R-Store: A Scalable Distributed System for Supporting Real-time Analytics

Feng Li[1], M. Tamer Özsu[2], Gang Chen[3], and Beng Chin Ooi[1]

[1]{li-feng,ooibc}@comp.nus.edu.sg, School of Computing, National University of Singapore, Singapore
[2]tamer.ozsu@uwaterloo.ca, David R. Cheriton School of Computer Science, University of Waterloo, Canada
[3]cg@zju.edu.cn, College of Computer Science, Zhejiang University, P.R. China

***Abstract*—It is widely recognized that OLTP and OLAP queries have different data access patterns, processing needs and requirements. Hence, the OLTP queries and OLAP queries are typically handled by two different systems, and the data are periodically extracted from the OLTP system, transformed and loaded into the OLAP system for data analysis. With the awareness of the ability of big data in providing enterprises useful insights from vast amounts of data, effective and timely decisions derived from real-time analytics are important. It is therefore desirable to provide real-time OLAP querying support, where OLAP queries read the latest data while OLTP queries create the new versions.**

In this paper, we propose R-Store, a scalable distributed system for supporting real-time OLAP by extending the MapReduce framework. We extend an open source distributed key/value system, HBase, as the underlying storage system that stores data cube and real-time data. When real-time data are updated, they are streamed to a streaming MapReduce, namely Hstreaming, for updating the cube on incremental basis. Based on the metadata stored in the storage system, either the data cube or OLTP database or both are used by the MapReduce jobs for OLAP queries. We propose techniques to efficiently scan the real-time data in the storage system, and design an adaptive algorithm to process the real-time query based on our proposed cost model. The main objectives are to ensure the freshness of answers and low processing latency. The experiments conducted on the TPC-H data set demonstrate the effectiveness and efficiency of our approach.

I. Introduction

Database systems implemented for large scale data processing are typically classified into two categories: OLTP systems and OLAP systems. The data stored in OLTP systems are periodically exported to OLAP systems through Extract-Transform-Load (ETL) tools. In recent years, MapReduce [8] framework has been widely used in implementing large scale OLAP systems because of its scalability, and these include Hive [26], Pig [23]. Most of these only focus on optimizing OLAP queries, and are oblivious to updates made to the OLTP data since the last loading. However, with the increasing need to support real-time analytics, the issue of freshness of the OLAP results has to be addressed, for the simple fact that more up-to-date analytical results would be more useful for time-critical decision making. The idea of supporting real-time OLAP (RTOLAP) has been investigated in traditional database systems. The most straightforward approach is to perform near real-time ETL by shortening the refresh interval of data stored in OLAP systems [27]. Although such an approach is easy to implement, it cannot produce fully real-time results and the refresh frequency affects system performance as a whole. Fully real-time OLAP entails executing queries directly on the data stored in the OLTP system, instead of the files periodically loaded from the OLTP system. To eliminate data loading time, OLAP and OLTP queries should be processed by one integrated system, instead of two separate systems. However, OLAP queries can run for hours or even days, while OLTP queries take only microseconds to seconds. Due to resource contention, an OLTP query may be blocked by an OLAP query, resulting in a large query response time. On the other hand, since complex and long running OLAP queries may access the same data set multiple times, and updates by OLTP queries are allowed as a way to avoid long blocking, the result generated by the OLAP query would be incorrect (the well-known dirty data problem).

In this work, we try to address the problem of large scale RTOLAP processing using MapReduce. The RTOLAP is defined as follows: a real-time OLAP (RTOLAP) query accesses, for each key, the latest value preceding the submission time of the query [9]. Specifically, we propose and design a scalable distributed RTOLAP system called R-Store, in which the storage system supports multi-versioning, and each version is associated with a timestamp. Each OLAP query operates on the version of data that exists at the time it is submitted whereas each OLTP transaction creates a new version. R-Store uses the MapReduce framework where the mappers of the OLAP query directly access the real-time data stored in the storage system. The storage system is implemented by extending HBase [1]. HBase supports the `HBaseScan` operation that takes a timestamp as input and returns the version of the data with the largest timestamp before the scan operation. Though this can be used to offer consistent data to OLAP queries, simply using this default scan operation for querying the data stored in HBase is inefficient due to the following reasons:

1) HBase only stores a fixed number of versions for each key, and automatically removes the versions that exceed this cap by its default compaction policy. To support real-time querying, this number has to be set to infinity in case the old versions are removed during the running of an OLAP query. However, this will lead to continuously increasing of the data size, and waste too much space to store the unused data.
2) For each OLAP query, the entire HBase table has to be scanned and shuffled to the mappers, which is a very costly process.

