HaCube: Extending MapReduce for Efficient OLAP Cube Materialization and View Maintenance

Zhengkui Wang[1(✉)], Yan Chu[2(✉)], Kian-Lee Tan[3], Divyakant Agrawal[4], and Amr EI Abbadi[4]

[1] Singapore Institute of Technology, Singapore, Singapore
zhengkui.wang@singaporetech.edu.sg
[2] Harbin Engineering University, Harbin, China
chuyan@hrbeu.edu.cn
[3] National University of Singapore, Singapore, Singapore
tankl@comp.nus.edu.sg
[4] University of California, Santa Barbara, USA
{agrawal,amr}@cs.ucsb.edu

Abstract. Data cubes are widely used as a powerful tool to provide multi-dimensional views in data warehousing and On-Line Analytical Processing (OLAP). However, with increasing data sizes, it is becoming computationally expensive to perform data cube analysis. In this paper, we introduce `HaCube`, an extension of MapReduce, designed for efficient parallel data cube computation on large-scale data. We also provide a general data cube materialization solution which is able to facilitate the features in MapReduce-like systems towards an efficient data cube computation. Furthermore, we demonstrate how `HaCube` supports view maintenance through either incremental computation (e.g. used for SUM or COUNT) or recomputation (e.g. used for MEDIAN or CORRELATION). We implement `HaCube` by extending Hadoop and evaluate it based on the TPC-D benchmark over billions of tuples on a cluster with over 320 cores. The experimental results demonstrate the efficiency, scalability and practicality of `HaCube` for cube computation over a large amount of data in a distributed environment.

1 Introduction

In many industries, such as sales, manufacturing and finance, there is a need to make decisions based on aggregation of data over multiple dimensions. `Data cubes` [9] are one such critical technology that has been widely used in data warehousing and On-Line Analytical Processing (OLAP) for data analysis in support of decision making.

In OLAP, the attributes are classified into **dimensions** (the grouping attributes) and **measures** (the attributes which are aggregated) [9]. Given n dimensions, there are a total of 2^n `cuboids`, each of which captures the aggregated data over one combination of dimensions. To speed up query processing,

S.B. Navathe et al. (Eds.): DASFAA 2016, Part II, LNCS 9643, pp. 113–129, 2016.
DOI: 10.1007/978-3-319-32049-6_8

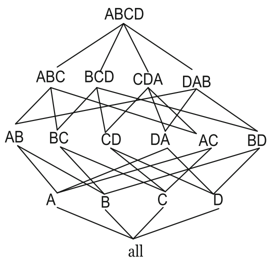


Fig. 1. A cube lattice with 4 dimensions: A, B, C and D

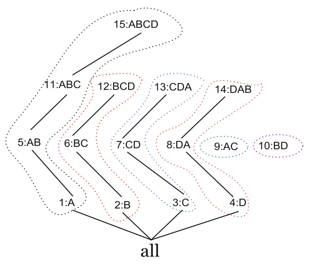


Fig. 2. The numbered cube lattice with execution batches

these cuboids are typically stored into a database as views. The problem of **data cube materialization** is to efficiently compute all the views ($\mathbb{V}$) based on the data ($\mathbb{D}$). Figure 1 shows all the cuboids represented as a cube lattice with 4 dimensions A, B, C and D.

In many append-only applications (no UPDATE and DELETE operations), the new data ($\Delta\mathbb{D}$) will be incrementally INSERTed or APPENDed to the data warehouse for view update. For instance, the logs in many applications (like the social media or stocks) are incrementally generated/updated. There is a need to update the views in a manner of one-batch-per-hour/day. The problem of **view maintenance** is to efficiently calculate the latest views while $\Delta\mathbb{D}$ are produced.

Both data cube materialization and view maintenance are computationally expensive, and have received considerable attention in the literature [12,16,20,23]. However, existing techniques can no longer meet the demands of today's workloads. On the one hand, the amount of data is increasing at a rate that existing techniques (developed for a single server or a small number of machines) are unable to offer acceptable performance. On the other hand, more complex aggregate functions (like complex statistical operations) are required to support complex data mining and statistical analysis tasks. Thus, this calls for new scalable systems to efficiently support data cube analysis over a large amount of data.

Meanwhile, MapReduce (MR) [7] has emerged as a powerful computation paradigm for parallel data processing on large-scale clusters. Its high scalability and append-only features have made it a potential target platform for data cube analysis in append-only applications. Therefore, exploiting MR for data cube computation has become an interesting research topic. However, deploying an efficient data cube computation using MR is non-trivial. A naive implementation of cube materialization and view maintenance over MR can result in high overheads.

Therefore, in this paper, we are motivated to explore the techniques of developing new scalable data cube analysis systems by leveraging the MR-like paradigm, as well as to develop new techniques for efficient data cube computation to broaden the application of data cubes primarily for append-only environments. Our main contributions are as follows:

1. *New system design and implementation:* We present HaCube, an extension of MR, for large-scale data cube computation. HaCube tries to integrate the good features from both MR and parallel DBMS. It extends MR to better support data cube computation by integrating new features, e.g. a new local store for data reuse among jobs, a layer with user-friendly interfaces and a new computation paradigm MMRR (MAP-MERGE-REDUCE-REFRESH). HaCube illustrates one way to develop a scalable and efficient decision making system, such that cube computation can be utilized in more applications.
2. *A General Cubing Algorithm:* We provide a general and efficient data cubing algorithm, CubeGen, which is able to complete the entire cube lattice using one MR job. We show how cuboids can be batched together to minimize the read/shuffle overhead and salvage partial work done. On the basis of batch processing principle, CubeGen further leverages the ordering property of the reducer input provided by the MR-like framework for an efficient materialization.
3. *Efficient View Maintenance Mechanisms:* We demonstrate how views can be efficiently updated under HaCube through either recomputation (e.g. used for MEDIAN or CORRELATION) or incremental computation (e.g. used for SUM or COUNT).
4. *Experimental Study:* We evaluate HaCube based on the TPC-D benchmark with more than two billions tuples. The experimental results show that HaCube has significant performance improvement over MR.

