lfloor \frac{temp_1}{A_1} \right\rfloor \\ & X_2 = temp_2 \% A_2 \; temp_3 = \left\lfloor \frac{temp_2}{A_2} \right\rfloor \\ & \ldots \\ & X_n = temp_n \% A_n \end{aligned} \tag{6}$$

As far as a cube is concerned, let x be a cell referred by d_1, d_2, ..., d_n. Let v^i, to which x refers, be dimension value of d_i, $code(v^i)$ be x_i, then the coordinate of x is $(x_1, x_2, \ldots, x_n)$. We can calculated the linearized index of x by Eq. (5) in which $(X_1, X_2, \ldots, X_n)$ is $(x_1, x_2, \ldots, x_n)$ and $\langle A_1, A_2, \ldots, A_n \rangle$ is $\langle |d_1|, |d_2|, \ldots, |d_n| \rangle$. The reverse-linearization is the same as Eq. (6). The cell is linearized and stored as a record in the file, which includes linearized coordinate of the cell and measures in the cell.

As far as a chunk is concerned, let y be a chunk, which is a partition of the cube, $|y| = \langle \lambda_1, \lambda_2, \ldots, \lambda_n \rangle$ and the coordinate of y is $(y_1, y_2, \ldots, y_n)$, in which $y_i = \lfloor x_i/\lambda_i \rfloor$. The name of chunk file is the linearized index of chunk. The linearized index of y can be calculated by Eq. (5) in which $(X_1, X_2, \ldots, X_n)$ is $(y_1, y_2, \ldots, y_n)$ and $\langle A_1, A_2, \ldots, A_n \rangle$ is $\langle \lambda_1, \lambda_2, \ldots, \lambda_n \rangle$, and reverse-linearization is the same as Eq. (6).

In the implementation, a chunk is a *MapFile* of Hadoop HDFS and the coordinate of the chunk after linearization is the name of the chunk, which is convenient to addressing. In a chunk file, records are stored in the form of *key-values*, in which the *values* are responded to the measures and the *keys* are the index of the corresponding cells, linearized according to Eq. (5). In order to avoid the sparity problem of measures, for the measure whose value is *NULL* and its key, they are not stored in the chunk. The index is integer, but the size of cube would be very large in the big data environment. Taking Java as an example, the value of index can overflow the range of long integer (2^{64}), so we use string to represent the integer. In addition, the cost of storing *key* may be much larger than that of *values* in *key-values* data model because index is a big number, we optimize the *key-values* storage, in which only the minimum index and the offsets are stored in the chunk file.

4.5. Chunk selection algorithm

OLAP operations mainly includes roll up, drill down, slice, dice and pivot. Actually, slice and dice can be treated as the range query operation, while roll up and drill down can be treated as the combination of query and aggregation, and pivot can be treated as create the different view of the cube. Thus, the OLAP operations could be abstracted as aggregated query, which aggregates data of a certain range.

In data warehouses, the measures are referred and queried by dimensions, therefore the query conditions are defined at each dimension. *Range* indicates a query range of cells in *Target*. Obviously, a *Range* is a multidimensional tuple. Let $\{d_1, d_2 \ \ldots \ d_n\}$ be the dimensions of *Target*, and an ordered pair $\langle \alpha_i, \beta_i \rangle$ $(\alpha_i < \beta_i)$ be given query range on dimension d_i, therefore *Range* = $\{\langle \alpha_i, \beta_i \rangle | \ i \in [1,n]\}$. For convenience, *Range* is defined as an ordered pair:

$$\langle \{\alpha_i\}, \{\beta_i\} \rangle = \langle \{\alpha_1, \alpha_2, \cdots, \alpha_n\}, \{\beta_1, \beta_2, \cdots, \beta_n\} \rangle$$

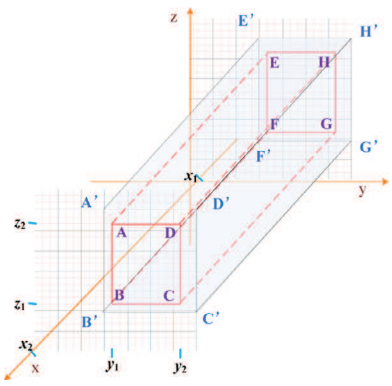


Fig. 3. Example of chunks selection algorithm.

The OLAP query includes three steps, such as chunks selection, data filtering, and data aggregation. Many researches have discussed how to implement the query and aggregation in MapReduce (Jing-hua et al., 2012; Wu et al., 2013). In this section, chunks selection is introduced. HaoLap provides chunks selection algorithm to reduce queried space by selecting chunks that matches the query range.

Let a *Range* be $\langle \{\alpha_1, \alpha_2, \ldots, \alpha_n\}, \{\beta_1, \beta_2, \ldots, \beta_n\} \rangle$, coordinator of the chunk be $(c_1, c_2 \ldots c_n)$, and the chunk size be $<\lambda_1, \lambda_2, \ldots, \lambda_n>$. If $\forall i \in [1,n]$ satisfy Eq. (7), we say that the chunk matches the *Range*.

$$\left\lfloor \frac{\alpha_i}{\lambda_i} \right\rfloor \le c_i \le \left\lfloor \frac{\beta_i}{\lambda_i} \right\rfloor \tag{7}$$

Chunks that match the *Range* are processed by MapReduce jobs. Taking Fig. 3 for example, assuming that the *Range* is $\langle \{x_1, y_1, z_1\}, \{x_2, y_2, z_2\} \rangle$, the point B is (x_1, y_1, z_1) and the point H is (x_2, y_2, z_2), and then the sub-cube $ABCD$–$EFGH$ contains the required cells. Through Eq. (7), we get the smallest cube $A'B'C'D'$–$E'F'G'H'$ that is composed by integral chunks and contains $ABCD$–$EFGH$. These chunks are input files of MapReduce job. Chunk selection avoids most of the cells that do not match the query conditions. Namely, the cells that are in the centric of $A'B'C'D'$–$E'F'G'H'$ match the query condition and avoid further data filtering. However, at the boundary of $A'B'C'D'$–$E'F'G'H'$, there are still some cells not matching the query conditions and that need to be filtered in MapReduce job.

5. Implementation

In this section, the data loading and OLAP implementations based on MapReduce framework are introduced. A MapReduce job consists four parts, including *InputFormatter*, *Mapper*, *Reducer*, and *OutputFormatter*. The MapReduce based OLAP process is shown as Fig. 4. Before the MapReduce job is performed, the query quadruple is submitted by client and verified in order to avoid the deterministic failures. Then HaoLap detects the *chunk-files-list* through the chunks selection algorithm mentioned in Section 4.5. In *InputFormatter*, the chunk files in the *chunk-file-list* are read and the cells in the chunks are scanned. Meanwhile, coordinates of cell are reverse-linearized and checked by the query conditions. If a coordinate match the query conditions, the linearized coordinate of the cell and the measure are pass to *Mapper*. Otherwise the cell is abandoned.

Both input and output of *Mapper* and *Reducer* functions are *key-value* pairs. In which, $\langle Key_M{}^{In}, Value_M{}^{In} \rangle$ and $\langle Key_M{}^{Out}, Value_M{}^{Out} \rangle$ are input and output format of *Map* function, respectively; $\langle Key_R{}^{In}, Value_R{}^{In} \rangle$ and $\langle Key_R{}^{Out}, Value_R{}^{Out} \rangle$ are input and output format of *Reduce* function, respectively. The $Key_M{}^{Out}$ should be implicitly converted to $Key_R{}^{In}$, and $Value_R{}^{In}$ is the collection of $Value_M{}^{Out}$.

In HaoLap, the $Key_M{}^{In}$ is the linearized coordinate of a cell and $Value_M{}^{In}$ is the value of measures read by *InputFormatter*. The

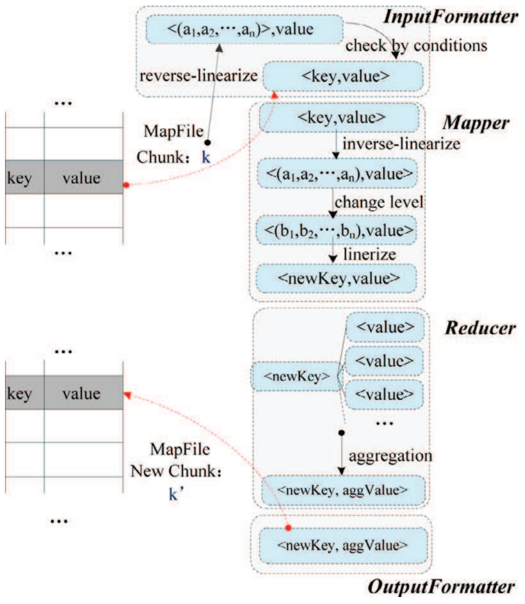


Fig. 4. MapReduce job of OLAP process.