To facilitate efficient processing of RTOLAP queries, we periodically materialize the real-time data into a data cube and implement an `IncrementalScan` operation in HBase to avoid the shuffling of the entire HBase table to MapReduce during real-time querying. To the best of our knowledge, this is the first work that proposes a scalable RTOLAP distributed system based on MapReduce framework. In summary, the contributions of this paper are as follows:

1) We propose a scalable distributed system framework called R-Store, for performing RTOLAP. R-Store evaluates an OLAP query by transforming it into a MapReduce job, which is run on our modified HBase (in the remaining of this paper, we name it as HBase-R in order to differentiate it from HBase), to obtain the real-time data.
2) We propose an efficient storage model for caching the data cube result. The data cube is treated as historical data, while the data updated after the refresh time of the data cube are real-time data. We also propose a more efficient scan operation in the storage model for obtaining the real-time data.
3) We integrate streaming MapReduce into our system, which maintains a real-time data cube in the reducers, and periodically materializes the data cube. This data cube update method is much faster than the data cube re-computation method, and in turn accelerates the processing of RTOLAP since fewer real-time data are scanned during the query execution.
4) We design an algorithm to efficiently process the RTOLAP queries, which takes both the historical data cube and the real-time table as input. We also propose a cost model that guides the adaptive processing of RTOLAP.
5) We perform an extensive experimental study on a cluster with more than one hundred nodes, which confirms the effectiveness of the cost model, and the efficiency and scalability of R-Store.

The remaining of the paper is organized as follows. In Section II, we review some related research. We subsequently present the architecture, design and implementations of R-Store in Section III and IV. In Section V, we discuss the processing of real-time OLAP. We evaluate the performance of R-Store in Section VI and conclude the paper in Section VII.

II. Related Work

Our proposal touches on a number of areas such as OLAP processing, distributed processing and data cube maintenance. We review related work that are most relevant to ours.

A. Real-Time Data Warehousing

The growing demand for fast business analysis coupled with increasing use of stream data have generated great interest in real-time data warehousing [28]. Some have proposed near real-time ETL [27], as a means to shorten the data warehouse refresh intervals. These works require fewer modifications to the existing systems, but they cannot achieve 100% real-time. Other studies proposed online updates in data warehouses by using differential techniques [12], [25], or multi-version concurrency control [17]. In C-store [25] two separate stores are used to handle in-place updates. The updates are stored in a write-store (WS), while queries run against the read-store (RS), and merged with the WS during execution. In existing studies, the incoming updates are usually cached to improve the performance. The cached data are then flushed to disk once the size exceeds the upper bound. The performance of these studies are limited by the size of the memory, and MaSM [4] overcomes these limitations by utilizing SSDs to cache incoming updates.

B. Distributed Processing

MapReduce is a parallel data processing framework for large scale data processing [8]. Its programming model consists of two user-defined functions, `map` and `reduce`, that operate on key/value pairs. A survey on database management using MapReduce can be found in [19], and a detailed performance study for MapReduce has been proposed in [14]. In this paper, we augment MapReduce processing platform with an extended HBase for data cube storage and maintenance.

There have been some researches on supporting both OLTP and OLAP in one hybrid system. Previously, we have proposed epiC [5], [13], an elastic power-aware data-itensive cxloud platform for supporting both OLAP and OLTP. As part of the epiC work, in this paper, we investigate how to efficiently process the RTOLAP queries in such a hybrid system. The concepts proposed in this paper can also be applied to epiC though they're implemented using MapReduce. In addition, there are also some main memory database systems (HyPer [16], HYRISE [10]) that tries to address both OLTP and OLAP. The focus of our work is different from these systems: our work tries to address the problem of efficiently processing the RTOLAP queries in a large scale environment where the data are stored on disk.