The rest of the paper is organized as follows. In Sect. 2, we provide an overview of HaCube. Sections 3 and 4 present our proposed cube materialization and view maintenance approaches. We report our experimental results in Sect. 5. In Sects. 6 and 7, we review some related works and conclude the paper.

2 HaCube: The Big Picture

2.1 Architecture

Figure 3 gives an overview of the basic architecture of HaCube. We implement HaCube by modifying Hadoop which is an open source equivalent implementation of MR [1]. Similar to MR, all the nodes in the cluster are divided into two different types of function nodes, including the master and processing nodes. The master node is the controller of the whole system and the processing nodes are used for storage and computation.

Master Node: The master node consists of two functional layers:

1. The **cube converting layer** contains two main components: Cube Analyzer and Cube Planner. The cube analyzer is designed to accept the user request of data cube analysis, analyze the cube, such as figuring out the cube id (the identifier of the cube analysis application), analysis model (materialization or view update), measure operators (aggregation function), and input and output paths etc.

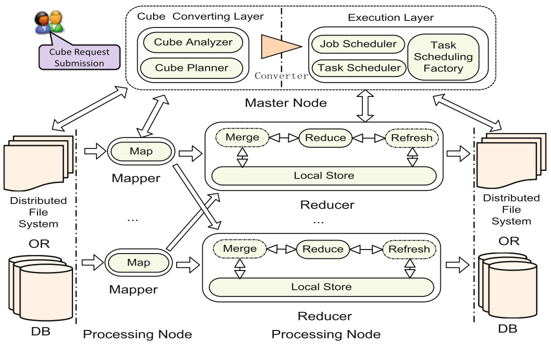


Fig. 3. HaCube architecture

The cube planner is developed to convert the cube analysis request into an execution job (either a materialization job or a view update job). The execution job is divided into multiple tasks each of which handles part of the cuboid calculation. The cube planner consists of several functional components such as the execution plan generator (combine the cuboids into batches to reduce the overhead), and load balancer (assign the right number of computation resources for each batch).

2. The **execution layer** is responsible for managing the execution of jobs passed from the cube converting layer. It has three main components: `job scheduler`, `task scheduler` and `task scheduling factory`. We use the same job scheduler as in Hadoop which is used to schedule different jobs from different users. In addition, we add a task scheduling factory which is used to record the task scheduling information of a job which can be reused in other jobs. Furthermore, we develop a new task scheduler to schedule the tasks in terms of the scheduling history stored in the task scheduling factory rather than the random scheduler used in MR.

Processing Node: A processing node is responsible for the task execution assigned from the master node. Similar to MR, each processing node contains one or more processing units each of which can either be a `mapper` or a `reducer`. Each processing node has a `TaskTracker` which is in charge of communicating with the master node through heartbeats, reporting its status, receiving the task, reporting the task execution progress and so on. Unlike MR, there is a `Local Store` built at each processing node running `reducers`. The local store is developed to cache useful data of a job in the local file system of the reducer node. It is a persistent storage in the local file system and will not be deleted after a job execution. In this way, tasks (possibly from other jobs) assigned to the same reducer node can access the local store directly from the local file system.

2.2 Computation Paradigm

`HaCube` inherits some features from MR, such as data read/process/write format of (key, value) pairs, sorting all the intermediate data and so on. However, it further enhances MR to support a new computational paradigm. `HaCube` adds two optional phases - a `Merge` phase and a `Refresh` phase before and after the `Reduce` phase - to support the MAP-MERGE-REDUCE-REFRESH (MMRR) paradigm as shown in Fig. 3.

The `Merge` phase has two functionalities. First, it is used to cache the data from the reduce input to the local store. Second, it is developed to sort and merge the partitions from mappers with the cached data in the local store. The `Refresh` phase is developed to perform further computations based on the reduce output data. Its functionalities include caching the reduce output data to the local store and refreshing the reduce output data with the cached data in the local store. These two additional phases are intended to fit different application requirements for efficient execution support.

As mentioned, these two phases are optional for the jobs. Users can choose to use the original MR computation or MMRR computation. More details can be found in Sect. 4 about how MMRR benefits the data cube view maintenance.

3 Cube Materialization

In this section, we provide our proposed data cubing algorithm, `CubeGen`, under the MR-like systems. We first present some principles of sharing computation through cuboid batching and a batch generator, then followed by the detail implementation. For simplicity, we assume that we are materializing the complete cube. Note that our techniques can be easily generalized to compute a partial cube (compute only selective cuboids). We also omit the cuboid "all" from the lattice. This special cuboid can be easily handled through an independent processing unit.

3.1 Cuboid Computation Sharing

To build the cube, computing each cuboid independently is clearly inefficient. A more efficient solution, which we advocate, is to combine cuboids into batches so that intermediate data and computation can be shared and salvaged.

We have the following observation: Let A and B be a set of dimensions such that $A \bigcap B = \emptyset$. In MR-like systems, given cuboids A and AB, A can be combined and processed together with AB, once AB is set of the key and is partitioned by A in one MR job. A is referred to as the **ancestor** of AB (denoted as $A \prec AB$). Meanwhile, AB is called the **descendant** of A. Note that the ancestor and descendant require them share the same prefix. This observation is the formal basis for combining and batching the cuboids computation under the MR-like systems.

