MRDataCube: Data Cube Computation Using MapReduce

Suan Lee
Department of Computer Science
Kangwon National University
192-1 Hyoja-dong,
Chuncheon, Kangwon, Korea
Email: suanlee@kangwon.ac.kr

Sunhwa Jo
Department of Computer Science
Kangwon National University
192-1 Hyoja-dong,
Chuncheon, Kangwon, Korea
Email: jsunf@kangwon.ac.kr

Jinho Kim
Department of Computer Science
Kangwon National University
192-1 Hyoja-dong,
Chuncheon, Kangwon, Korea
Email: jhkim@kangwon.ac.kr

***Abstract*—Data cube is used as an OLAP (On-Line Analytical Processing) model to implement multidimensional analyses in many fields of application. Computing a data cube requires a long sequence of basic operations and storage costs. Exponentially accumulating amounts of data have reached a magnitude that overwhelms the processing capacities of single computers. In this paper, we implement a large-scale data cube computation based on distributed parallel computing using the MapReduce (MR) computational framework.**

For this purpose, we developed a new algorithm, MRDataCube, which incorporates the MR mechanism into data cube computations such that effective data cube computations are enabled even when using the same computing resources. The proposed MRDataCube consists of two-level MR phases, namely, MRSpread and MRAssemble. The main feature of this algorithm is a continuous data reduction through the combination of partial cuboids and partial cells that are emitted when the computation undergoes these two phases. From the experimental results we revealed that MRDataCube outperforms all other algorithms.

***Keywords*—cube; OLAP; multi-dimensional analysis; data cube computation; MapReduce; Hadoop; distributed parallel algorithm;**

I. Introduction

The amount of data generated continues to increase exponentially, and with the ever-growing magnitude of accumulation, it is becoming increasingly important to perform rapid analyses of large datasets for enabling rapid decision-making. The OLAP (On-Line Analytical Processing) concept for multidimensional data analysis has been widely applied in a large variety of fields, with the data cube being its representative model [1]. Once a data cube is constructed, aggregate operations and analyses can be performed in multiple dimensions. In other words, a rapid data cube computation is a key factor determining a rapid multidimensional analysis and the corresponding decision-making[2], [3].

One of the intrinsic features of a data cube is its high computational costs. An increase in data amounts exponentially boosts the computational costs. A single computer is overwhelmed not only by the size of a dataset, but also by the multidimensionality of the analysis. In fact, a large workable data cube cannot be implemented without the use of distributed parallel processing. However, distributed parallel processing has a priori to solve problems, such as an interconnection of distributed resources, communication processing, and the means of fast expansion [4], [5], [6]. To address this problem, we use an MR framework capable of easily constructing distributed parallel computing environments [7], [8], allowing large data cubes to be processed in a distributed and parallel manner [9], [10].

Existing MR-based data cube algorithms have fundamental problems such as high costs and limited capacities. An increase in the data amount incurs an exponential increase in the processing costs, and there are even cases in which processing becomes impossible. While increasing the number of nodes can bring about computational gain, the number cannot be increased ad infinitum owing to excessively high costs. An ideal solution would be a new algorithm capable of effectively performing cube computations of large datasets using the same computing resources. To find such an ideal algorithm, we carried out a long series of experiments applying one existing data cube algorithm after another. As a result, we determined that it is crucial to construct a data cube computation algorithm that optimally matches the distributed parallelism of MR.

This paper proposes MRDataCube, a distributed parallel data cube algorithm optimized for the MR architecture. MRDataCube has proven its superiority over all other existing data cube algorithms. Its computation is run using two-level MR phases comprising MRSpread and MRAssemble, and the input data continuously undergo an amount reduction while passing through each phase based on the concepts of a partial cuboid and partial cell. The experiment results clearly demonstrate that the MRDataCube algorithm can construct cubes for large datasets in a rapid and cost-effective manner. The algorithm can also be used efficiently in a variety of fields where large-scale data cubes are required.

II. Preliminaries

This chapter describes a data cube and MR, which constitute the basic concepts. Fig. 1 shows an example of raw data comprising four attributes, represented by A, B, C, and D, used for a general understanding of this paper.

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c1	d3
a1	b1	c2	d1
a1	b2	c2	d2
a1	b2	c2	d3

Fig. 1. Example of raw data

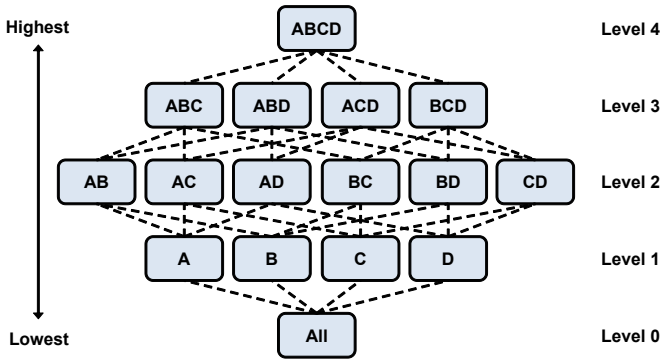

Fig. 2. An example of cube lattice

A. Data Cube Model

A data cube is a data model or data structure utilized for an online presentation of the results of a multidimensional analysis of a large dataset for support in decision-making. A data cube is composed of measurement values pre-computed by applying dimension attributes as the analysis perspectives, as well as various aggregate operations. A data cube performs aggregate operations to extract all possible combinations of the dimension attributes of a given raw dataset, and stores them to provide rapid responses to any analysis queries.