While MapReduce provides an efficient and simple platform for scalable distributed processing, it is not efficient for supporting online and continuous stream processing. HStreaming [2] and MapReduce Online [7] are extensions made to the MapReduce framework that support stream processing as follows: (1) the input of the mappers could be stream data; (2) the data are streamed from mappers to reducers; and (3) the output of one MapReduce job can be streamed to the next job. S4 [22] is another distributed stream system that adopts the actor model as its computation model. In these works, the new data are usually appended to the system to form the data stream, and the previously inserted data are not changed. In contrast, in our work, we support the OLTP operations such as insert, update and delete, and these changes in turn have an effect on the result of the real-time OLAP queries.

C. Data Cube Maintenance

Data cube maintenance has been studied for a long time. The earliest works focused on efficient incremental view maintenance for data warehouses [6], [11]. However, as the number of dimension attributes increases, the cost of incrementally updating data cube increases significantly. To improve the performance of data cube maintenance, instead of generating the delta value for all the cuboids during the update process, an method of refreshing multiple cuboids by the delta value of a

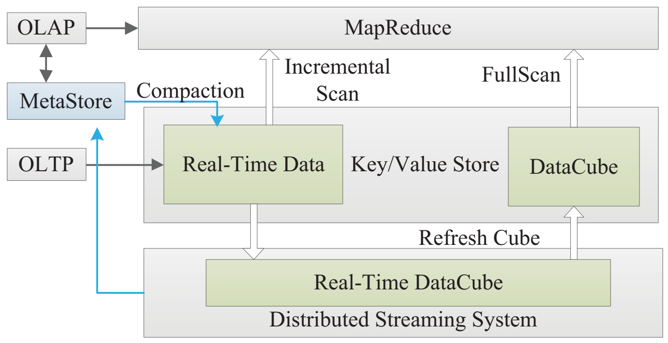


Fig. 1. Architecture of R-Store

single cuboid has been proposed [18]. Most of these algorithms were designed for a single node configuration and are not scalable to a distributed environment. However, MapReduce has been used to construct data cube in a large scale distributed environment [24]. The MR-Cube algorithm [21] was proposed to efficiently compute the data cube for holistic measures. In these works, the data cube is usually used for processing OLAP queries without the real-time requirement, while our system considers both the data cube and the real-time data to process RTOLAP queries.

III. R-Store Architecture and Design

In this section, we present the architecture of R-Store, the design philosophy of the storage system, and how the data cube is maintained.

A. *R-Store Architecture*

Figure III-A illustrates the architecture of R-Store. The system consists of four components: a distributed key/value store, a streaming system for maintaining the real-time data cube, a MapReduce system for processing large scale OLAP queries, and a MetaStore for storing some global variables and configurations.

The OLTP queries are submitted directly to the key/value store, while the OLAP queries are processed by the MapReduce system. The simplest method of supporting RTOLAP for MapReduce is to scan the whole real-time table and obtain the latest version before the submission time of the OLAP query for every key/value pair (`FullScan` operation), as the input of the MapReduce job. The key/value store has to support multi-version concurrency control in case the OLTP queries and OLAP queries are blocked by each other. However, this method is not efficient because obtaining one version for each key/value pair is a costly operation in large scale distributed systems. Note that in real applications, such as social networks, the updates usually follow a Zipf distribution, and within a time interval, only a small portion of keys are updated in the table. Based on this observation, we try to accelerate OLAP queries by materializing the real-time table into a data cube. When an OLAP query is submitted to the system, it first connects to *MetaStore* to acquire the timestamp of the query for consistency. The statistics stored in *MetaStore* are also used to optimize the query based on our proposed cost model (Section V-C). After the optimization by the cost model, the OLAP query can be transformed to a MapReduce job that takes as input both the historical data in the data cube and the real-time data in the key/value store. To efficiently access real-time data, the key/value store is designed to support incremental scan (Section III-B1). The real-time data is scanned by the `IncrementalScan` operation, while the data cube is scanned by the `FullScan` operation. The `IncrementalScan` operation only shuffles the key/value pairs that are updated after the last building of the data cube, and thus is much faster than `FullScan` because fewer data are shuffled.