The above observation can be generalized using transitivity: Since we can combine the processing of the pair of cuboids $\{A, AB\}$ and the pair $\{AB, ABC\}$, we can also combine the processing of the three cuboids $\{A, AB, ABC\}$. Thus, given one cuboid, all its ancestors can be calculated together as a batch. For instance, in Fig. 1, as $A \prec AB \prec ABC \prec ABCD$, the cuboids A, AB, ABC

can be processed with $ABCD$. Note that BC cannot be processed with $ABCD$ because $BC \nprec ABCD$.

Given a batch, the principle to calculate this batch is to set the *sort dimensions* as the key and partition the (k,v) pairs based on the *partition dimensions* in the key in the MR-like paradigm. We formally define these two dimension classes below:

Definition 1 ***Sort Dimensions:*** *The dimensions in cuboid A are called the sort dimensions if A is the descendant of all other cuboids in one batch.*

Definition 2 ***Partition Dimensions:*** *The dimensions in cuboid A are called the partition dimensions if A is the ancestors of all other cuboids in one batch.*

For instance, given the batch $\{A, AB, ABC, ABCD\}$, $ABCD$ and A can be set as the sort and partition dimensions respectively.

The benefits of this approach are: (1) In the reduce phase, the group-by dimensions are all in sorted order for every cuboid in the batch, since MR would sort the data before supplying to the reduce function. This is an efficient way of cube computation since it obtains sorting for free and no other extra sorting is needed before aggregation. (2) All the ancestors do not need to shuffle their own intermediate data but use their descendant's. This would significantly reduce the intermediate data size, and thus remove a lot of data sort/partition/shuffle overheads.

3.2 Plan Generator

A plan generator is developed to generate the batches among the given cuboids. Intuitively, the more cuboids can be combined, the more sharing operations can be achieved. Therefore, the plan generator is responsible for generating the minimum number of batches based on the aforementioned principles. Note that each cuboid may have different permutations. For instance, the cuboid $ABCD$ can also be permutated as $ABDC$, $ACBD$, $BCDA$, $CDAB$, $DABC$ and so on. Thus, as the number of dimensions increases, it is no longer applicable to enumerate all the possible plans exhaustively. As such, some heuristic algorithm can be used to find a suboptimal execution plan.

Recall that one cuboid can be batched with all its ancestors. In this paper, we adopt a greedy algorithm to combine one cuboid with as many of its ancestors as possible. Intuitively, each batch construction starts from one unbatched cuboid with the maximum number of dimensions. The chosen cuboid then searches its different permutations with all its unbatched ancestors and the one with the largest number of ancestors is used to form this batch. The construction continues until all the cuboids are batched. In addition, we propose different optimizations to further reduce the search space, such as how to choose the right permutation and how to stop the permutation evaluation earlier. More details and proof can be found in our technical report [18]. For instance, given $2^n - 1$ cuboids (excluding "*all*") in Fig. 1, the algorithm generates 6 batches marked using the dotted lines as shown in Fig. 2.

Algorithm 1. **CubeGen Algorithm**

```
   Function: Map(t)
1  # t is the tuple value from the raw data
2  Let $\mathbb{B}$ (resp. $I_i$) be the batch set with $B_0$, $B_1$, ..., $B_{b-1}$ (resp. the identifier of batch $B_i$)
3  for each $B_i$ in $\mathbb{B}$ do
4      k (resp. v) $\Leftarrow$ get sort dimensions (resp. the measure m) in $B_i$ from t
5      # If there are multiple measures (e.g. $m_1, m_2$), then v $\Leftarrow$ $(m_1, m_2)$
6      v.append($I_i$); emit(k,v);
   Function: Partitioning($k, v$)
7  Let $R_i$ (resp. $attr$) be the number of reducers (resp. the partition dimensions) for $B_i$
8  $S_i \Leftarrow \sum_{j=0}^{i-1} R_j$
9  return $S_i + hash(attr, R_i)$;
   Function: Reduce/Combine ($k, \{v_1, v_2, ..., v_m\}$)
10 Let $\mathbb{C}$ (resp. $\mathbb{M}$) be the cuboid set in the batch identifier (resp. the aggregate function)
11 for $C_i$ in $\mathbb{C}$ do
12     if $C_i$ is ready then
13         $k''$ (resp. $v''$)$\Leftarrow$ get the group-by dimensions in $C_i$ (resp. $\mathbb{M}(v_1, ..., v_m, v_1^{'}, ..., v_k^{'}, ...)$)
14         # Perform multiple aggregate functions e.g. $(\mathbb{M}_1, \mathbb{M}_2)$ here: $v_1'' \Leftarrow \mathbb{M}_1(v_1, ..., v_m, v_1^{'}, ..., v_k^{'}, ...)$ and $v_2'' \Leftarrow \mathbb{M}_2(v_1, ..., v_m, v_1^{'}, ..., v_k^{'}, ...)$
15         emit($k'', v''$);
16     else
17         Buffer the measure for aggregation
```

3.3 Implementation of CubeGen

Consider the batch plan B with b batches (B_0, B_1, ..., B_{b-1}) generated from the plan generator. There is a need to determine the number of computation resources (reducers) assigned to each batch. To achieve this, we also propose one load balancing approach based on sampling to guarantee that the computation task in each reducer can be balanced. Due to space constraint, we omit the discussion; interested readers are referred to [18]. Suppose that the number of reducers needed for each batch is R=(R_0, R_1, ..., R_{b-1}). Given B and R, the proposed `CubeGen` algorithm materializes the entire cube in one job and its pseudo-code is provided in Algorithm 1.