1) Cuboid: A data cube performs aggregate operations to extract each attribute combination; the aggregate result of each computation, called a cuboid, is identical with that of a group-by. For example, an AD cuboid can be expressed using the following SQL query:

SELECT A, D FROM R GROUP BY A, D;

A data cube with four dimensions of A, B, C, and D has the cuboids of $ABCD$, ABC, ABD, ACD, BCD, AB, AC, AD, BC, BD, CD, A, B, C, and D, and a total of $2^4 = 8$ cuboids are extracted. All cuboid denotes dimensionless combinationsİn general cuboid C is expressed in the form of $(A_1, A_2, \cdots, A_n, M)$, where the dimension attribute A_i has n-dimension attributes, and M denotes a measurement.

2) Cell: Cells constituting a cuboid represent the values stored in the given cuboid, and they can be expressed in the form $(a_1, a_2, \cdots, a_n, m)$, where m denotes the aggregate measurement of a cuboid cell. A data cube takes an architecture comprising such cells. As an example, the cells in cuboid AC in Fig. **??** are expressed as $(a_1, *, c_1, *, 3)$ and $(a_1, *, c_2, *, 3)$, where $*$ denotes the empty dimension (attribute) of a cuboid.

3) Cube Lattice: A data cube has a lattice formation with a directed acyclic graph (DAG) architecture in which all cuboids are expressed in a parent-child relationship. The highest and lowest positions are taken by those cuboids with the highest and lowest or zero dimensions, respectively. Fig. 2 is an example of a four-dimensional cube lattice, where $ABCD$ is called a top cuboid and All indicates an empty cuboid.

B. MapReduce

MapReduce (MR) is a distributed parallel processing software framework developed to facilitate the processing of large-scale datasets. Google published a paper dealing with MR-based distributed parallel processing[7]. Subsequently, based on the contents of this paper, the MR framework was developed as part of the Hadoop project[8].

The basic architecture of MR consists of four phases: a map phase, a combine phase, a shuffle phase, and a reduce phase. The basic unit used in MR has the form $\langle key; value \rangle$ pair, in which key serves as the standard for a basic array and value denotes the key value, similar to a hash structure.

1) Map: The input data received by the map function are the outputs from splitting all raw data. An input reader reads each input file record by record and converts it into a $\langle key; value \rangle$ pair form. Mappers operate in parallel for each split file, whereby the number of mappers is determined by the file number or size. Mappers usually emit key-value pairs in easily computable forms or in a specified amount by filtering the inputted records.

2) Combine: The combine function receives the outputs of the map function, and its output is inputted into the reduce function. The combine function plays an important role in reducing the amount of data to be further processed during the sort and shuffle phase. Its function is to combine the data with the same key in the data emitted by the mappers.

3) Shuffle: The shuffle function is arrayed in alignment with each key of the inputted data, whereby all data with the same key are merged together and emitted in the form of $\langle key; [value_1, value_2, \cdots] \rangle$.

4) Reduce: MR ensures a key-based array during the shuffle phase and receives the data arrayed in the reduce function as input data. Each reducer then writes the data emitted during the shuffle phase.

C. Motivation

Data cube computations require a significant number of processing steps and high storage costs. The increase in the size of the input data for a data cube computation sharply increases the computation costs. If we let T be the number of tuples (records) of input data and D be the number of dimensions (attributes), then the maximum cost required is $T \times 2^D$.

The constantly growing size of the data to be analyzed for decision-making compels analyzers to speed up the computa-

tion speed for rapid decision-making. A single-node computation can no longer deal with an exponential growth of a data cube from increases in T and D of the input data for real-life cube computations. A distributed parallel process has therefore become indispensable for rapid computations. We chose MP as a distributed parallel framework to speed up the data cube computations. However, the following problems arise when calculating a data cube using an MR-based computation model: (i) size-dependent high processing costs owing to the cost-intensive shuffle phase, (ii) failure of single-node computing or an excessive processing time for computing large-scale data, and (iii) an increase in input-output costs owing to the increased number of phases when large-scale data are divided to solve the problem described in (ii).

An efficient MR-based data cube computation can be ensured by reducing the output amount of the map function and the data processing amount during the shuffle phase. By fully exhausting the reduction potential using the MR mechanism, we developed MRDataCube, a new algorithm capable of computing a data cube at low cost. Unlike other data cube MR algorithms studied to date, which extended existing data cube computation algorithms to enable MR applications, we focused on developing a data cube algorithm tailored to the MR architecture. As a result, the proposed MRDataCube is an algorithm that can perform rapid data cube computations based on the PR mechanism.