The data cube is also stored in the distributed key/value store and is periodically refreshed based on the real-time table. The versions of the key/value pairs before the refresh time of the data cube are compacted in order to accelerate the scan time of the real-time table. The performance of refreshing the data cube is crucial to our system because if the data cube is refreshed fast, more data are compacted by our compaction scheme, and fewer real-time data are accessed during the scan operation. In an extreme case where no update is submitted since the data cube refresh, the MapReduce job only needs to scan the data cube. To efficiently refresh the data cube, the updates applied to the key/value store are streamed to the streaming system, and a real-time data cube is maintained in the local storage of the streaming system. The real-time data cube is periodically materialized to the key/value store to refresh the data cube. Based on our experimental results, this method is much faster than the method of re-computing the data cube, and the throughput of this method is sufficiently high to process the update streams from the key/value store.

Once this refresh process is completed, the timestamp of the latest data cube is sent to *MetaStore*, and the compaction process is invoked to compact the real-time data. The *MetaStore* also stores other global information, including the submission time of each OLAP query, the frequency of materializing the data cube, etc.

B. *Storage Design*

The key/value store must support multi-version concurrency control techniques to ensure that the OLAP query and the OLTP query do not block each other. In addition, our storage design considers many other features including efficient file scan operations, compaction scheme and load balancing, which we discuss below.

1) Full and Incremental Scans: To handle OLAP queries and to build the data cube, a scan operation needs to be implemented in the key/value store. Two types of scan operations are required, which are used in different scenarios:

FullScan(T_i). For each *region* of the table in the key/value store, the `FullScan` operation takes a timestamp T_i as input, and returns the latest version of the value before T_i for all the keys. The data returned by this operation can be used to create or re-compute the data cube.

IncrementalScan(T_1, T_2). This operation takes two timestamps T_1 and T_2 ($T_1 < T_2$) as input, and returns two versions for the keys updated after T_1. The first version is the latest value before T_2, and the second version is the latest value before T_1. If a new key is inserted (not updated) into the store after T_1, only one version is returned. This operation can be used in the RTOLAP query processing algorithm.

2) Global and Local Compactions: Since each key may have several versions, the scan operations read more than one version of the data to obtain the required versions, incurring unnecessary I/O cost. To reduce the number of stored versions for each key and to improve the scan performance, the data are automatically compacted. We provide two forms of compaction:

Global Compaction. The global compaction process is launched immediately following each data cube refresh. For the same key, all the versions inserted before the data cube refresh are merged into one version. We call this version V_{DC}, which is consistent with the data cube and is used in updating the data cube.

Local Compaction. The local compaction process is invoked on each node. At first, the submission time of the current running scan process (T_{scan}) on the *region* is acquired by the compaction process. For each key, the latest version of the data before T_{scan} is accessed by this scan process. Thus, the local compaction only compacts the older versions that will not be accessed by any scan process. Furthermore, V_{DC} is not changed during this compaction so that the new data cube can be computed correctly when the data cube is refreshed.

3) Load Balancing: In most applications, some key ranges might be updated more frequently than others, causing skewed load on the nodes. In addition, since the update operation inserts a new version of the key into the node instead of replacing the old version, there is a skew on the amount of data on the nodes. This affects the performance of the scan process on those nodes. The solution is to split heavily updated ranges in the key/value store and move some data to other nodes.

C. Data Cube Maintenance

To improve the performance of OLAP queries, a data cube is maintained in the key/value store. A data cube could be either a full data cube, an Iceberg cube, or a closed cube. The selection of the best suitable data cube depends on the applications, which is not the focus of this work. To make it general, we only consider the full data cube, which consists of a lattice of cuboids. There are two approaches to refresh the data cube:

Re-Computation. To re-compute the data cube, the `FullScan` operation is used. It takes a timestamp T_i as input, and returns the latest version of the value before T_i for all the keys stored in this *region*. Each mapper of the MapReduce job takes the results of the `FullScan` operation on one *region* as input. For each cuboid, a key/value pair is generated. The map output key is the combination of the dimension attributes for the cuboid, while the map output value is the numeric value. The reducers compute the aggregation value for each cell of each cuboid, and output the result to the key/value store.

Incremental Update. The second approach is performed in two steps: propagation step and update step. The propagation step computes $\triangle DC$ (change of the data cube) from $\triangle T$ (change of the table), and the update step updates data cube based on $\triangle DC$. However, not all data cubes can be incrementally updated. The incremental update only works for self-maintainable aggregate functions [20] (the new cell value can be computed from the old cell value and the updated tuples) such as SUM, COUNT, and the algebraic functions derived from them.