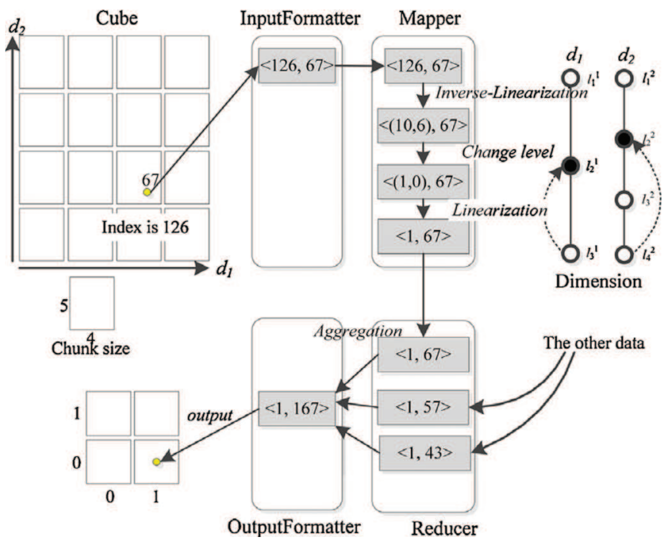


Fig. 5. The process of changing of key-value.

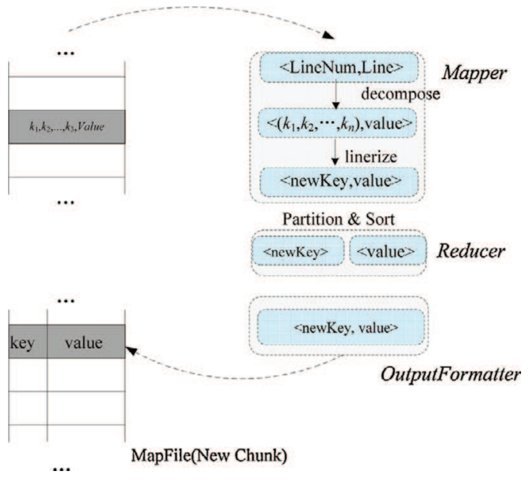


Fig. 6. MapReduce job of data loading.

${Key_M}^{Out}$ is the linearized coordinate of cell that is updated according to the *Result* of query quadruple while the value of ${Value_M}^{Out}$ is the same as ${Value_M}^{In}$. ${Key_R}^{In}$ is the same as ${Key_M}^{Out}$ while ${Value_R}^{In}$ is a collection of measures, which share the same ${Key_R}^{In}$. ${Key_R}^{Out}$ is the same as ${Key_R}^{In}$ while ${Value_R}^{Out}$ is an aggregative value of the measures.

In *Mapper*, the ${Key_M}^{In}$ is reverse-linearized firstly, then updated to the new dimension level according to the *Result* of query quadruple. Finally the updated key of *Mapper* is linearized and emitted from the *Mapper* as ${Key_M}^{Out}$ while the value of measure remain unchanged thus ${Value_M}^{Out} = {Value_M}^{In}$.

In *Reducer*, firstly, ${Value_R}^{In}$ is aggregated according to the *Aggregation* of query quadruple and ${Value_R}^{Out}$ is acquired. Finally, the aggregated value is emitted from the *Reducer* as ${Value_R}^{Out}$ while the key of *Reducer* remain unchanged, thus ${Key_R}^{out} = {Key_R}^{In}$.

The output of each *Reducer* is passed to *OutputFormatter*. *OutputFormatter* calculates the index of the chunk, creates a *MapFile* as chunk file to store the chunk, and names chunk file with chunk's index. When all files are created, these chunks are logically combined into a cube as *Result*.

Fig. 5 illustrates the core operation of roll up as an example. Let d_1 and d_2 be dimensions-which are divided into p_1 and p_2 partitions respectively and the details are showed as below.

(i)	(ii)
$L(d_1) = \{{l_1}^1, {l_2}^1, {l_3}^1\}$	$L(d_2) = \{{l_1}^2, {l_2}^2, {l_3}^2, {l_4}^2\}$
$\lvert {l_1}^1\rvert = 16,\ \lvert {l_2}^1\rvert = 8,\ \lvert {l_3}^1\rvert = 1$	$\lvert {l_1}^2\rvert = 20,\ \lvert {l_2}^2\rvert = 10,\ \lvert {l_3}^2\rvert = 5,\ \lvert {l_4}^2\rvert = 1$
$p_1 = 4$	$p_2 = 4$

The cube is divided into 4^2 chunks. If the cell (index = 126, measure = 67) is processed, the index of processed chunk is 6. In *InputFormatter* the cell is selected and passed to *Mapper* and ${Key_M}^{In} = 126$, ${Value_M}^{In} = 67$. In *Mapper*, firstly ${Key_M}^{In} = 126$ is reverse-linearized to (10,6) according to Eqs. (5) and (6). If we want to roll up to ${l_2}^1$ of d_1 and ${l_2}^2$ of d_2, then the (10,6) is updated to (1,0) according to Eq. (1) and (2). Finally the ${Key_M}^{In} = (1,0)$ is linearized as ${Key_M}^{Out} = 1$ and emit with the value of measure, thus the output of *Mapper* is (${Key_M}^{Out} = 1$, ${Value_M}^{Out} = 67$). Let the other *Mapper* emit the value like (${Key_M}^{Out} = 1$, ${Value_M}^{Out} = 57$) and (${Key_M}^{Out} = 1$, ${Value_M}^{Out} = 43$), therefore the *Reducer* receives the input like (${Key_R}^{In} = 1$, ${Value_R}^{In} = \{67, 57, 43\}$). Let aggregation be *sum*() function, then output of *Reducer* is (${Key_R}^{Out} = 1$, ${Value_R}^{Out} = 167$). Finally the *OutputFormatter* generates a chunk file to the HDFS which contains a new cell (index = 1, measure = 167).

The MapReduce based data loading implementation includes two steps. The first step is loading data to distributed file system (HDFS as an example) as original files in a parse-able format and contain entire data of dimensions and measures, in which one line of a original file consists of measures and its dimensions. The second step is generating chunk files by a MapReduce job, shown in Fig. 6, which is similar with the MapReduce job of OLAP process. In the data loading job, *Mappers* reads data of original files. In a *Mapper*, ${Key_R}^{In}$ is a line number and ${Value_R}^{In}$ is a measure and its corresponding dimensions, in which dimensions are coded and linearized. ${Key_M}^{Out}$ is linearized code of dimensions and ${Value_M}^{Out}$ is the measures. Before the *Reducers* are started, ${Value_M}^{Out}$s are sorted by ${Key_M}^{Out}$. The number of *Reducers* is set to be the number of chunks, and the cells of same chunk are distributed to the same *Reducer* by *Partition*, so the outputs of each *Reducer* is a chunk file which is stored in HDFS. The chunk files are generated when job is completed, and original data files could be removed.