III. MRDataCube

A. Overview of the MRDataCube

The basic architecture of MRDataCube consists of two basic phases, MRSpread and MRAssemble, with which all cuboids can be computed and cubes can be constructed. Fig. 3 provides an overview of the operational steps of the MRDataCube algorithm. During the MRSpread phase, all partial cells p are emitted, and during the MRAssemble phase, all partial cuboids P are computed and emitted as cuboid C.

a) Definition 1: A partial cell means an incomplete cell, and a complete cell means the sum of all partial cells. A partial cell $p_1 = (a_1, a_2, \cdots, a_3, m_1)$ may share the same set of cell attributes with another partial cell $p_2 = (a_1, a_2, \cdots, a_3, m_2)$, where m1 and m2 are partial measurements. The condition for the status of a complete cell is having only one cell that shares the same attribute. A complete cell has the form (a_1, a_2, a_3, M), where measurement M has the final aggregate result using function $F(m_1, m_2)$, which is the same as $sum()$.

b) Definition 2: A partial cuboid is a set of partial cells to be computed. Partial cuboid $P_j = p_1, p_2, \cdots, p_n$ for attribute $j = A_1, A_2, \cdots, A_3$ is a partial cuboid with one or more partial cells $p(p \subseteq P)$. Here, n denotes the total number of cells within $P_j (1 < n < P_j)$, and P_j contains one or more identical partial cells. If all partial cells $p_1, p_2, \cdots, p_n$ are completely computed, their values are the same as that of cuboid $C_j (P_j = C_j)$.

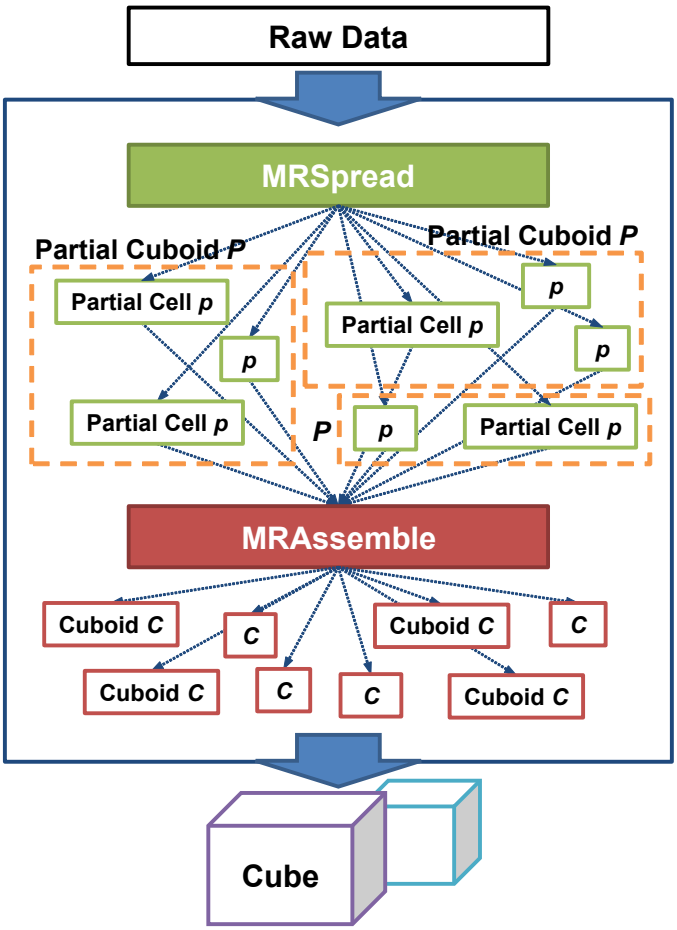

Fig. 3. Overview of the MRDataCube algorithm

B. MRDataCube Computation

MRDataCube computes a data cube through a two-phase MR process consisting of MRSpread and MRAssemble. The MR process is divided into these two phases for two reasons: 1) maximum use of the combined performance of MR, and 2) minimization of the data amount to be computed during the shuffle phase. MRDataCube basically has two phases: combine and shuffle.

In MRDataCube, raw data read by each mapper are reduced into an aggregated form by the MRSpread combiners. Further data reduction is implemented during the shuffle process by grouping data pertaining to other mappers by cell unit and arraying them again in an aggregated form. Reducers then generate partial cuboids in a fully computed cuboid. These partial cuboids are fed to MRAssemble as inputs and undergo the combine process, before moving from the map function to the shuffle process, in which the data are further reduced through a partial aggregation of the same cells in their respective partial cuboids. These reduced cell data are emitted as final cuboids using the shuffle process and reduce function.

Algorithms 1, 2, 3, 4, and 5 show the computation algorithms of MRDataCube. This algorithm uses the

MRSpreadMapper(), MRCombiner(), MRSpreadReducer(), MRAssembleMapper(), MRAssembleReducer(), and MRDataCube() procedures.

MRDataCube() launches the job execution by establishing an MR-related environment and carries out two MR phases: MRSpread and MRAssemble. In the former, after performing MRSpreadMapper() along with MRCombiner, reducers perform an MRSpreadReducer() function. In the latter, partial cells are read through MRAssembleMapper() functions, and identical cells are combined using an MRCombiner() function, followed by the completion of all cuboids by an MRAssembleReducer() function.