In R-Store, the re-computation approach is used to build the first data cube, while the incremental update approach is adopted to maintain a real-time data cube in the stream processing module. The streaming system updates its data cube with the update streams coming from the key/value store, and periodically materializes the data cube into the storage system. As the updating of the data cube consists of two phases, which can be processed by the MapReduce processing logic in natural, a streaming version of MapReduce is used as the stream processing module of R-Store.

IV. R-Store Implementations

In this section, we present the implementations of R-Store. Specifically, we show how we implement our storage system, namely HBase-R, on top of HBase to fulfill the design philosophy discussed in Section III-B.

A. Implementations of HBase-R

HBase [1] is an open source distributed key/value store. A table stored in HBase is partitioned to several *regions*, which are assigned to a certain nodes, and each node runs a *region* server to manage *regions* and serve the transactions. Inside a region, the data of the same column family (a group of columns) are stored in the same structure, which is called *store*. A *store* has an in-memory structure, *memstore*, and several in-disk files, *storefiles*. When a new version of data is about to be inserted into this *store*, it is first inserted into the *memstore* and appended to the write ahead logs. Once the size of the *memstore* reaches its upper bound, the data in the *memstore* are transferred to a *storefile*. The *storefiles* are sorted in inverse chronological order. Inside the *memstore* or *storefile*, the data are sorted by keys, and the versions for each key are sorted in inverse chronological order. HBase only supports the `FullScan` operation, so we designed and implemented `IncrementalScan` in HBase-R.

1) IncrementalScan: For a *store* in a *region*, by accessing the same key across the *storefiles* and *memstore* in parallel, the `IncrementalScan` operation scans the keys in ascending order. For each key, the version with the larger timestamp is scanned earlier. For all the versions of a key, the algorithm checks the timestamp of each version and returns the required two versions. If the key has only one version, which means the operation on the key is an insertion, the `IncrementalScan` only returns that version for the key.

For real-time queries and data cube update, scanning the key/value pairs in HBase-R is the most costly step. It is, therefore, important to improve the performance of `IncrementalScan`. For this purpose, we propose an adaptive incremental scan algorithm.

First, we maintain an in-memory structure to estimate $d(T)$, the number of distinct keys updated since the last refresh of the data cube. Estimating $d(T)$ in a data stream has been well studied [15]. A straightforward method is to keep all the keys in memory, and, for each key, to maintain a bit value to indicate whether or not it has been updated. However, this method requires a considerable amount of memory to store the keys. In HBase-R, the size of a *region* is configured before the data are inserted. Thus, the number of keys for a *region* has an upper bound (M), which can be estimated by $SizeOfRegion/SizeOfKeyValue$. Since each *region* usually stores a range of consecutive keys, a hash function $h(key)$ can be used to map a key to a value between 0 and $M-1$, and a bit array of size M, *DistinctKeys*, is maintained in memory to indicate whether or not a key has been updated. Using this bit array, to compute the number of updated values on a node with even one billion distinct keys, only 128 MB of memory are required.

To improve the performance of `IncrementalScan`, the above data structure is used in the adaptive incremental scan algorithm (Algorithm 1). When an `IncrementalScan` request is sent to a *region* server, the first parameter (T_1) is always set to the refresh time of the current data cube (T_{DC}), and the second parameter (T_2) equals to the submission time of the query (T_Q). Instead of scanning all the key/value pairs before T_Q, the key/value pairs in *memstore* are scanned first. Note that in *memstore*, there might be several versions for a key, and only the newest version is cached in *kvMap* (line 1). The number of key/values updated after T_{DC} but not in *memstore* is then computed (line 7), and the random read cost of these key/values is estimated. If this cost is smaller than the cost of scanning all the data between T_{DC} and T_Q, the *storefile* index is used to directly read the values for these keys (lines 8 to 14). In this way, the latest versions for the updated keys are obtained. Then, by simply scanning the key/values before T_{DC}, the latest versions before T_{DC} for the updated keys are returned to the client. Since the cost of scanning *memstore* (in-memory structure) is much lower than the cost of scanning *storefile*, when $d(T)$ is large, the adaptive incremental scan is almost the same as the default *IncrementalScan*. In contrast, when $d(T)$ is small, this adaptive scan strategy incurs fewer I/O operations.