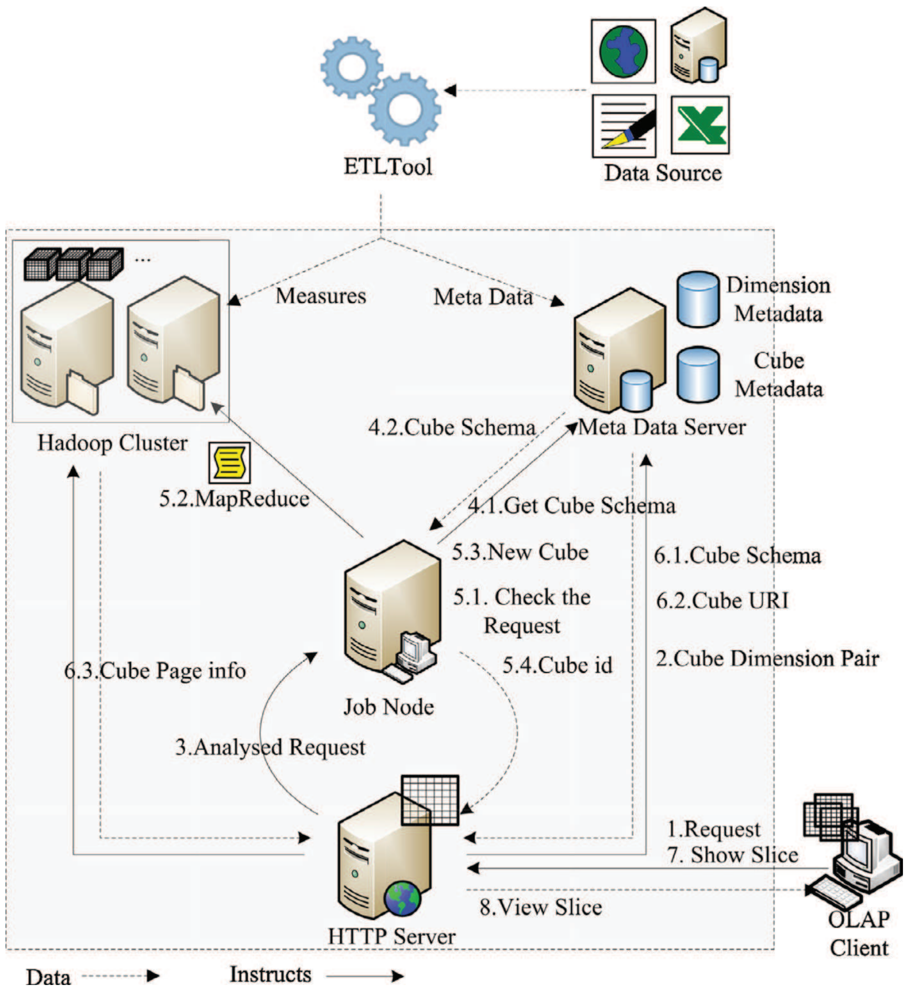


Fig. 7. HaoLap system architecture.

6. HaoLap system architecture

In order to improve OLAP performance and system scalability, HaoLap adopts share-nothing architecture. When OLAP query is processing, HaoLap not only reduces the queried scope but also ensures the parallelism, and distributes OLAP job to data to reduce the data transfer time. Fig. 7 shows the system architecture and the OLAP execution process. HaoLap includes five components, i.e., *Hadoop Cluster*, *Metadata Server*, *Job Node*, *OLAP Service Facade*, and *OLAP Client*. The ETL in Fig. 7 is a user-defined component to extract data from the specific data source.

6.1. *Hadoop Cluster*

Hadoop Cluster maintains a MapReduce instance and a HDFS instance, therefore measures are stored in the *Hadoop Cluster*. If the *Job Node* submits a MapReduce job of OLAP or data loading job, the *Hadoop Cluster* will run the job using MapReduce framework.

6.2. *Metadata Server*

Firstly, the metadata is stored in the *Metadata Server*, which includes two parts, such as dimension metadata and cube metadata. The dimension metadata describes the dimensional structure, which is persisted in XML file. Cube metadata is the cubes' information, such as identifiers, the file paths, and the dimensions of the cube, etc. Different from traditional MOLAP, dimensions and cube cost ignorable storage in HaoLap, which is explained in Section 7.5. Secondly, the *Metadata Server* is a metadata maintainer receiving instructions from *Job Node*.

6.3. *Job Node*

Job Node is the core of HaoLap that responds for checking processed OLAP commands, generating and monitoring OLAP jobs, and processing metadata with *Metadata Server*.

At first, *Job Node* receives an OLAP command from *OLAP Service Facade*, which is syntactically validated and converted into XML format. Then, *Job Node* checks the command according to the metadata from *Metadata Server* in order to ensure that it could be accomplished. If the command passes the checking process, *Job Node* will configure a MapReduce job, write the command into its configuration file, and submit the job to *Hadoop Cluster*. Finally, *Job Node* monitors the result of MapReduce job and reports it to *OLAP Service Facade*. At the same time, *Job Node* connects with *Metadata Server* to generate the metadata of the result cube.

6.4. *OLAP Service Facade*

OLAP Service Facade receives original request from *OLAP client*. Firstly, the *OLAP Service Facade* parses and validates the request, and generates an intermediate form of the request through inquiring

Metadata Server. Then, the *OLAP Service Facade* submits the intermediate form of the request to *Job Node* and waits for the result. Finally, the *OLAP Service Facade* reports the result to the *OLAP client*. The other responsibilities are retrieving the measures from *Hadoop Cluster*, generating the result view, and reporting to the *OLAP Client*.

OLAP Service Facade also provides REST interface so that the remote clients can communicate with HaoLap through HTTP protocol. The REST interface includes *Create Cube* (*Query*), *Save Cube*, *Delete Cube*, *Show Cube*, *Roll Up*, *Drill Down*, *Dice*, and *Slice*. We define that input of these operations contains the data cube specified by its identity, and the output of these operations, except *Delete Cube* and *Show Cube*, is stored in cube whose identity is returned. When *Roll Up*, *Drill Down*, *Dice*, and *Slice* are executed, *Show Cube* shows the result as a cube view, which is a section of a slice of a cube, only has two dimensions, a certain range of dimension values and measures.

6.5. OLAP Client

The *OLAP client* is an application that collects user's input, connects with the *OLAP Service Facade*, and reports the *OLAP Service Facade*'s response to the user. Meanwhile, *OLAP client* also response for showing the cube views.

7. Experiments

In this section, we design a series of test cases to evaluate the OLAP performance of HaoLap and compare it with a few cloud data warehouse systems. The section includes subsections of setup, loading, dice, roll up, and storage. In the setup section we describes the fundamental of the experiments including the purpose of the experiment and the experiment plain. The rest of the subsections are analysis of the experiment result.

7.1. Setup

7.1.1. Scope

Analyze HaoLap and other cloud data warehouse systems for the purpose of evaluation with respect to their performance of OLAP from the point of views of the loading, dice, roll up and storage in the context of 13 nodes physical machine cluster.

7.1.2. Experiment environment

We execute our experiments on the 13 nodes physical machine cluster with 8 GB memories, Inter Core i5 2.80 GHz, 1TB hard disk, network card, 64-bit platform, and moderate I/O performance. The network is connected by *Dell PowerConnect* 5548, a gigabit Ethernet switch.

7.1.3. Selection of competitors

We compare HaoLap with Hive (Thusoo et al., 2009), HadoopDB (Abouzeid et al., 2009), Olap4Cloud (akolyade, 2013), and HBaseLattice (dlyubimov, 2013). In order to cover the main types of the OLAP system based on MapReduce, we choose the competitors above. The main differences between the OLAP systems based on MapReduce are OLAP implement and storage design. The OLAP implement has two types, including ROLAP and MOLAP and the storage design has three types, including distributed file system, RDBMS and NoSQL database or data warehouse. Hive and HadoopDB are ROLAP systems, while HBaseLattice and Olap4Cloud are MOLAP systems. In storage design, Hive is based on HDFS. HadoopDB is based on PostgreSQL an open source RDBMS system, while HBaseLattice and OLAP4Cloud are based on HBase a NoSQL data warehouse system. The source code of HaoLap is available on https://github.com/MarcGuo/HaoLap.

7.1.4. Experiment data

We select the real applications of scientific data analysis. However, we do not take the domain of the applications as an impact factor of the experiments. The differences between the applications from different domain mainly include two aspect–the amount of the dimension values and the sparsity of measures. The reasons why we believe the domain of application do not affect the performance of HaoLap are listed as follows. (1) HaoLap exploits MapReduce framework to execute OLAP, therefore HaoLap does not instantiate the cube in the query process that leads to avoid massive amount of dimension values problem. (2) In HaoLap, the mapping of measures and dimensions is achieved by calculate that means the complexity of traversing dimension in HaoLap is $O(1)$, therefore the amount of dimension does not influence the performance of roll up in HaoLap. (3) HaoLap does not preserve the measure whose value is *NULL* that leads HaoLap avoid the sparity of measures. In summary, the differences between different domains do not influence the performance of HaoLap.