Algorithm 1 MRSpreadMapper

```
1: procedure MRSPREADMAPPER
2:     for all Cell c in top cuboid do
3:         cellc ← t                                  ▷ t is a tuple
4:         measurem ← measure ofc
5:         emit(cell, measure)
6:     end for
7: end procedure
```

Algorithm 2 MRCombiner

```
    procedure MRCOMBINER
2:      M ← function(m_1, m_2, ···, m_n)
            ▷ each MRSpreadMapper() emits (m_1, m_2, ···, m_n)
4:      emit(cell, measure)
    end procedure
```

Algorithm 3 MRSpreadReduce

```
    procedure MRSPREADREDUCE
        M ← function(M_1, M_2, ···, M_n)
3:          ▷ each MRCombiner() emits (M_1, M_2, ···, M_n)
        emit(cell, M)                                  ▷ top cuboid
        for all Partial cuboid P do
6:          p ← partial cell inP
            emit(p, M)
        end for
9: end procedure
```

Algorithm 4 MRAssembleMapper

```
    procedure MRASSEMBLEMAPPER
        emit(cell, measure)
    end procedure
```

C. MRSpread: Spread the Cells

MRSpread has the function of emitting to all partial cells the raw data need. To enable a data cube calculation, MRSpread consists of two preparatory processing steps of data reduction and partial cuboid generation. As a result of the MRSpread phase, both the top cuboid and partial cuboids are generated.

Algorithm 5 MRAssembleReducer

```
    procedure MRASSEMBLEREDUCER
        M ← function(M_1, M_2, ···, M_n)
            ▷ each MRCombiner() emits (M_1, M_2, ···, M_n)
        emit(cell, M)
5: end procedure

    procedure MRDATACUBE
        R ← MRSpreadMapper() ∪ MRCombiner() ∪
    MRSpreadReducer()
        R′ ← MRAssembleMapper() ∪ MRCombiner() ∪
    MRAssembleReducer()
        return R ∪ R′
10: end procedure
```

1) Data Reduction: Raw data usually have many overlapping values. Removing them alone reduces the data size considerably. Data reduction is directly related to the cost reduction for data cube calculation, and thus that of the computing resources. In other words, a data reduction is a crucial task in an optimized data cube calculation method.

We use a method capable of performing a data reduction easily using the characteristics of MR. MR includes a merging phase in which identical keys are merged. We merge identical cells by allocating cell and measurement values to the key and $value$, respectively, in the MR in the form of $\langle key; value\rangle$. If raw data $R = \{(a_1, b_1, c_1), (a_1, b_1, c_1)\}$, then $\langle a_1, b_1, c_1; 1\rangle$ and $\langle a_1, b_1, c_1; 1\rangle$ merge together and are finally emitted as $\langle a_1, b_1, c_1; 2\rangle$.

Additionally, we use the combine function (combiner) to perform a partial data reduction prior to the MR shuffle phase. Once the cell values expected to be emitted during the MRSpread map phase are accumulated in the memory buffer, those cell values are aggregated by the combiners. In other words, the task to be performed by the reducers is partially conducted in advance during the map phase to the extent of the data buffered in memory during this phase.

a) Example 1: Let us assume that we use two map functions, μ_1 and μ_2, to compute the raw data $R = \{(a_1, b_1, c_1), (a_1b_1, c_1), (a_1, b_1, c_1), (a_1, b_1, c_2)\}$. Using the input data $r_1 = \{(a_1, b_1, c_1),$
$(a_1, b_1, c_1)\}, r_2 = \{(a_1, b_1, c_1), (a_1, b_1, c_2)\}$ resulting from splitting R in half, r_1 and r_2 are then used as inputs pertaining to μ_1 and μ_2, respectively. Next, $\langle a_1, b_1, c_1; 2\rangle$ is emitted from μ_1, and $\langle a_1, b_1, c_1; 1\rangle$ and $\langle a_1, b_1, c_2; 1\rangle$ are emitted from μ_2. In other words, although r_1 and r_2 have identical cells $\langle a_1, b_1, c_1\rangle$ when split in half, they are separately combined for each map function.

2) Top Cuboid to Partial Cuboids: The MRSpread generates a top cuboid by aggregating all overlapping cells of raw data, and emits $2^D - 1$ partial cuboids from the top cuboid. As MRSpread emits partial cuboids using a top cuboid, in which the input data are aggregated and reduced instead of the raw data, the compotation load is considerably reduced.

a) Example 2: If we let the MRSpread data inputted into the reduce phase be $\langle a_1, b_1, c_1; 3\rangle$, then $P_a b = \{a_1, b_1, 3\}$, $P_a c = \{a_1, c_1, 3\}$, $P_b c = \{b_1, c_1, 3\}$, $P_a = \{a_1, 3\}$, $P_b = \{b_1, 3\}$, and $P_c = \{c_1, 3\}$ are emitted as partial cuboids. In other words, the emitted partial cuboids have the same measurement values, but different partial cells.

3) MapReduce Phase of MRSpread: This section describes MRSpread Mapper, MRCombiner, and MRSpread Reducer, which are defined to explain the MR operations of MRSpread.