2) Compaction: HBase's default compaction process combines all the *storefiles* into one file and retains only one version for each key. The global compaction in HBase-R is similar to HBase's default, but with a different triggering condition; local compaction only compacts the versions earlier than a certain timestamp. To ensure that the compaction process does not block the scan processes, the compacted data are stored in different files, instead of directly replacing the un-compacted data. The files that contain the old versions are replaced by the compacted files when they are not accessed by any scan process. Since the compaction process competes with OLAP queries for CPU and I/O resources, there is a trade-off between the frequency of the compaction and the performance of the whole system. We define a threshold so that the local compaction process is triggered when $(numberOfTuples)/(numberOfDistinctKeys)$ exceeds this threshold.

3) Load Balancing: HBase has its default *region* size, which is 256MB. If the size of the data for a *region* is larger than this size, it is automatically split to two sub-*regions*, which are distributed to other nodes. In HBase's default setting, only a fixed number of versions for a key are stored. Once the number of versions for all the keys in this *region* reaches the maximum number, the size of the *region* would not change regardless of the frequency of key updates in this *region*. This requires users to manually split the hot *region*. In contrast, in R-Store, we do not strictly remove the old versions of the updated keys once the number of versions exceeds HBase's default setting. We wait until the size of frequently updated *region* reaches its upper bound, and the split happens automatically.

Algorithm 1: Adaptive IncrementalScan

```
   input: Timestamp $T_{DC}$, Timestamp $T_Q$, boolean[]
          DistinctKeys, int NumDistinctKeys
1  kvMap ← new HashMap<Key, Value>();
2  for KeyValue kv ∈ MemStore do
3      if kvMap.contain(kv.key) then
4          continue;
5      else
6          kvMap.put(kv.key, kv.value);
7  NumKeysNotInMemory ← NumDistinctKeys -
   kvMap.size();
8  if $CostOfRandom \times NumKeysNotInMemory <$
   $CostOfScan \times NumOfUpdatedKeyValues$ then
9      for key updated but not in kvMap do
10         kv ← randomRead(key);
11         kvMap.put(kv.key, kv.value);
12     for each kv before $T_{DC}$ do
13         if kvMap.exist(kv.key) then
14             send kvMap(kv.key) and kv;
15 else
16     delete kvMap;
17     invoke the default IncrementalScan($T_{DC}$, $T_Q$)
```

B. Real-Time Data Cube Maintenance

R-Store adopts HStreaming for maintaining the real-time data cube (note that other streaming MapReduce systems can also be used in R-Store). Each mapper of HStreaming is responsible for processing the updates within a range of keys. The map function of the data cube update algorithm is shown in Algorithm 2. When an update for a key arrives, the old value for this key is retrieved from the local storage if exists. To efficiently retrieve the old value, a clustered index is built for the key/values, and the frequently updated keys are cached in memory. In reality, the updates are usually on a small range of keys, and the old value of the updates have a high probability to be directly retrieved from the cache. If the key is new (thus, does not exist in local storage), for each cuboid, one key/value pair is generated and shuffled to the reducers. The map output key is the combination of the dimension attributes, and the map output value is the numeric value. If the key of the update exists in local storage and the updated key/value pair falls into the same cell for a cuboid, one key/value pair is shuffled to the reducer, and the numerical value is equal to the value change. Otherwise, two key/value pairs are generated, one is the new value with a tag "+", and the other is the old value with a tag "-".