In experiments, we build a data cube named *OceanCube* according to practical dataset of oceanography data, which is collected by CTD and issued in our previous researches (Bao and Song, 2010; Song et al., 2009). From the oceanography data, we extracted three dimensions such as *Time*, *Area*, and *Depth*. Dimension of *Time* has five levels (excluding *ALL* level), such as *Year*, *Season*, *Month*, *Day*, and *Slot*. *Slot* means morning, afternoon and evening of a day. *Area* has 7 levels (excluding *ALL* level) such as 1°, 1/2°, 1/4°, 1/8°, 1/16°, 1/32°and 1/64°. 1° square is the region whose length of side is 1° of the longitude and the latitude. The earth could be divided into 360 × 180 1° squares and 4 × 360 × 180 1/2° squares and so on. Dimension of *Depth* has three levels, such as 100 m, 50 m, and 10 m. 100 m layer means the depth of ocean is divided in every 100 m. We assume that each month has 31 days, thus a few empty values are used to fill the invalid values. In the example, we assume that the cube contains data of 5 years, 5° squares, and 1000 m depth. The dimensions can be modeled as follows.

(i) $Time = \{\langle Year,10\rangle,\langle Season,4\rangle,\langle Month,3\rangle,\langle Day,31\rangle,\langle Slot,3\rangle\}$;
(ii) $Area = \{\langle 1°,5\rangle,\langle 1/2°,2\rangle,\langle 1/4°,2\rangle,\langle 1/8°,2\rangle,\langle 1/16°,2\rangle,\langle 1/32°,2\rangle,\langle 1/64°,2\rangle\}$;
(iii) $Depth = \{\langle 100\text{ m},10\rangle,\langle 50\text{ m},2\rangle,\langle 10\text{ m},5\rangle\}$.

Then we can calculate the size of each dimension as below.

(i) $|Time| = 11160$;
(ii) $|Area| = 320$;
(iii) $|Depth| = 100$.

Thus $|OceanCube|$ and $|OceanCube|^n$ are:

(i) $|OceanCube| = \langle 11160,320,100\rangle$;
(ii) $|OceanCube|^n = 357120000$.

Suppose we calculate out that:

(i) $|OceanChunk| = \langle 72,64,50\rangle$
(ii) $|OceanChunk|^n = 230400$.

More specifically, we have 1550 chunks and each chunk contains 230400 cells. Each cell contains *Temperature* of seawater as a measure. In Section 7, we adopted *OceanCube* in experiments.

We use semi-synthetic datasets (S_1, S_2 and S_3) generated from the practical dataset (ocean scientific data received by CTD (NMDIS, 2014)), whose data model is illustrated above. The size of each dataset is presented as $Size(S_i)$ $(1 \le i \le 3)$ whose unit is the data volume because the storage models of different data warehouses are

Table 3
Data sets description.

Dimension	Data set name		
	S_1	S_2	S_3
Time (year)	3	30	30
Area (1°)	1	1	5
Depth (100 m)	5	5	10
Size(S_i) (item)	10^8	10^9	10^{10}

Table 4
Symbols used in experiment illustration.

Symbol	Description
S_iC_j	A dice case which refers to the dice query C_j executed on dataset S_i
S_iU_j	A roll up case which refers to the roll up query U_j executed on dataset S_i
ResultSize()	The size of the result of the test case.
$Time_{[system]}()$	The time consumption of the case which is executed in the system

different and the size of data file are different too. The details of each datasets are descripted in Table 3.

7.1.5. *Experiment content*

The execution time and storage cost are essential aspects of OLAP performance, therefore we have designed two types of experiments. We compare HaoLap with Hive, HadoopDB, HBaseLattice and Olap4Cloud from the point of views of the loading, dice, roll up and storage. Firstly in the loading, dice and roll up experiments we measure the execution time of each operation on the systems, and meanwhile we analyze the reasons of the phenomena in the experiment to analyze the advantages of HaoLap on OLAP performance. When we execute each case for 10 times, and calculate the mean of all times' result as the final result which will appear in figure. Secondly in storage analysis, we evaluate and analyze the storage cost of HaoLap from the aspects of storing data and storing multidimensional model.

In the loading experiment, the generated data files (S_1, S_2 and S_3) are loaded to each system according to the loading approach of the system itself, but in order to measure the loading time we omit the loading analyze of the system whose loading process is manual process, e.g., Hive and HadoopDB.

In the dice experiment, three dice (slice) operations – C_1, C_2, and C_3 *are designed*. Each operation is executed on S_1, S_2, and S_3 respectively. We have nine cases for each system. The subscript *j* ($1 \le j \le 3$) of C_j stands for the dimension number, so if we use ROLAP implement, the number of joined tables of C_j would be $j + 1$ (*j* dimension tables and 1 fact table). The operations presented in SQL are shown as follows:

C_1	
SELECT	*Temperature.value*
FROM	*Temperature, Time*
WHERE	*Temperature.t_id* = *Time.t_id* **AND** *Time.year* = 2000
C_2	
SELECT	*Temperature.value*
FROM	*Temperature, Time, Area*
WHERE	*Temperature.t_id* = *Time.t_id* **AND** *Temperature.a_id* = *Area.a_id* **AND** (*Time.year* Between 2000 And 2001) **AND** (*Area.bound* Between 0 And 0.5)
C_3	
SELECT	*Temperature.value*
FROM	*Temperature, Time, Area, Depth*
WHERE	*Temperature.t_id* = *Time.t_id* **AND** *Temperature.a_id* = *Area.a_id* **AND** *Temperature d_id* = *Depth.d_id* **AND** (*Time.year* Between 2000 And 2002) **AND** (*Area.bound* Between 0 And 0.75) **AND** (*Depth.value* Between 1 And 230)

We denote each case as S_iC_j, and let $1 \le i \le 3$ and $1 \le j \le 3$ by default if not specified. The size of each result as *ResultSize*(S_iC_j) and the time consumption as $Time_{[system]}(S_iC_j)$ (*system* ∈ {*HaoLap*, *Hive*, *HadoopDB*, *HBaseLattice*, *Olap4Cloud*}). By default, the *ResultSize*(S_iC_j) of S_iC_j, expected S_3C_1 and S_3C_2, *is 3.5 million items approximately. The ResultSize*(S_iC_j) *of* S_3C_1 and S_3C_2 *are 35 million items and 10 million items respectively.*

In the *roll up experiment*, three roll up operations (U_1, U_2, and U_3) are designed. Each operation is executed on S_1, S_2 and S_3 respectively. The query conditions of U_j are same as C_j. In comparison with C_j, U_j has a grouping condition to aggregate measures from *Slot* level to *Month* level at the *Time* dimension. We represent the each case as S_iU_j ($1 \le i \le 3$, $1 \le j \le 3$), the size of each result as *ResultSize*(S_iU_j) ($1 \le i \le 3$, $1 \le j \le 3$) and the time consumption of each case on the system as $Time_{[system]}(S_iU_j)$ ($1 \le i \le 3$, $1 \le j \le 3$, *system* ∈ {*HaoLap*, *Hive*, *HadoopDB*, *HBaseLattice*, *Olap4Cloud*}). Because of the aggregation, the *ResultSize*(S_iU_j) is declined to 1% of *ResultSize*(S_iC_j).

In the storage experiment. Firstly, we compare the sizes of data set S_i, mentioned above, in each data warehouse. Then we introduce the approach of cube storage of HaoLap. Through the theoretical and experimental analysis, it is proven that the storage cost of cube in HaoLap is very low even if high dimensional dataset is involved.

7.1.6. *Symbols*

There are few symbols involved in the experiment analysis which are introduced in above illustrations. We list the symbols in Table 4.

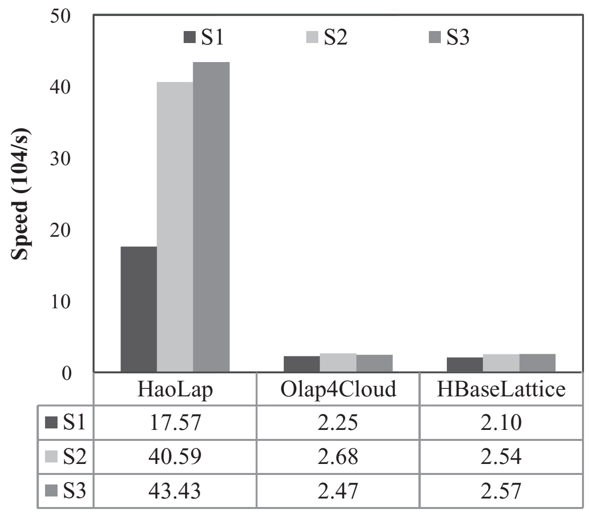
7.2. *Loading*

Since HadoopDB load data manually and Hive load data by uploading data file and modifying the metadata file, we do not compare loading performance of HadoopDB and Hive with the others. In order to import data to HaoLap, Olap4Cloud and HBaseLattice, we have to generate the data file in HDFS first. Then HaoLap and Olap4Cloud import data by MapReduce jobs, while HBaseLattice uses Pig (Apache, 2013a,b) scripts. In loading experiment, the loading performance of HaoLap is much higher than Olap4Cloud and HBaseLattice.