An MRSpread Mapper receives the records of raw data R as input data. A mapper emits $\langle cell; measure\rangle$ multi-set pairs s amounting to the number of records of R, T. Here, a cell is tantamount to a record, and a measurement is used for aggregation. Multiple mappers are generated according to the size of R and the extent of the split. The first mapper is defined as $\mu_1 = \{s_1, s_2, ??s_t\}$, where t is the number of records split from R in correspondence with the number of mappers.

An MRCombiner retains the outputs emitted from the MRSpread Mapper until the memory buffer is filled up or the mappers finish their operation, and aggregates them and emits the outputs. Combiners operate on each of $\mu_1, \mu_2, \cdots, \mu_n$, where n is the number of mappers. A combiner aggregates every i$^{\text{th}}$ μ_i by cell, and emits them in the form of $\langle cell; F(m)\rangle$, where F is the aggregate function to be analyzed as queried by the user.

An MRSpread reducer receives input data in the form of $\langle cell; m_1, m_2, \cdots, m_n\rangle$ that have passed through the shuffle phase and have identical cells. Reducers compute unique cells using the aggregate function, F, allocated by the user. The outputs of the computation are emitted in the form of $\langle cell; F(m_1, m_2, \cdots, m_n)\rangle$. This can be considered the process of cell computation for the final top cuboid. Upon completion of a cell computation, partial cells constituting each partial cuboid are emitted.

An MRSpread mapper operates in parallel, and whose number of operations amounts to μ_n. An MRSpread combiner aggregates the adjacent data in advance, thus providing an effective data reduction. In addition, an MRSpread reducer operates in parallel in correspondence with the number of reducers allocated.

4) Data Flow of MRSpread: Fig. 4 shows an example of MRSpread. First, raw data are inputted into the map function; in the example case, the count is used as the aggregate function, and a measurement value of 1 is emitted. During the combine phase, the same two cells, $\{a_1, b_2\}$, are combined into $\langle a_1\ b_2; 2\rangle$, and the two cells, $\{a_2\ b_1\}$, are combined into $\langle a_2\ b_1; 2\rangle$, which are then emitted. During the shuffle phase, $\{a_1, b_1\}$ cells are merged and emitted in the form of $\langle a_1\ b_1; 1, 1\rangle$ for the reduce phase. During the reduce phase, $\langle a_1\ b_1; \{1, 1\}\rangle$ is aggregated into $\langle a_1\ b_1; 2\rangle$. In addition, the cells inputted into the reduce phase are used to generate partial cells pertaining to A and B cuboids and emitted as partial cells. When cell $\langle a_1\ b_1; 2\rangle$ is inputted, partial cells $\langle a_1\ *; 2\rangle$ and $\langle *\ b_1; 2\rangle$ are emitted, and cell $\langle a_1\ b_2; 2\rangle$ is emitted as partial cells $\langle a_2\ *; 2\rangle$ and $\langle *\ b_1; 2\rangle$.

D. MRAssemble: Assemble the Cell

MRAssemble collects partial cells and aggregates them to generate a complete data cube. It also generates complete cells. As with MRSpread, a large amount of data reduction is implemented through the combine phase in MRAssemble. The combiners used in MRAssemble are the same as those used in MRCombiner, as described in definition 4. Final cuboids are obtained after the process of partitioning each cell.

1) Partitioning by Cuboid: MRAssemble outputs complete cells as a result of the computation. To do so, it is necessary to partition each cell according to the given cuboid. The number of partitions is $2^D - 2$, and each partition is named after its cuboid.

a) Example 3: In the example, it is assumed that partitioning is performed on the data, $R_{\text{shuffle}} = \{\langle a_1\ *; 4\rangle, \langle *\ b_1; 4\rangle\}$, which underwent the shuffle phase. In $(a_1, *)$, which is the cell value of $\langle a_1\ *; 4\rangle$, the second attribute is defined by the '*' symbol, indicating its empty value. This implies that only the first attribute exists, and is the cell for cuboid A. Likewise, in $(*, b_1)$, which is the cell value of $\langle *\ b_1; 4\rangle$, the first attribute has an empty value, and by the same token, the second attribute b_1 is the cell for cuboid B.

2) MapReduce Phase of MRAssemble: This section describes an MRAssemble mapper and MRAssemble reducer as defined above to explain the MR operational steps in MRAssemble.

An MRAssemble mapper receives input data in the form of a $\langle cell; measure\rangle$ pair, where a $cell$ means a partial cell as the output of MRSpread and has a string value. A $measure$ denotes the value used for aggregation and has an integer value. The task of an MRAssemble mapper is simply emitting the input splits after dividing the input data in as many splits as the number of mappers. A partial aggregation of the i$^{\text{th}}$ mapper, μ_i, is performed using MRCombiner.

The task of an MRAssemble partitioner is emitting partial cells to the reduce phase by grouping them into partial cuboids. Because partitioners distinguish data using a partial cuboid, an equal number of reducers as partial cuboids are generated, and are operated in parallel. Partitioners can easily distinguish the cuboid type through the '*' symbol contained in a cell.

An MRAssemble reducer aggregates the data with the same cell and emits the outputs. It computes using the aggregate function F allocated by the user, and emits the fully computed cells to the cuboid pertaining to the partition. The job completion of all reducers occurs when the computations of all outputs to the cuboids are completed.