Map Phase: The base data is split into different chunks each of which is processed by one mapper. `CubeGen` parses each tuple and emits multiple (k,v) pairs each of which is for one batch (lines 3–6). The sort dimensions in the batch are set as the key and the measure is set as the value.

To distinguish which (k,v) pair is for which batch with which cuboids, we add a batch identifier appended after the value. The identifier is developed as one Bitmap with 2^n bits where n is the number of dimensions and each bit corresponds to one cuboid. First, we number all the 2^n cuboids from 0 to $2^n - 1$. Second, if the cuboid is included in one batch, its corresponding bit is set as 1, otherwise 0. For instance, Fig. 2 depicts an example of a numbered cube lattice. Assume that B_0 consists of cuboids $\{A, AB, ABC$ and $ABCD\}$. The identifier for B_0 is set as '10001000 00100010'.

The partitioning function partitions the pairs to the appropriate partition based on the identifier and the load balancing plan R. CubeGen first schedules the data into the right range of reducers. Recall that the batch B_i is assigned R_i reducers. Therefore, the assigned reducers for batch B_i are from $\sum_{j=0}^{i-1} R_j$ to $\sum_{j=0}^{i-1} R_j + R_i$-1. Then the (k,v) pairs are hash partitioned among these R_i reducers according to the partition dimensions in the key (lines 7–9).

Reduce Phase: In the Reduce phase, the MR library sorts all the (k,v) pairs based on the key and passes them to the reduce function. Each reducer obtains its computation tasks (the cuboids in the batch) by parsing the batch identifier in the value. The reduce function extracts the measure and projects the group-by dimensions for each cuboid in the batch. For the descendant cuboid, the aggregation can be performed directly based on input tuple, since each input tuple is one complete group-by cell. For other cuboids, the measures of the group-by cell are buffered until the cell receives all the measures it needs for aggregation (lines 11–17). We develop multiple file emitters to write different aggregated results to different destinations.

Note that if the (k,v) pairs can be pre-aggregated in the map phase, users can specify a combine function to conduct a first round aggregation. The combine function is normally similar to the reduce function as shown in lines 10–17, but only aggregates the pairs with the same key. This pre-aggregation is able to reduce the data shuffle size between mappers to reducers.

We emphasize that if there are muliple measures (e.g. m_1, m_2, ..., m_n) and multiple aggregate functions ($\mathbb{M}_1$, $\mathbb{M}_2$, .., $\mathbb{M}_m$), they can be processed in the same MR job as shown in the line 5 and 14 in Algorithm 1. Compared to the naive solution, CubeGen minimizes the cube materialization overheads by sharing the data read/shuffle/computation to the maximum, which obtains significant performance improvement as we shall see in Sect. 5.

4 View Maintenance

There are two different manners to update the views, namely recomputation and incremental computation. Recomputation computes the latest views by reconstructing the cube based on the entire base data $\mathbb{D}$ and $\Delta\mathbb{D}$. In append-only applications, this manner is normally used for the holistic aggregate functions, e.g. STDDEV, MEDIAN, CORRELATION and REGRESSION [9].

Incremental computation, on the other hand, updates the views using only $\mathbb{V}$ and $\Delta\mathbb{D}$ in two steps: (1.) In the propagate step, a delta view $\Delta\mathbb{V}$ is calculated based on the $\Delta\mathbb{D}$. (2.) In the refresh step, the latest view is obtained by merging

Algorithm 2. A Refresh Job in MR

```
   Function: Map(t)
1  # t is the tuple value from either $\mathbb{V}$ or $\Delta\mathbb{V}$
2  k (resp. v) $\Leftarrow$ get dimensions (resp. aggregate value) from t;
3  emit(k,v)
   Function: Reduce(k, $\{v_1, v_2\}$)
4  emit($k, \mathbb{M}(v_1, v_2)$)
```

$\mathbb{V}$ and $\Delta\mathbb{V}$ without visiting $\mathbb{D}$ [14]. In append-only applications, this manner is normally used for the distributive and algebraic aggregate functions, e.g. SUM, COUNT, MIN, MAX and AVG [9]. Note that the update for these functions can also be conducted through recomputation.

4.1 Supporting View Maintenance in MR

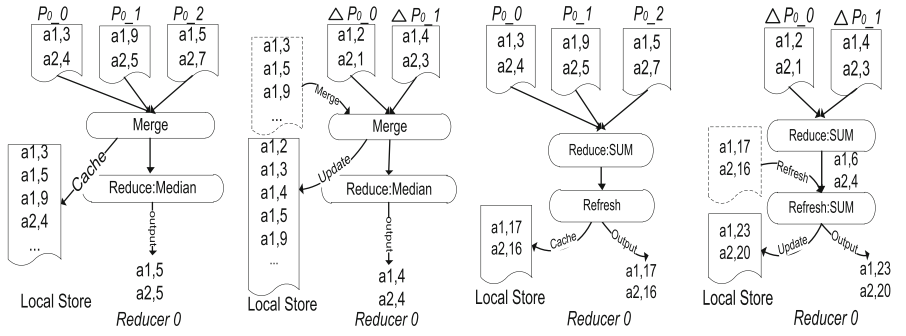
To support recomputation in MR, when $\Delta\mathbb{D}$ is inserted, the latest views can be calculated by issuing one MR job using our `CubeGen` algorithm to reconstruct the cube over $\mathbb{D} \cup \Delta\mathbb{D}$. The key problem with such an MR-based recomputation view updates is that reconstruction from scratch in MR is expensive. This is because the base data (which is large and increases in size at each update) has to be reloaded to the mappers from DFS and shuffled to the reducers for each view update, which incur significant overheads.