The reduce function is invoked at a time interval w_r

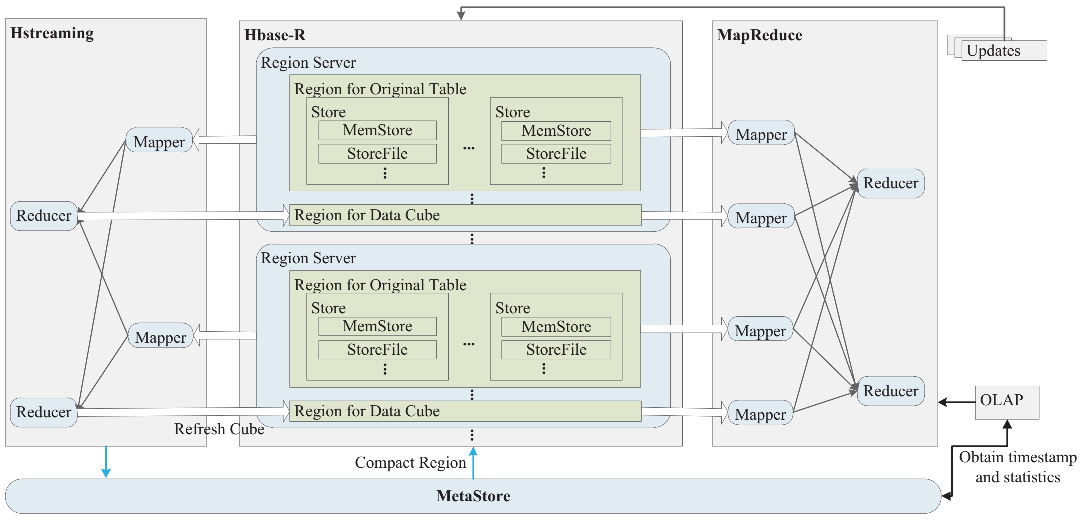


Fig. 2. Data Flow of R-Store

```
Algorithm 2: Map Function for Incremental Update
   input: KeyValue kv
1  oldkv = retrieveFromLocal(kv.key);
2  if oldkv == null then
3      for cuboid in data cube do
4          CuboidK ← extractCuboidKey(cuboid,
           kv.value);
5          CuboidV ← extractCuboidValue(kv.value);
6          CuboidV.setTag(“+”);
7          Emit(CuboidK, CuboidV);
8      insertToLocal(kv);
9  else
10     oldCuboidV ← extractCuboidValue(oldkv.value);
11     oldCuboidV.setTag(“-”);
12     newCuboidV ← extractCuboidValue(kv.value);
13     newValue.setTag(“+”);
14     for cuboid in data cube do
15         oldCuboidK ← extractCuboidKey(cuboid,
           oldkv.value);
16         newCuboidK ← extractCuboidKey(cuboid,
           kv.value);
17         if oldCuboidK == newCuobidK then
18             newCuboidV.set(computeChangeOfCell
               (oldCuboidV,newCuboidV));
19             Emit(newCuboidK, newCuboidV);
20         else
21             Emit(oldCuboidK, oldCuboidV);
22             Emit(newCuboidK, newCuboidV);
23     updateToLocal(kv);
```

specified by the user. For example, if the time interval is set to one second, the reducers will cache the incoming intermediate data within the past second, and apply the reduce function to them. Another time interval, w_{cube}, defines how frequently the data cube is materialized. The reduce function to incrementally update the data cube is shown in Algorithm 3. A reducer merges the local data cube (DC) with the intermediate

```
Algorithm 3: Reduce Function for Incremental Update
   input: Key key, List¡Value¿ vlist, Context context
1  i ← 0, sum ← 0;
2  for Value v in vlist do
3      if v.timestamp < $T_{DC}$ then
4          MergeWith(key, v, $DC$)
5      else
6          MergeWith(key, v, $\triangle DC'$)
```

key/value pairs that it receives from mappers (which is a cell in a cuboid) if these are due to an update before next cube refresh time (T_{DC}). Otherwise, it stores these key/value pairs in $\triangle DC'$. When the timestamps of the incoming updates on all mappers are larger or equal to T_{DC}, the data cube refresh process is invoked, which writes the local data cube to HBase-R (different cuboids are written to separate HBase-R tables). The incoming cells during this refresh process are still written to $\triangle DC'$ since their timestamps are no less than T_{DC}. When this refresh process is completed, T_{DC} is incrementally changed, and DC is merged with $\triangle DC'$. In streaming system, to deal with fault tolerance, the accumulated states of the stream computation have to be checkpointed periodically. The data streams after the checkpointing time are stored in logs and will be used during the recovering process. In R-Store, the data cube materialized to key/value store is indeed a checkpointing of the real-time data cube. Since the key/value pairs after the last data cube refresh are still stored in the storage (even though some intermediate versions of the key/value pairs might be removed by the local compaction process, the necessary versions for building the next data cube are still there), the real-time data cube maintenance process can be recovered using the data cube and the real-time table without extra efforts of checkpointing.