3) Data Flow of MRAssemble: Fig. 5 illustrates an example of MRAssemble. The figure shows that MRAssemble uses partial cuboids, the outputs of MRSpread, as inputs, and the same cells $\langle a_1\ *; 2\rangle$ and $\langle a_1\ *; 2\rangle$ are combined into $\langle a_1\ *; 4\rangle$ by the combiners before passing from the map function to the shuffle phase. During the shuffle phase, the two identical cells in $\langle *\ b_1; 2\rangle$ are merged into $\langle *\ b_1; \{2, 2\}\rangle$ and thus emitted in a reduced state. During the reduce phase, $\langle *\ b_1; \{2, 2\}\rangle$ is aggregated into $\langle *\ b_1; 4\rangle$, and the outputs of cuboids A and B are finally emitted. The AB cuboid generated in MRSpread,

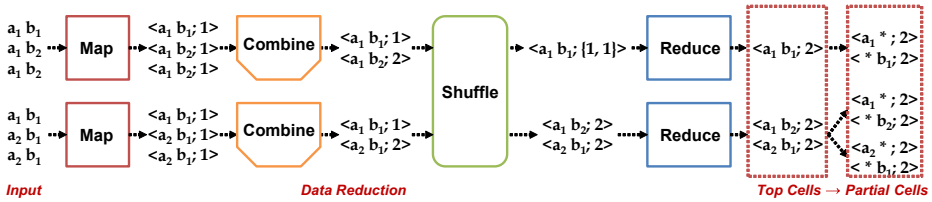

Fig. 4. A data flow example of MRSpread

and the cuboids A and B generated in MRAssemble, are computed to generate a full data cube.

4) Multiple MRAssemble: Even an MR designed to process large-scale data can encounter difficulties coping with excessively large data amounts or multidimensional data. In principle, it is possible to process large-scale data through additional nodes using scalability, which is a fundamental advantage of MR. However, this incurs additional costs. Large incomputable datasets can also be made computable by extending MRAssemble, although this is a time-consuming process.

If an MR cannot host all partial cells of $2^D - 2$ cuboids emitted in MRSpread, MRAssemble can be performed in iterations. In this case, MRAssemble processes $2^D - 2$ cuboids during the iterated MR phase to ensure the computation of all partial cuboids. The basic MRDataCube is run during the two PR phases of MRSpread and MRAssemble. An extended algorithm passes through MRSpread + (MRAssemble $\times$ P) phases, where $P = \frac{2^D-1}{\chi}$, and χ is the number of partial cuboids computable in one MP phase.

IV. Experimental Evaluations

A total of 21 PCs were used for the experiments conducted in this study. One was used for the NameNode of Hadoop, and the rest were utilized for the DataNode. The specifications of NameNode are a 3.0 GHz dual core CPU, 1 GB of RAM, and a 400 GB HDD, and those of DataNode were a 3.0 GHz dual core CPU, 512 MB of RAM, and a 200 GB HDD. All 21 PCs were interlinked using a 100 Mbps Ethernet. We used Ubuntu Server 11.04 as the OS, Linux kernel version 2.6.38, and Hadoop version 0.20.2. All algorithms were generated using Java version 1.6.

A. Experimental setup

1) Datasets: In this study, synthetic datasets were used for comparative experiments according to the characteristics of large-scale multidimensional data. These datasets are similar to real-world datasets because they are produced using Zipf? law[**?**]. We synthesized the datasets with a maximum magnitude of 1,000,000,000 tuples and six dimensions. A Zipf distribution satisfies Formula 1, where the values vary according to discrete variables $(n = i)$ and the frequency of each variable f varies according to the value of α. In other words, distribution bias increases in proportion to the increase in α. A Zipf distribution enables the modeling of real-world data such as the frequency of dictionary entries of words and population estimates. In the present study, under the assumption of generating data in accordance with a common real-world data distribution, we assigned 1 as the value of α, which is the expected value of a Zipf distribution.

$$f_i \propto \frac{1}{i^\alpha}(i = 1, \cdots, N) \tag{1}$$

B. Existing Algorithm

We performed comparative experiments to prove the superiority of the proposed algorithm over several comparable algorithms [2], [3], [10]. Using MR2^D, MRNaive, MRGBLP, and MRPipeSort, we examined the cube computation procedure and characteristics of each algorithm, and explored the differences from the proposed algorithm. MRNaive is the most basic MR-based data cube computation algorithm which computes cuboids $(n = 2^D)$ in one operation and scans raw data only once for a cube computation. MR2^D algorithm performs MR phases in multiple operations, which computes each cuboid at a different MR phase thus employs 2^D MR phases. MRGBLP is a distributed parallel algorithm extended from the MR-based GBLP algorithm[1]. It computes a child cuboid within a cube lattice from its parent cuboid not from raw data. When having multiple parent cuboids, it uses the smallest parent cuboid to compute the child cuboid. MRPipeSort is an algorithm extended from the PipeSort algorithm[2] in a distributed parallel paradigm. It efficiently computes the cuboids within a pipeline together by a signle MR phase which share the same prefix as their cuboid names thus can be executed at the same time through a single scan of raw data. It employs as the same number of MR phases as the one of pipelines.