Fig. 8 describes the loading speed of each data warehouse system. According to Fig. 8 the loading speed of HaoLap is the fastest. Taking loading process of S_2 for example, HaoLap is 15× and 16× faster than Olap4Cloud and HBaseLattice respectively. The reasons of the phenomenon can be summarized as follows. (1) The data model of HaoLap is more simplified than Olap4Cloud and HBaseLattice. The models of the latter two are based on HBase and have to maintain many metadata such as table information and column storage model. (2) HaoLap does not need to generate index structure in data loading process because HaoLap achieves mapping of dimensions and measures by calculation, while Olap4Cloud and HBaseLattice need to generate an index table during data loading process.

7.3. *Dice*

Fig. 9 summarizes the overall time consumption of all cases. According to Fig. 9, HaoLap always performs better than the others except HBaseLattice. Fig. 10 illustrates the overall performance comparison of each system with HaoLap. According to Fig. 10 the performance of HaoLap is better than Hive, HadoopDB and

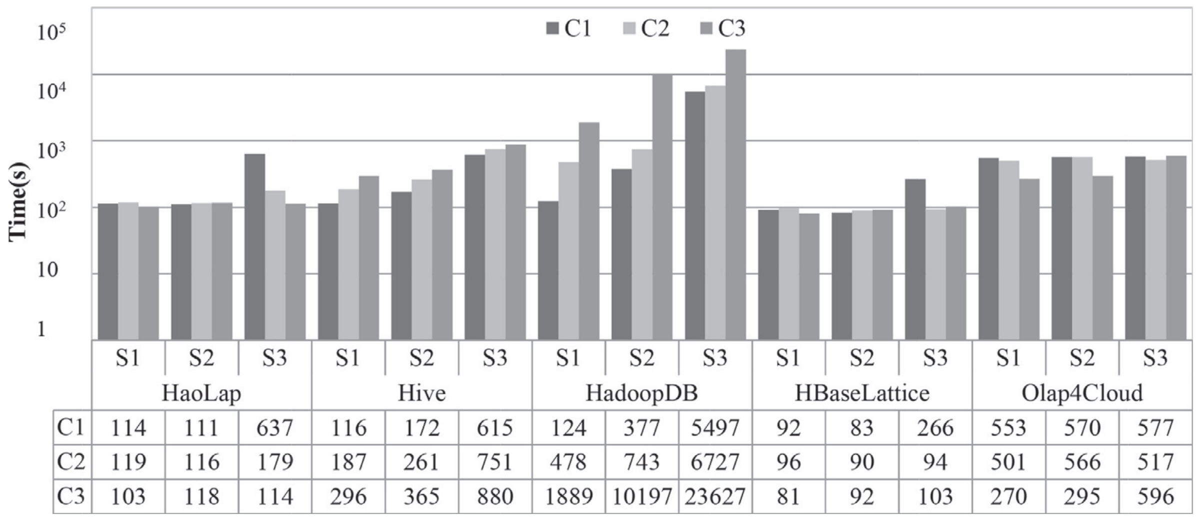


	HaoLap	Olap4Cloud	HBaseLattice
■ S1	17.57	2.25	2.10
■ S2	40.59	2.68	2.54
■ S3	43.43	2.47	2.57

Fig. 8. Comparison of loading speed among different system and cases.

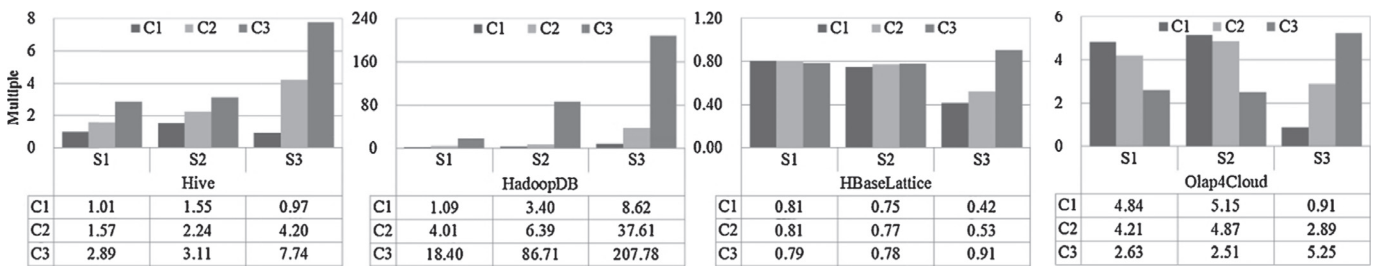
Olap4Cloud in most case, for example, in all cases the performance of HaoLap is 7×, 207× and 5× faster than that of Hive, HadoopDB and Olap4Cloud at the most. The averages of performance improvement of HaoLap are 2.81×, 41.56×, 3.69× compared with Hive, HadoopDB and Olap4Cloud respectively and meanwhile, the variances of each group of cases are 4.54, 4630.69 and 2.31 respectively. The situation can be explained as follows. (1) Hive and HadoopDB adopt ROLAP whose performance cannot compare with that of MOLAP because the join operations in ROLAP are costly. (2) In the small datasets and simple cases S_iC_j $(1 \le i \le 2, 1 \le j \le 2)$, the $\mathrm{Time}_{[\mathrm{Hive}]}(S_iC_j)$ and $Time_{[\mathrm{HadoopDB}]}(S_iC_j)$ are close to $Time_{[\mathrm{HaoLap}]}(S_iC_j)$ because of join optimization. Hive uses distributed cache to propagate hash-table file for map-side join operation, and HadoopDB use index to accelerate the join operation. Hive and HadoopDB adopt HDFS and DBMS as storage respectively, therefore they tightly integrated with local file system and enhance local data access and reduce network transmission. (3) $\mathrm{Time}_{[\mathrm{Olap4Cloud}]}(S_iC_j)$ is the worst one among MOLAP systems. Olap4Cloud adopts *HBase* as storage and depends on a precise index mechanism which is costly when dataset is large. (4) $Time_{[HBaseLattice]}(S_iC_j)$ is a little bit better than $Time_{[HaoLap]}(S_iC_j)$ because HBaseLattice do not need the mapping operation of dimensions and measures. In HBaseLattice, dimensions values and measures both are stored in the logically unitary table in which the mapping is immutable, but the price the advantage pays for is that the operations on dimension levels, such as roll-up and dill-down, are not supported. On the contrary, such operations are well supported in HaoLap by dimension coding and traverse algorithm.

Fig. 11a describes the performance tendency of HaoLap. It is concluded that $Time_{[HaoLap]}(S_iC_j)$ is steady except $Time_{[HaoLap]}(S_3C_2)$ is 1.5× larger and $Time_{[HaoLap]}(S_3C_3)$ is 6× larger than others. As we know, $ResultSize(S_3C_1)$ is about 35 million and $ResultSize(S_3C_2)$ is about 10 million, while the result size of rest cases are all about 3.5 million. Obviously in HaoLap the time consumption of dice operation relates to the size of the result set regardless of the size of dataset and the amount of dimensions involved. On one hand, given a certain operation, such as C_3, its performance is usually positively related to the amount of input data, while HaoLap provides chunk partition and selection algorithms, therefore in HaoLap the input of dice operation is selected chunks which are related to the operation and unchanged no matter how larger the data amount is. On the other hand, chunk selection algorithm is based on calculation but searching, the performance of chunk selection is



	HaoLap			Hive			HadoopDB			HBaseLattice			Olap4Cloud		
	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3
C1	114	111	637	116	172	615	124	377	5497	92	83	266	553	570	577
C2	119	116	179	187	261	751	478	743	6727	96	90	94	501	566	517
C3	103	118	114	296	365	880	1889	10197	23627	81	92	103	270	295	596

Fig. 9. The overall time consumption of dice operations (logarithmic scale).