To support incremental computation in MR, the latest views can be calculated by issuing two MR jobs. The first propagate job generates $\Delta\mathbb{V}$ from $\Delta\mathbb{D}$ using our proposed `CubeGen` algorithm. The second refresh job merges $\mathbb{V}$ and $\Delta\mathbb{V}$ as shown in Algorithm 2. However, this would incur significant overheads. For instance, the materialized $\Delta\mathbb{V}$ from the propagate job has to be written back to DFS, reloaded from DFS again and shuffled from mappers to reducers in the refresh job. Likewise, $\mathbb{V}$ has to be reloaded and shuffled around in the refresh job. Therefore, it is highly expensive to support view update operations directly over the traditional MR.

4.2 HaCube Design Principles

`HaCube` avoids the aforementioned overheads through storing and reusing the data between different jobs. We extend MR to add a local store in the reducer node which is intended to store useful data of a job in the local file system. Thus, the task shuffled to the same reducer is able to reuse the data already stored there. In this way, the data can be read directly from the local store (and thus significantly reducing the overhead that would have been incurred to read the data from DFS and shuffle them from mappers).

We further extend MR to develop a new task scheduler to guarantee that the same task is assigned to the same reducer node and thus the cached data can be reused among different jobs. Specifically, the task scheduler records the



(a) MEDIAN: Cube Materialization (Map-Merge-Reduce) (b) MEDIAN: View maintenance (Map-Merge-Reduce) (c) SUM: Cube Materialization (Map-Reduce-Refresh) (d) SUM: View maintenance (Map-Reduce-Refresh)

Fig. 4. Recomputation for MEDIAN and incremental computation for SUM in HaCube

scheduling information by storing a mapping between the data partition number (corresponds to the task) and the TaskTracker (corresponds to the reducer node) and puts it to the task scheduling factory from one job. When a new job is triggered to use the scheduling history from previous jobs, the task scheduler fetches and adopts the scheduling information from the factory to distribute the tasks. The scheduler automatically checks the situation of the over-loaded nodes and re-assigns the task to a nearby processing node.

In addition, two computation phases (`Merge` and `Refresh`) are added to conduct more computation with the cached data locally. The `Merge` phase is added to either cache the intermediate reduce input data in one job or preprocess the data between the newly arriving data and cached data before the `Reduce` phase. The `Refresh` phase is added to either cache the reduce output data in one job or postprocess the reduce output result with the cached data after the `Reduce` phase.

4.3 Supporting View Maintenance in HaCube

Recomputation. The recomputation view update can be efficiently supported in `HaCube` using a Map-Merge-Reduce (MMR) paradigm. We demonstrate this procedure through one running example by introducing the cube materialization and update jobs.

In the first cube materialization job, `HaCube` is triggered to cache the intermediate reduce input data to the local store in the `Merge` phase, such that this data can be reused during the view update job. For instance, Fig. 4(a) shows an example of calculating the cuboid A for MEDIAN. Assume that reducer 0 is assigned to process cuboid A. In this job, each mapper emits one sorted partition for reducer 0, such as P_0_0, P_0_1 and P_0_2. Here, each partition is a sequence of (dimension-value, measure-value) pairs, e.g., (a1, 3), (a2, 4). Recall that once these partitions are shuffled to the reducer 0, it first performs a merge-sort (the same as MR does) to sort all the partitions based on the key in the `Merge` phase. The sorted data is further supplied to the reduce function to calculate the MEDIAN for each group-by cell (e.g. $< a1, 5 >$ and $< a2, 5 >$) where this view will be written to DFS.

Different to MR (which deletes all the intermediate data after one job), since recomputation requires the base data for update, `HaCube` caches the sorted reduce input data in the `Merge` phase for subsequent reuse. This caching operation is conducted while the `Reduce` phase finishes, which guarantees the atomicity of the operation - if the reduce task fails, the data will not be written to the local store. Meanwhile, the scheduling information is recorded.

A view update job is launched when $\Delta\mathbb{D}$ is added for view updates. Intuitively, this job conducts a cube materialization job using the `CubeGen` algorithm based on $\Delta\mathbb{D}$. It differs from the first materialization job in the scheduling and the `Merge` phase. For task scheduling, instead of randomly distributing the tasks to reducer nodes, it distributes the tasks according to the scheduling history from the first materialization job to guarantee that the same tasks are processed at the same reducer. For instance, the partitions of cuboid A (ΔP_0_0 and ΔP_0_1) are scheduled to the same node running reducer 0 as shown in Fig. 4(b). In the `Merge` phase, since the base data is already cached in the local store, `HaCube` merges the delta partitions with the cached base data from the local store. Recall that the cached data is the sorted reduce input data from the previous job, and so it has the same format as the delta partition. Thus, it can be treated as a local partition and a global merge-sort is further performed. Then the sorted data will be supplied to the reduce function for recalculation in the `Reduce` phase. When the `Reduce` phase finishes, the local store is updated with both the base and delta data (becoming an updated base dataset) for further view update use.

Compared to MR, `HaCube` does not need to reload the base data from DFS and shuffle them from mappers to reducers for recomputation. This significantly reduces the data read/shuffle overheads. Another implementation optimization is proposed to minimize the data caching overhead. To cache the data to the local store, it is expensive to push the data to the local store, as this would incur much overhead of moving a large amount of data. Based on the observation that the intermediate sorted data are maintained in temporary files in the local disk in each reducer, `HaCube` simply registers the file locations to the local store rather than moving them. Note that the traditional MR would delete these temporary files once one job finishes. As we shall see, the experimental study shows that there is almost no overhead added for caching the data with this optimization.