C. Experimental Results

We experimentally compared the MRDataCube algorithm proposed in the present study with the four different algorithms described above under the same conditions. We performed three types of experiments: comparing the running times while (i) increasing the number of tuples, (ii) increasing the number of dimensions, and (iii) decreasing the number of nodes.

1) Varying Number of Tuples: In this experiment, we varied the tuple-dependent data size. Fig. 6 shows the results of comparing the running time while varying the number of tuples from 10,000,000 to 100,000,000. The MR2^D algorithm showed the lowest performance and highest rate increase in the running time. While MRNaive demonstrated a high speed when processing small amounts of data, its speed slowed to

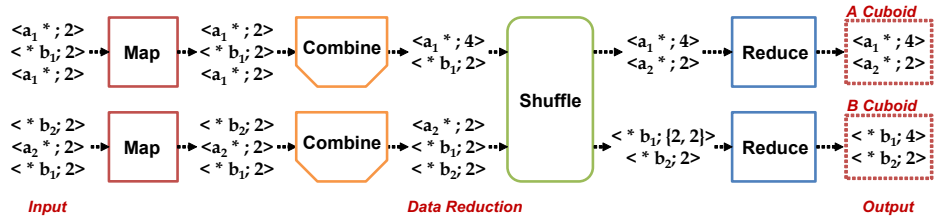

Fig. 5. A data flow example of MRAssemble

the level of MR2^D with an increase in the data amount. While MRGBLP and MRPipeSort showed low increasing rates, they were slower than MRNaive when processing small amounts of data. The proposed algorithm, MRDataCube, was faster than MRNaive even when processing small amounts of data, and outperformed all four algorithms with increasing numbers of tuples.

2) Varying Dimensions: In this experiment, the performance of the proposed algorithm was compared with that of the other four algorithms when increasing the data dimensions from three to seven. Fig. 7 shows the results of the comparative experiments, where the number of tuples was set to 10,000,000. MR2^D showed the lowest performance. MRNaive computed rapidly at lower dimensions, but was surpassed by MRGBLP for the seventh dimension. While MRPipeSort demonstrated a relatively superior performance, it was outperformed by MRDataCube.

3) Varying Number of Nodes: In this experiment, the running times of the five algorithms were tested as the number of nodes was increased from four to twenty. The number of tuples and dimensions were set to 50,000,000 and five, respectively. Fig. 8 shows that all algorithms operate increasingly faster as the number of nodes increases, with MRGBLP showing the smallest rate of change depending on the number of nodes. Even MR2^D and MRNaive, which require a lengthy running time by nature, demonstrated an increased computation speed with an increase in the number of nodes. The MRPipeSort and MRDataCube algorithms were also found to compute more rapidly as the number of nodes increased.

V. Conclusions

In this paper, we proposed MRDataCube, a new data cube computation algorithm based on the MR mechanism. Unlike existing MR-based data cube algorithms, the particularity of MRDataCube is a two-phase computation: the MRSpread phase, in which partial cuboid cells are emitted, and MRAssemble, which generates cuboids by computing all partial cells. MRDataCube takes maximum advantage of the MR combine function using the concepts of partially computed partial cells and partial cuboids composed of partial cells. Its additional advantage is a progressive reduction in the data size for each PR operational step.

MRDataCube is a distributed parallel data cube algorithm that achieves maximum benefit of the basic mechanism of the MR framework by optimally using its advantages. We performed experiments for a more detailed and accurate comparison of related algorithms by extending the representative data cube algorithms to incorporate the MR paradigm. The results of our various experiments verified the superiority of MRDataCube over all existing data cube algorithms. That is, MRDataCube showed a much higher cost-effectiveness and computation speed compared to other algorithms under the same computer environments.

Data cube is being used for a multidimensional analysis in many real-life application domains. Along with this trend, MapReduce is adopted by increasing the number of users for large-scale data processing. We expect that for an analysis of large-scale multidimensional data cubes, MRDataCube will soon prove its usability and efficiency in many real-life application fields. In fact, the proposed MRDataCube can be extended with respect to its concepts and ideas. In future studies, we will develop useful algorithms extended from MRDataCube and test them in various fields of application.

Acknowledgments

This work was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (2011-0011824).

References

[1] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals", *Data Mining and Knowledge Discovery*, vol.1, no.1, pp.29-53, 1997.

[2] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, S. Sarawagi, "On the computation of multidimensional aggregates," *Proc. of VLDB Conference*, 1996, pp.506-521.

[3] V. Harinarayan, A. Rajaraman, J. D. Ullman, "Implementing data cubes efficiently," *ACM SIGMOD Record*, vol. 25, no. 2, pp.205-216, 1996.

[4] Y. Chen, F. Dehne, T. Eavis, A. Rau-Chaplin, "Parallel rolap data cube construction on shared-nothing multiprocessors," *Proc. of Parallel and Distributed Processing Symposium*, 2003, pp.10-19.

[5] F. Dehne, T. Eavis, A. Rau-Chaplin, "The cgmcube project: Optimizing parallel data cube generation for rolap," *Distributed and Parallel Databases*, vol. 19, no. 1, pp.29-62, 2006.