Hive	S1	S2	S3
C1	1.01	1.55	0.97
C2	1.57	2.24	4.20
C3	2.89	3.11	7.74

HadoopDB	S1	S2	S3
C1	1.09	3.40	8.62
C2	4.01	6.39	37.61
C3	18.40	86.71	207.78

HBaseLattice	S1	S2	S3
C1	0.81	0.75	0.42
C2	0.81	0.77	0.53
C3	0.79	0.78	0.91

Olap4Cloud	S1	S2	S3
C1	4.84	5.15	0.91
C2	4.21	4.87	2.89
C3	2.63	2.51	5.25

Fig. 10. The overall performance comparison of each system with HaoLap.

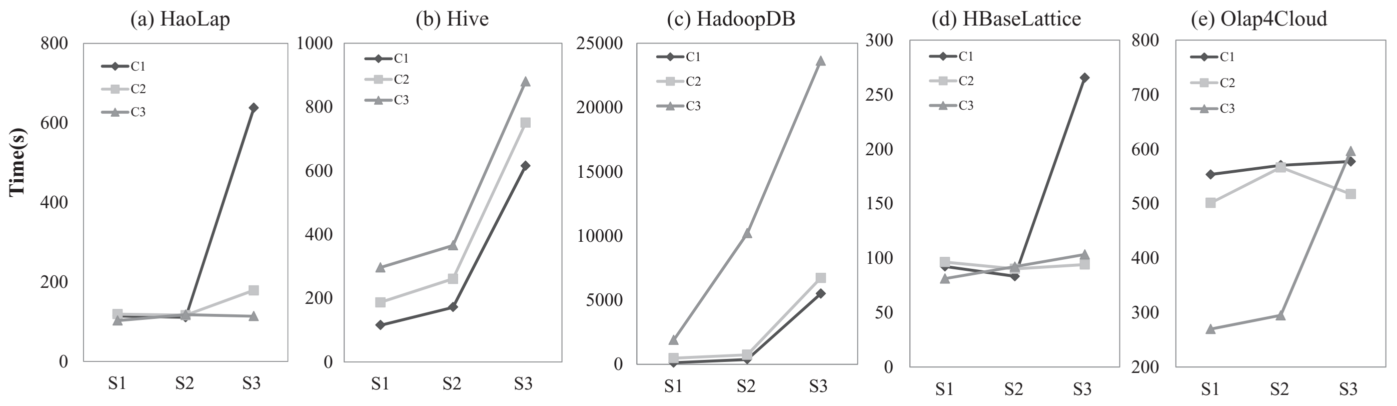


Fig. 11. Tendency of Dice operations.

unrelated to data amount too. Fig. 11d shows the performance tendency of HBaseLattice which is similar with that of HaoLap because both of them adopt MOLAP. Fig. 11e illustrates the performance tendency of Olap4Cloud, which is indistinctive. We analyzed the performance of Olap4Cloud in detail and prove that the underlying storage (HBase) dominates its performance for Olap4Cloud adopts MOLAP based on HBase. Therefore, the indistinctive performance tendency in Fig. 11e is actually follow the tendency of index size. The details are abbreviated in this paper.

Fig. 11b and c demonstrate the performance tendencies of Hive and HadoopDB respectively. Since these implementations are based on ROLAP, the number of join operations increases when the more dimensions are involved in the cases (C_1 to C_3), and join operation is more costly when joined tables is larger (S_1 to S_3) despite there are some join optimizations. As Fig. 11b shows, the dice performance of Hive is regular. $Time_{[Hive]}(S_iC_j)$ increases with the increment of $Size(S_i)$ and the case complexity (joined dimensions number). Hive only optimized map-side join by distributed cache that is suitable in limited cases while there are no index or partition mechanisms to accelerate the join and query operation, therefore the performance tendency of Hive follows the common regularity of ROLAP. HadoopDB is different from Hive for example HadoopDB adopts RDBMS as storage and it partitions data into each RDBMS and index them. In Fig. 11c, $Time_{[HadoopDB]}(S_2C_3)$ and $Time_{[HadoopDB]}(S_3C_3)$ is extreme larger compared with other cases, so tendencies of $Time_{[HadoopDB]}(S_iC_1)$ and $Time_{[HadoopDB]}(S_iC_2)$ are almost overlap. According to the details in data table of Fig. 9, it is concluded as follows. (1) Tendencies of $Time_{[HadoopDB]}(S_iC_1)$ and $Time_{[HadoopDB]}(S_iC_2)$ (i = 1, 2, 3) are follows the common regularity of ROLAP. HadoopDB adopts index to enhance query performance, so the time consumption of S_1C_1 and S_2C_1, S_1C_2 and S_2C_2 are close, while the benefits of index become less and less when data amount increases, therefore the time consumption of S_3C_1 and S_2C_1, S_3C_2 and S_2C_2 are quite different. (2) $Time_{[HadoopDB]}(S_2C_3)$ and $Time_{[HadoopDB]}(S_3C_3)$ is huge compared with others. HadoopDB adopts partition to accelerate join and avoid data transferring among node, but the benefits become less and less when more and more tables are joined, therefore the tendency of $Time_{[HadoopDB]}(S_iC_3)$ dramatic rise.

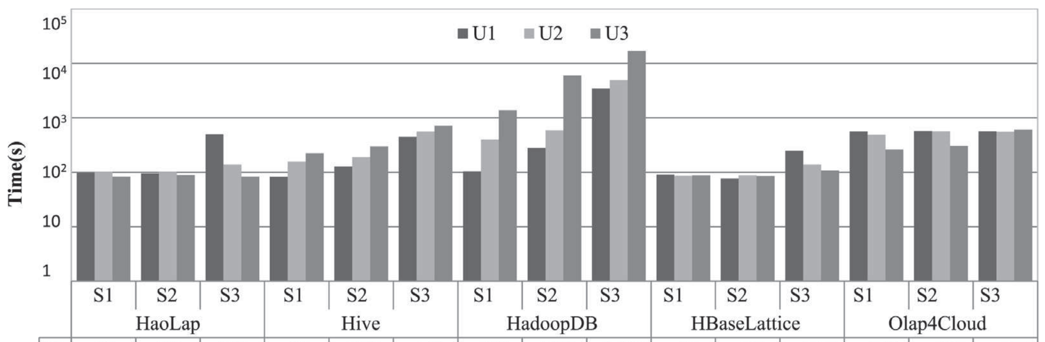
In conclusion, HaoLap has a great advantage in dice performance regardless of the size of the data set and the number of involved dimensions.

7.4. *Roll up*

The overall time consumption of roll up cases is illustrated as Fig. 12 and the tendency of each case on the each system is demonstrated as Fig. 13a–e. Actually, the performance of roll up operation is accord with the performance of the dice operation. According to Fig. 12 (logarithmic scale), HaoLap always performs better than others except HBaseLattice. The tendency of Fig. 13a–e is also exactly accord with Fig. 11a–e, respectively, because the aggregation operation is a local operation on result data, it does not affect the performance regularity.

7.5. *Storage*

The comparisons of the storage costs of each data warehouse are displayed in Fig. 14. Taking dataset S_3 for example, the storage cost of Olap4Cloud is eight times more than that of HaoLap



	HaoLap			Hive			HadoopDB			HBaseLattice			Olap4Cloud		
	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3
U1	98	95	501	82	128	447	103	280	3436	90	76	246	562	573	568
U2	102	101	138	157	190	559	397	589	4906	86	87	138	486	565	554
U3	83	89	83	223	298	707	1378	6007	16980	87	86	107	262	306	605

Fig. 12. The overall time consumption of roll up operations (logarithmic scale).