Incremental Computation. `HaCube` adopts a `Map-Reduce-Refresh` (MRR) para- digm for incremental computation. Intuitively, different to MR in the first materialization job, it triggers to invoke a `Refresh` phase after the `Reduce` phase, to cache the view $\mathbb{V}$ to the local store for further reuse. For instance, Fig. 4(c) shows an example of calculating cuboid A for SUM in reducer 0. In this job, $\mathbb{V}$ ($< a1, 17 >$ and $< a2, 16 >$) is cached to the local store in the `Refresh` phase, and the scheduling information is also recorded.

When $\varDelta\mathbb{D}$ is added for view updates, HaCube conducts both the propagate and refresh steps in one view update job, as $\mathbb{V}$ is already cached in the reducer node. This view update job in HaCube also executes in an MRR paradigm where MR (Map-Reduce) phases obtain $\varDelta\mathbb{V}$ based on $\varDelta\mathbb{D}$ (propagate step) and the Refresh phase merges $\varDelta\mathbb{V}$ with $\mathbb{V}$ locally (refresh step). Intuitively, this can be achieved by running the CubeGen algorithm on $\varDelta\mathbb{D}$ using the same scheduling plan as the previous materialization job. Meanwhile, the cached views in the local store will be updated with the latest ones. For instance, in Fig. 4(d), the Reduce phase calculates the $\varDelta\mathbb{V}$ ($< a1, 6 >$ and $< a2, 4 >$) based on $\varDelta\mathbb{D}$. In the Refresh phase, the updated view ($< a1, 23 >$ and $< a2, 20 >$) is obtained by merging $\varDelta\mathbb{V}$ with $\mathbb{V}$ ($< a1, 17 >$ and $< a2, 16 >$) cached in the local store.

Different to MR, HaCube is able to finish the incremental computation in one job where there is no need to reload and shuffle the delta views and old views among DFS and the cluster during the propagate and refresh steps. This provides an efficient view update using the incremental computation by removing much overheads.

5 Performance Evaluation

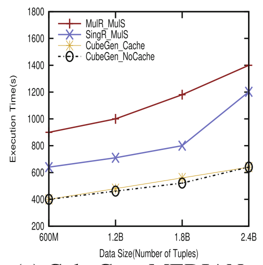
We evaluate HaCube on the Longhorn Hadoop cluster in TACC (Texas Advanced Computing Center) [2]. Each node consists of 2 Intel Nehalem quad-core processors (8 cores) and 48 GB memory. By default, the number of nodes used is 35 (and 280 cores).

We perform our studies on the classical dataset generated by TPC-D benchmark generators [3]. The TPC-D benchmark offers a rich environment representative of many decision support systems. We study the cube views on the fact table, *lineitem* in the benchmark. The attributes *l_partkey*, *l_orderkey*, *l_suppkey* and *l_shipdate* are used as the dimensions and the *l_quantity* as the measure. We choose MEDIAN and SUM as the representative functions for evaluation.

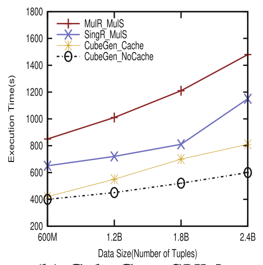
5.1 Cube Materialization Evaluation

Baseline Algorithms: To study the benefit of the optimizations adopted in CubeGen, we design two corresponding baseline algorithms to study each of them including MulR_MulS (compute each cuboid using one MR job) and SingR_MulS (compute all the cuboids using one MR job without batching them), which are widely used for cube computations in MR. MulR_MulS (Resp. SingR_MulS) is used to study the benefit of removing multiple data read overheads (Resp. sharing the shuffle and computation through batch processing).

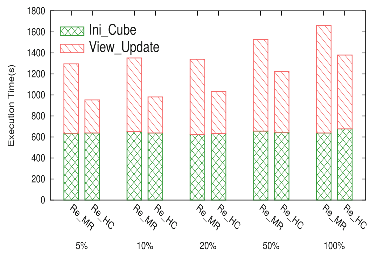
In the following set of experiments, we vary the data size from 600M (Million) to 2.4B (Billion) tuples.We study two versions of the CubeGen algorithm where CubeGen _Cache caches the data and CubeGen_NoCache does not. This provides insights into the overhead of caching the data to the local store.



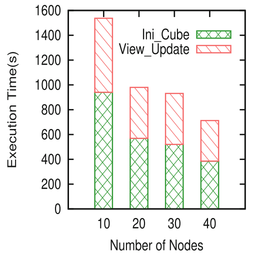
(a) CubeGen: MEDIAN



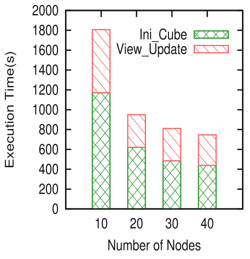
(b) CubeGen: SUM



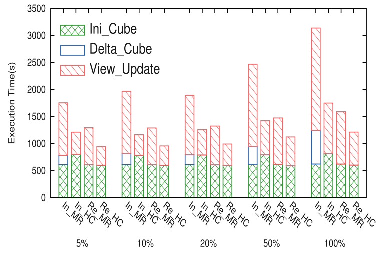
(c) Maintenance: MEDIAN



(d) Parallelism: MEDIAN



(e) Parallelism: SUM



(f) Maintenance: SUM

Fig. 5. *CubeGen* performance evaluation for cube materialization

Efficiency Evaluation. We first evaluate the performance improvement of `CubeGen` for cube materialization. Figure 5(a) and (b) show the execution time of all four algorithms for MEDIAN and SUM respectively. As expected, for both MEDIAN and SUM, our `CubeGen`-based algorithms are 2.2X and 1.6X faster than `MulR_MulS` and `SingR_MulS` on average respectively. This indicates that computing the entire cube in one MR job reduces the overheads significantly compared to the case where multiple MR jobs were issued which requires reading data multiple times. In addition, it also demonstrates that batch processing highly reduces the size of intermediate data which can consequently minimize the overheads of data sorting, shuffling as well as computing.