[6] K. Sergey, K. Yury, "Applying map-reduce paradigm for parallel closed cube computation," *Proc. of International Conference on Advances in Databases, Knowledge, and Data Applications*, 2009, pp. 62-67.

[7] J. Dean, S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, 2008.

[8] T. White, *Hadoop: the definitive guide*, OReilly, 2012.

[9] Y. Wang, A. Song, J. Luo, "A mapreducemerge-based data cube construction method," *Proc. of International Conference on Grid and Cooperative Computing*, 2010, pp. 1-6.

[10] A. Nandi, C. Yu, P. Bohannon, R. Ramakrishnan, "Distributed cube materialization on holistic measures," *Proc. of IEEE International Conference on Data Engineering*, 2011, pp. 183-194.

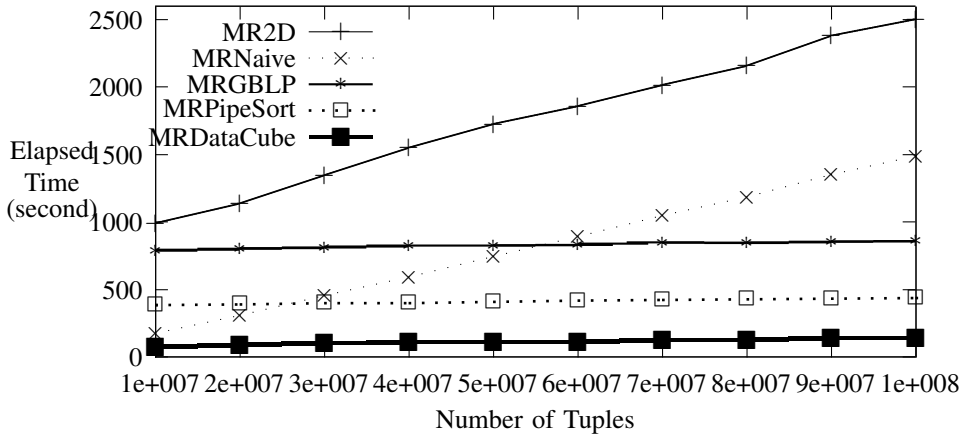

Fig. 6. Elapsed time after varying the number of tuples (10,000,000 to 100,000,000)

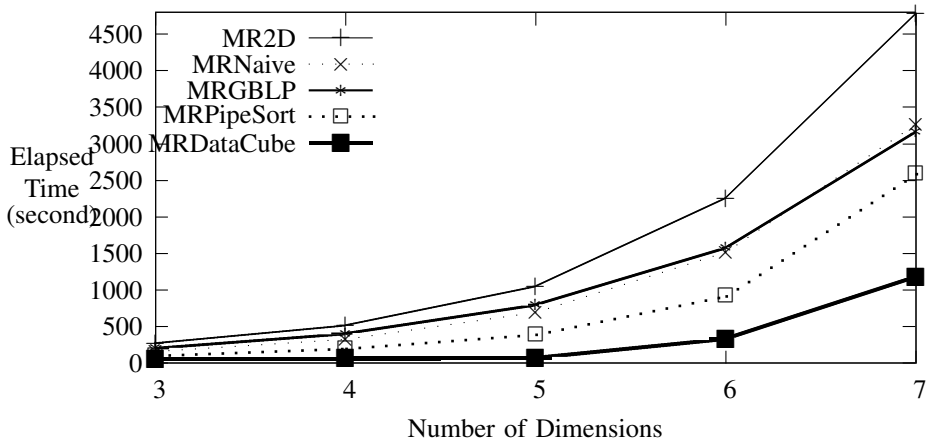

Fig. 7. Elapsed time by varying the number of dimensions (three to seven) with 10,000,000 tuples

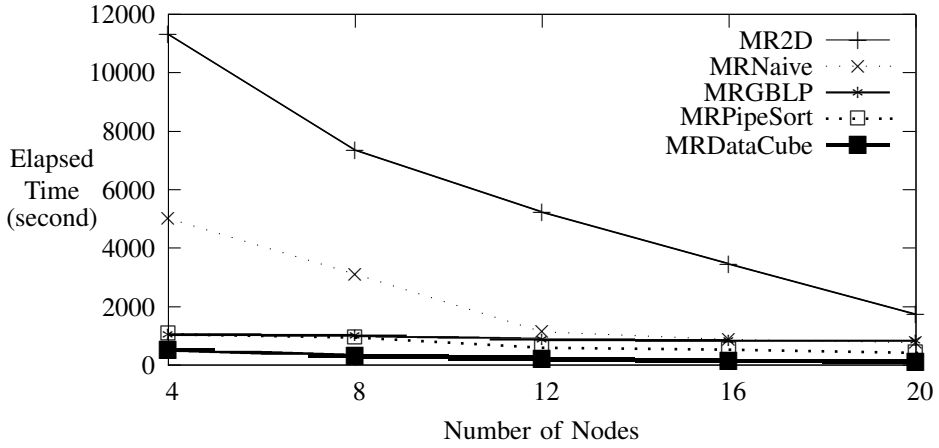

Fig. 8. Elapsed time by varying the number of nodes (four to twenty)