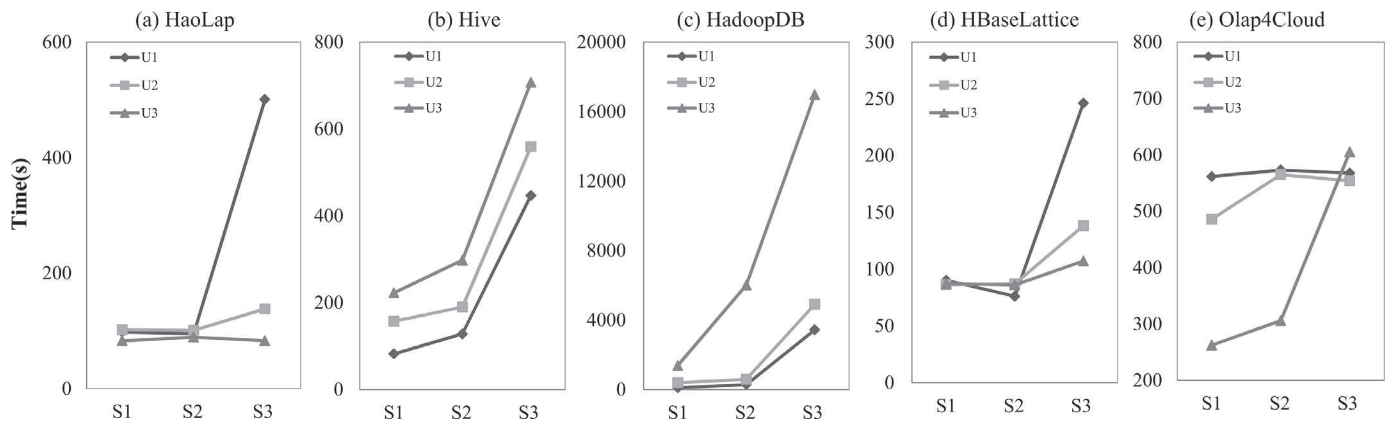
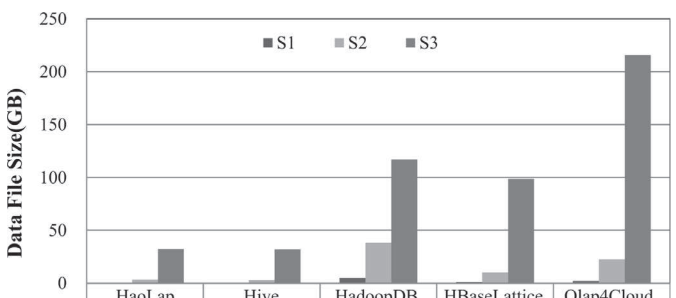


Fig. 13. Tendency of roll up operations.

and Hive and as twice as that of HadoopDB. The storage cost of HBaseLattice is three times more than that of HaoLap and Hive but it is smaller than that of HadoopDB. HaoLap and Hive consume less storage than other systems because they directly adopt the original file format of distributed file system (HDFS) to store data, and meanwhile their storage structures are simple and no extra metadata need to be stored. The storage cost of Olap4Cloud is higher than that of other systems, because Olap4Cloud is based on HBase, which consumes large space to store index and metadata, and meanwhile Olap4Could need to generate index table for optimization. HadoopDB adopts RDBMS as physical storage. In order to optimize join operation, HadoopDB adopts a hash algorithm to divide the fact table, which is stored in RDBMS of each data node. Meanwhile HadoopDB also copies dimensions into each data node. The operations above bring the extra storage cost for HadoopDB. Besides, in HaoLap, the size of S_2 is ten time large than S_1, and the storage of S_2 is also ten times larger than that of S_1. The situation indicates linear relation between the storage cost and the data size in HaoLap.

The storage cost of cube in HaoLap is very low even can be ignored. In order to accommodate to the big data environment, OLAP of HaoLap is different from traditional MOLAP in essence. HaoLap does not store large multidimensional array but calculation. According to the description of Section 3, HaoLap simplifies the dimension in which the codes of dimension values at the same level are continuous and each sibling of node in the hierarchy of dimension values has the same number of descendants. The approach greatly reduces the storage cost of dimension because each dimension of HaoLap only contains its name and scale. Taking the time dimension mentioned in Section 7.1 for example, only a piece of data {⟨*Year*,10⟩,⟨*Season*,4⟩, ⟨*Month*,3⟩,⟨*Day*, 31⟩, ⟨*Slot*,3⟩} is stored



	HaoLap	Hive	HadoopDB	HBaseLattice	Olap4Cloud
■ S1	0.33	0.28	4.91	0.98	2.16
■ S2	3.23	3.00	38.41	9.99	22.57
■ S3	32.30	31.89	117.02	98.56	215.70

Fig. 14. Storage comparisons of HaoLap, Hive, HadoopDB, HBaseLattice, and Olap4Cloud.

in *NameNode* and loaded into memory at runtime. HaoLap seeks the cell through dimension coding and traversal algorithms. Meanwhile HaoLap achieves the mapping from cell to physical location (*MapFile*) by linearization and cube partition algorithms, and adopts MapReduce to accomplish data query and aggregation. Therefore, we can conclude that HaoLap does not instantiate the cube. The experiment proves that the size of data objects in memory is just 50.3 kB if we store dimension of Time one thousand times as a high dimensional cube. The analysis and experiment above show that the storage cost of cube of HaoLap can be ignored.

8. Conclusions

This paper presents the design, implementation, and evaluation of HaoLap, an OLAP system for big data. HaoLap is built on Hadoop and based on the proposed models and algorithm: (1) specific multidimensional model to map the dimensions and the measures; (2) the dimension coding and traverse algorithm to achieve the roll up operation on hierarchy of dimension values; (3) the chunk model and partition strategy to shard the cube; (4) the linearization and reverse-linearization algorithm to store chunks and cells; (5) the chunk selection algorithm to optimize OLAP performance; and (6) the MapReduce based OLAP and data loading algorithm. We compared HaoLap performance with Hive, HadoopDB, HBaseLattice, and Olap4Cloud on several big datasets and OLAP applications.

In experiments, we compared the performance of loading, dice, and roll up on HaoLap, Hive, HadoopDB, HBaseLattice, and Olap4Cloud. Our experimental results prove that: (1) HaoLap has performance advantages of loading data; (2) compared with other ROLAP or MOLAP system, HaoLap has a great advantage in the OLAP performance regardless of the size of data sets and number of involved dimensions; (3) the OLAP performance of HaoLap is steady and efficient; (4) HaoLap does not introduce additional storage cost. There are some reasons leading to the advantages. (1) HaoLap draws the experiment of MOLAP, therefore the high performance of OLAP in HaoLap is nature. (2) It is same as works done by Yu et al. (2011) and D'Orazio and Bimonte (2010), HaoLap takes advantages of special data model, simplified dimension model that the basis of HaoLap, to keep the OLAP simple and efficient. (3) The dimension coding and traversing algorithms maps the dimension values with measures in an efficient calculation method but heavy index approach. In OLAP, HaoLap adopts chunk partition and selection methods to boost the OLAP speed to accommodate to the big data environment. (4) Owing to the simplified dimension model, the loading process of HaoLap is efficient. In summary, the performance of HaoLap is ensured by keeping the OLAP simply and efficient. (5) It is same as the work done by Tian (2008), HaoLap exploits MapReduce to execute OLAP that leads to the keep query process parallel and efficient.

In future, we would optimize HaoLap in the following aspect: discuss whether chunks should be compressed, how to compress them, how to perform OLAP on compressed cube, and whether the compression affects the OLAP performance. The other aspect is taking advantage of pre-computation, which is utilized in the work done by Jinguo et al. (2008).

Acknowledgements

This research is supported by the National Natural Science Foundation of China (61202088); the Fundamental Research Funds for the Central Universities (N120817001); the Science Foundation of China Post-doctor (2013M540232); the National Research Foundation for the Doctoral Program of Higher Education of China (20120042110028).

References

Abelló, A., Ferrarons, J., Romero, O., 2011. Building cubes with MapReduce, In: Proceedings of the ACM 14th International Workshop on Data Warehousing and OLAP. ACM, pp. 17–24.

Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., et al., 2009. HadoopDB: an architectural hybrid of mapreduce and DBMS technologies for analytical workloads. Proc. VLDB Endow. 2, 922–933.

akolyade, 2013. Olap4Cloud Home Page accessed at http://code.google.com/p/olap4cloud/.

Apache, 2013. Hadoop Home Page accessed at http://hadoop.apache.org.

Apache, 2013. Pig Home Page accessed at http://pig.apache.org/.

Bao, Y., Song, J., 2010. Case study on navigating data elements by faceted classifications in scientific data warehouse, In: IEEE International Conference on Computer Application and System Modeling (ICCASM) pp. V2-238-V232-V242.