Impact of Caching Data: Figure 5(a) and (b) also depict the impact of caching data. For MEDIAN, the execution time of the `CubeGen_Cache` is almost the same as `CubeGen_NoCache` as shown in Fig. 5(a). This confirms that our optimization to cache the data through file registration instead of actual data movement does not cause much overhead. For SUM, we observe that `CubeGen_Cache` performs worse than `CubeGen_ NoCache`. This is not surprising as the former needs to write an extra view to the local file system. However, even though `CubeGen_Cache` incurs around 16 % overhead to cache the view, as we will see later, it is superior to `CubeGen_NoCache` when it comes to view updates.

5.2 View Maintenance Evaluation

Efficiency Evaluation: We next study the efficiency of performing the view maintenance in `HaCube` compared with Hadoop. We fix $\mathbb{D}$ with 2.4B tuples in the first cube materialization job and vary the size of $\Delta\mathbb{D}$ from 5 % to 100 % of $\mathbb{D}$ for view updates.

Figure 5(c) shows the execution time for both the cube materialization (`Ini_Cube`) and the view updates (`View_Update`) for MEDIAN. In this set of experiments, we adopt recomputation for view updates of MEDIAN using MR (`Re_MR`) and `HaCube` (`Re_HC`). The result shows that `Re_HC` is 2X and 1.4X faster than `Re_MR`, when $\Delta\mathbb{D}$ is 5 % and 100 % respectively. The gains come from avoiding reloading and reshuffling $\mathbb{D}$ among the cluster. Thus, the larger $\mathbb{D}$ is, the bigger the benefit will be.

Figure 5(f) depicts the result for SUM. As view updates for SUM can either be done by incremental computation or recomputation, we evaluate both approaches to update the view. In Fig. 5(f), `In_MR` and `Re_MR` (resp. `In_HC` and `Re_HC`) are MR (resp. `HaCube`) -based methods using incremental computation and recomputation respectively.

`In_MR` and `Re_MR` are implemented in the way described in Sect. 4.1. In `In_MR`, `Delta_Cube` (in the figure) corresponds to the propagate job to generate the delta view and `View_Update` is the refresh job. The result shows that, for incremental computation, `In_HC` is 2.8X and 2.2X faster than `In_MR` when $\Delta\mathbb{D}$ is in 5 % and 100 % as shown in Fig. 5(f). For recomputation, `Re_HC` is about 2.1X and 1.4X faster than the `Re_MR` when the $\Delta\mathbb{D}$ is in 5 % and 100 % as shown in Fig. 5(f). This indicates that `HaCube` has significant performance improvement compared to MR for the view update for both recomputation and incremental computation.

We observe that incremental computation performs worse than recomputation in both MR and `HaCube`. While this seems counter-intuitive, our investigation reveals that DFS does not provide indexing support; as such, in incremental computation, the entire view which is much larger than the base data (in our experiments) has to be accessed. Another insight we gain is the smaller the $\Delta\mathbb{D}$ is, the more effective `HaCube` is. As future work, we will integrate more existing techniques (e.g. indexing) in DBMS into `HaCube`, which will further improve the view update performance.

Impact of Parallelism: We further analyze the impact of parallelism on `HaCube` for both cube materialization and view update while varying the number of nodes from 10 to 40. The experiments use $\mathbb{D}$ with 600M tuples and $\Delta\mathbb{D}$ in 20 % of $\mathbb{D}$.

Figures 5(d) and (e) report the execution time for MEDIAN and SUM. Note that, in this experiment, incremental computation is used for SUM. We observe that for both recomputation and incremental computation, `HaCube` scales linearly on the testing data set from 10 to 20 nodes, where the execution time almost reduces to half when the resources are doubled. From 20 nodes to 40 nodes, the benefit of parallelism decreases a little bit. This is reasonable, since the entire overheads include two parts, the setup of the framework and the cube computation; the former one may reduce the benefits of increasing the computation resources while cube computation cost is not big enough.

Due to the space limitation, interested readers are referred to our technical report [18] for more experimental evaluations (e.g. load balancing, impact of dimensions) and other issues (e.g. fault tolerance mechanism, storage analysis).

6 Related Work

Much research has been devoted to the problem of data cube analysis [9]. A lot of studies have investigated efficient cube materialization [4,20,21,23] and view maintenance [12,16]. Three classic cube computation approaches (Top-down [23], Bottom-Up [4] and Hybrid [20]) have been well studied to share computation among the lattice in a centralized system or a small cluster environment. Different to these approaches, `CubeGen` adopts a new strategy to partition and batch the cuboids according to their prefix order to tackle the new challenges brought by MR. It utilizes the sorting feature better in MR-like systems such that no extra sorting needed during materialization.