Bolosky, W.J., Douceur, J.R., Ely, D., et al., 2000. Feasibility of a Server Less Distributed File System Deployed on an Existing Set of Desktop PCs. Assoc Computing Machinery, New York.

Chang, F., Dean, J., Ghemawat, S., et al., 2008. Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) 26, 4.

Chaudhuri, S., Dayal, U., 1997. An Overview of Data Warehousing and OLAP Technology, vol. 26. ACM Sigmod Record, pp. 65–74.

Chaudhuri, S., Dayal, U., Narasayya, V., 2011. An Overview of Business Intelligence Technology, vol. 54. Communications of the ACM, pp. 88–98.

Dean, J., Ghemawat, S., 2008. Mapreduce: Simplified Data Processing on Large Clusters, vol. 51. Communications of the ACM, pp. 107–113.

dlyubimov, 2013. HBaseLattice Home Page accessed at https://github.com/dlyubimov/HBase-Lattice.

D'Orazio, L., Bimonte, S., 2010. Multidimensional arrays for warehousing data on clouds, In: 3rd International Conference on Data Management in Grid and Peer-to-Peer Systems, Globe 2010. Springer Verlag, Bilbao, Spain, September 1–2, 2010, pp. 26–37.

Goil, S., Choudhary, A., 1998. High performance data mining using data cubes on parallel computers, In: IEEE Computer Society Proceedings of the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing. Orlando, FL, USA, March 30–April 3, 1998, pp. 548–555.

Gray, J., Liu, D.T., Nieto-Santisteban, M., et al., 2005. Scientific data management in the coming decade. SIGMOD Rec. 34, 34–41.

Jing-hua, Z., Ai-mei, S., Ai-bo, S., 2012. OLAP aggregation based on dimension-oriented storage, In: IEEE 26th International Symposium on Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), pp. 1932–1936.

Jinguo, Y., Jianqing, X., Pingjian, Z., 2008. A parallel algorithm for closed cube computation, In: 7th IEEE/ACIS International Conference on Computer and Information Science (ICIS'08). Piscataway, NJ, USA, May 14–16, 2008, pp. 95–99.

Khetrapal, A., Ganesh, V., 2006. HBase and Hypertable for Large Scale Distributed Storage Systems. Dept. of Computer Science, Purdue University.

Lai, Y., ZhongZhi, S., 2010. An efficient data mining framework on Hadoop using Java persistence API, In: IEEE 10th International Conference on Computer and Information Technology (CIT), pp. 203–209.

Leonardi, L., Orlando, S., Raffaeta, A., et al., 2014. A general framework for trajectory data warehousing and visual OLAP. Geoinformatica 18, 273–312.

Lima, A., Furtado, C., Valduriez, P., et al., 2009. Parallel OLAP query processing in database clusters with data replication. Distrib. Parallel Databases 25, 97–123.

Lin, W.Y., Kuo, I.C., 2004. A genetic selection algorithm for OLAP data cubes. Knowl. Inf. Syst. 6, 83–102.

Miller, H.J., 2010. The Data Avalanche is here. Shouldn't we be digging?. J. Reg. Sci. 50, 181–201.

NMDIS, 2014. China Oceanic Information Network accessed at http://mds.coi.gov.cn/jcsj.asp.

Ravat, F., Teste, O., Tournier, R., et al., 2007. Graphical querying of multidimensional databases. In: Ioannidis, Y., Novikov, B., Rachev, B. (Eds.), Proceedings on Advances in Databases and Information Systems, pp. 298–313.

Riha, L., Malik, M., El-Ghazawi, T., 2013. An adaptive hybrid OLAP architecture with optimized memory access patterns. Clust. Comput. J. Netw. Softw. Tools Appl. 16, 663–677.

Sarawagi, S., Stonebraker, M., 1994. Efficient organization of large multidimensional arrays, In: IEEE Proceedings of the 10th International Conference on Data Engineering. Houston, TX, USA, February 14–18, 1994, pp. 328–336.

Shan, W., Ju1, W.H., Pai, Q.X., et al., 2011. Architecting big data: challenges, studies and forecasts. Chin. J. Comput. 34, 1741–1752.

Shim, J.P., Warkentin, M., Courtney, J.F., et al., 2002. Past, present, and future of decision support technology. Decis. Support Syst. 33, 111–126.

Shvachko, K., Kuang, H., Radia, S., et al., 2010. The hadoop distributed file system, In: IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10.

Song, J., Bao, Y., Shi, J., et al., 2009. Case study on multi-classifications based scientific data management and analysis system, Ninth IEEE International Conference on Computer and Information Technology, CIT'09, pp. 272–277.

Song, Z.Y., Min, J., Wei, W.Z., et al., 2011. One-size-fits-all OLAP technique for big data analysis. Chin. J. Comput. 34, 1936–1946.

Song, J., Li, T., Liu, X., et al., 2012. Comparing and analyzing the energy efficiency of cloud database and parallel database, 2nd International Conference on Computer Science, Engineering and Applications, ICCSEA 2012, vol. 2. Springer Verlag, New Delhi, India, May 25–27, 2012, pp. 989–997.

Song, J., Guo, C.P., Wang, Z., Zhang, Y.C., Yu, G., Pierson, J.M., 2014. Distributed MOLAP technique for Big Data analysis. Ruan Jian Xue Bao/Journal of Software 25 (4), 731–752. doi:10.13328/j.cnki.jos.004569.

Taleb, A., Eavis, T., Tabbara, H., 2013. Query optimization for the NOX OLAP algebra, Transactions on Large-Scale Data-and Knowledge-Centered Systems VIII. Springer, pp. 53–88.

Thusoo, A., Sarma, J.S., Jain, N., et al., 2009. Hive: a warehousing solution over a map-reduce framework. Proc. VLDB Endow. 2, 1626–1629.

Tian, X., 2008. Large-scale SMS messages mining based on map-reduce, IEEE International Symposium on Computational Intelligence and Design. Piscataway, NJ, USA, October 17–18, 2008, pp. 7–12.

Wikipedia, 2014. Online Analytical Processing accessed at http://en.wikipedia.org/wiki/OLAP.

Wu, L., Sumbaly, R., Riccomini, C., et al., 2012. Avatara: OLAP for webscale analytics products. Proc. VLDB Endow. 5, 1874–1877.

Wu, S., Ooi, B.C., Tan, K.-L., 2013. Online Aggregation, Advanced Query Processing. Springer, pp. 187–210.

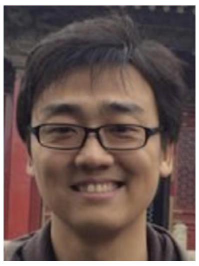
Xiaofeng, M., Xiang, C., 2013. Big data management: concepts, techniques and challenges. J. Comput. Res. Dev. 50, 146–169.

Yu, C., Chun, C., Fei, G., et al., 2011. ES2: a cloud data storage system for supporting both OLTP and OLAP, 27th IEEE International Conference on Data Engineering (ICDE 2011). Piscataway, NJ, USA, 11–16 April, 2011, pp. 291–302.

Jie Song received the Ph.D. degree in computer science from Northeastern University of China in 2008. He has been an associate professor of software college in Northeastern University since 2010. His main research interests include big data management, data intensive computing, and energy efficient computing.

Chaopeng Guo received the B.E. degree in software engineering from Northeastern University of China in 2013. He is a master student of software engineering in Northeastern University and will graduate in June, 2015. He researches on the data intensive computing and iterative computing currently.

Zhi Wang received the B.E. degree in software engineering from Northeastern University of China in 2013. He is a master student of software engineering in Northeastern University and will graduate in June, 2015. He researches on the cloud computing and its application currently.



Yichuan Zhang received the Ph.D. degree in computer science from Northeastern University of China in 2012. He is a lecture of software college in Northeastern University. His main research interests include data intensive computing and iterative computing.

Ge Yu received his Ph.D. degree in computer science from Kyushu University of Japan in 1996. He has been a professor at Northeastern University of China since 1996. His research interesting includes database theory and technology, distributed and parallel systems.

Jean-Marc Pierson received his Ph.D. in computer science from Ecole Normale SupErieure de Lyon. He has been a professor of the Paul Sabatier University, Toulouse 3 since 2006. His research interesting includes distributed systems and high performance computing, distributed data management, and energy aware distributed computing.