Existing works [17,22] have adopted MR to build closed cubes for algebraic measures. However, both of these works do not provide a generic algorithm that can balance the load to materialize the cube for different measures. Nandi et al. [15] provided a solution to a special case during the cube computation under MR where one reducer gets the "hot spot" group-by cell with a large number of tuples. This complements our work and can be employed to handle such a case in `HaCube`. We note that `HaCube` is able to support all these existing cube materialization algorithms. More importantly, none of these aforementioned works have developed any techniques for view maintenance. In addition, [13] provided one OLAP system by extending HBase for real-time analysis and [19] provides pagrol system for graph OLAP computation.

Our work is also related to the problem of incremental computations. Existing works [5,10,11] have studied some techniques for incremental computations for single operators in MR. HaLoop [6] is designed to support iterative operations through a similar caching mechanism which is used for different purposes under a different application context. Restore [8] also shares the similar spirit to keep the intermediate results (either the output of one MR job or the data operated within one job) to DFS in a workflow and reuse them in the future. For data cube computation, as the size of intermediate results is large, `HaCube` adopts a different data caching mechanism to guarantee the data locality that the cached data can be directly used from local store. This avoids the overhead incurred by Restore in reloading and reshuffling data from DFS. Furthermore, none of these existing works provide explicit support and techniques for data cube analysis under OLAP and data warehousing semantics.

7 Conclusion

It is of critical importance to develop new scalable and efficient data cube computation systems on a big cluster with low-cost commodity machines to tackle the challenges brought by the large-scale of data, to provide a better query response and decision making support. In this paper, we made one step towards developing such a system, `HaCube` an extension of MapReduce, by integrating the good features from both MapReduce (e.g. Scalability) and parallel DBMS (e.g. Local Store). We showed how to batch and share the computations to salvage partial work done by facilitating the features in MapReduce-like systems towards an efficient cube materialization. We also demonstrated how `HaCube` supports an efficient view maintenance by facilitating the extension leveraging a new computation paradigm. The experimental results showed that our proposed cube materialization approach is at least 1.6X to 2.2X faster than the naive algorithms and `HaCube` performs at least 2.2X to 2.8X faster than Hadoop for view maintenance. We expect `HaCube` to further improve the performance by integrating more techniques from DBMS, such as indexing techniques.

Acknowledgements. Kian-Lee Tan is partially supported by the MOE/NUS grant R-252-000-500-112. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

References

1. Hadoop. http://hadoop.apache.org/
2. Tacc longhorn cluster. https://www.tacc.utexas.edu/
3. Tpc-h, ad-hoc, decision support benchmark. www.tpc.org/tpch/
4. Beyer, K.S., Ramakrishnan, R.: Bottom-up computation of sparse and iceberg cubes. In: SIGMOD, pp. 359–370 (1999)
5. Bhatotia, P., Wieder, A., Rodrigues, R., Acar, U.A., Pasquini, R.: Incoop: mapreduce for incremental computations. In: SOCC (2011)
6. Yingyi, B., Howe, B., Balazinska, M., Ernst, M.D.: Haloop: efficient iterative data processing on large clusters. PVLDB **3**(1), 285–296 (2010)
7. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
8. Elghandour, I., Aboulnaga, A.: Restore: reusing results of mapreduce jobs. PVLDB **5**(6), 586–597 (2012)
9. Gray, J., Bosworth, A., Layman, A., Reichart, D.: Data cube: a relational aggregation operator generalizing group-by cross-tab and sub-totals. In: ICDE, pp. 152–159 (1996)
10. Jörg, T., Parvizi, R., Yong, H., Dessloch, S.: Incremental recomputations in mapreduce. In: CloudDB, pp. 7–14 (2011)
11. Lämmel, R., Saile, D.: Mapreduce with deltas. In PDPTA, (2011)
12. Lee, K.Y., Kim, M.H.: Efficient incremental maintenance of data cubes. In: VLDB, pp. 823–833 (2006)
13. Feng Li, M., Ozsu, T., Chen, G., Ooi, B.C.: R-store: a scalable distributed system for supporting real-time analytics. In: ICDE, pp. 40–51 (2014)
14. Mumick, I.S., Quass, D., Mumick, B.S.: Maintenace of data cubes and summary tables in a warehouse. In: SIGMOD, pp. 100–111 (1997)
15. Nandi, A., Cong, Y., Bohannon, P., Ramakrishnan, R.: Distributed cube materialization on holistic measures. In: ICDE, pp. 183–194 (2011)
16. Palpanas, T., Sidle, R., Cochrane, R., Pirahesh, H.: Incremental maintenance for non-distributive aggregate functions. In: VLDB, pp. 802–813 (2002)
17. Sergey, K., Yury, K.: Applying map-reduce paradigm for parallel closed cube computation. In: DBKDA, pp. 62–67 (2009)
18. Wang, Z., Chu, Y., Tan, K.-L., Agrawal, D., Abbadi, A.E., Xiaolong, X.: Scalable data cube analysis over big data. In: CORR (2013). arxiv:1311.5663

19. Wang, Z., Fan, Q., Wang, H., Tan, K.-L., Agrawal, D., El Abbadi, A.: Pagrol: parallel graph olap over large-scale attributed graphs. In: ICDE, pp. 496–507 (2014)
20. Xin, D., Han, J., Li, X., Wah, B.W.: Computing iceberg cubes by top-down and bottom-up integration: the starcubing approach. TKDE **19**(1), 111–126 (2007)
21. Xin, D., Han, J., Wah, B.W.: Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In VLDB, pp. 476–487 (2003)
22. You, J., Xi, J., Zhang, P., Chen, H.: A parallel algorithm for closed cube computation. In ACIS-ICIS, pp. 95–99, (2008)
23. Zhao, Y., Deshpande, P.M., Naughton, J.F.: An array-based algorithm for simultaneous multidimensional aggregates. In: SIGMOD, pp. 159–170 (1997)