C. Data Flow of R-Store

Figure 2 illustrates the data flow between HBase-R, HStreaming and MapReduce in R-Store. Each HBase-R *region* server handles several *regions*. Some of these *regions* belong to the real-time table, while the others belong to the data cube. An OLTP query is submitted to one of the *region* servers, and stored in *memstore* of the *region* it belongs to. If the size of the *memstore* reaches its upper bound, the data are written into HDFS as a *storefile*. Once the update is written to HBase-R, it is streamed to a mapper in HStreaming based on the key of this update. In the mappers of HStreaming, the change of a cell for each cuboid is computed and shuffled to reducers. On the reduce side, the real-time data cube is updated and cached in local disk. At time interval, HStreaming materializes its local data cube into HBase-R and notifies *MetaStore* with the timestamp of the latest data cube. The compaction process is then launched to compact the versions of data before data cube is refreshed.

When an OLAP query arrives, it acquires a timestamp from the *MetaStore*, together with the statistics of the real-time table stored in HBase-R. It is then transformed to a MapReduce job based on the data statistics, and submitted to the system. Each mapper starts a scan operation over its input *region* belonging to either the real-time table or the data cube. At the end of the job, the results of OLAP query are stored in HBase-R.

V. Real-Time OLAP

Section III to IV described in detail the architecture and implementation of R-Store. In this section, we discuss how the real-time OLAP queries are processed. In R-Store, if the input of the MapReduce job is only the data cube, the performance of the scan phase on the map side is maximized, but the result might be stale. To maximize the freshness of the OLAP query, all the updated key/value pairs before the submission time of the query must be considered. Thus, not only the data cube, but also the real-time table must be scanned.

Suppose the creation time of the data cube is T_{DC} and the submission time of the query is T_Q. For each updated key after T_{DC}, `IncrementalScan` running on the real-time table returns both the old version before T_{DC} and the latest version before T_Q, if its two parameters are set to T_{DC} and T_Q respectively. By merging these two versions with the numeric values of each cuboid, the latest cuboid value can be computed on demand, and the freshness of the OLAP query can be satisfied. In the following subsection, we present the query processing algorithm (called *IncreQuerying*) making use of the `IncrementalScan` operation.

A. Querying Incrementally-Maintained Cube

We implement *MultiTableInputFormat* so that each MapReduce job can scan the data of multiple tables, and the scan operation of each table can be configured as either full scan or incremental scan. Using this input format, the MapReduce job for *IncreQuerying* can access two types of input tables: one is the cuboid table for which a full scan is performed, and the other is the real-time table over which the incremental scan is used.

Map. Algorithm 4 describes the map function. The mappers filter the cell and the real-time tuple based on the filtering condition. The cells and tuples that will be aggregated are assigned the same partition key and shuffled to the same reducer. The output value for the cell is the selected numeric value, while the output value for the real-time tuple is the original value, which will be used to re-compute the numeric value. The value is attached with a tag "Q", "-" or "+" to indicate whether it is the cell value of a cuboid, the old value of a key/value pair, or the new value, respectively. This phase is similar to the map phase of incrementally updating the data cube, except that a filtering process is added, and the partition key could be different from the dimension attributes of the data cube.

Algorithm 4: Map Function for IncreQuerying Algorithm

```
input: KeyValueList kvlist, Context context
1  key ← null, value ← null;
2  if kvlist.size == 1 then
3      key ← extractKey(kvlist[0].key);
4      if key is not filtered then
5          value ← kvlist[0].value;
6          value.setTag("Q");
7          Emit(key, value);
8  else
9      key ← extractKey(kvlist[0].value);
10     if key is not filtered then
11         value ← extractValue(kvlist[0].value);
12         value.setTag("+");
13         context.write(key, value);
14         value ← extractValue(kvlist[1].value);
15         value.setTag("−");
16         Emit(key, value);
```

TABLE I. Data Cube Operations

Operator	Parameters
`addFilter`	attribute name, function, value
`addGroupBy`	group-by attribute
`setAggregationFunc`	aggregation function name
`setNumericAttribute`	numeric attribute name

Reduce. The reduce function calculates the new value of each cell based on the old cell value, the change of the cell and the aggregation function. The cell key of the reduce function is different from that of Algorithm 3. For example, for the TPC-H $part$ table, to compute a rectangular subset of the cube ($mfgr$ = "Manufacturer#13"), the key of the reduce function is the combination of the attributes (*brand* to *container*) after removing $mfgr$.

Figure 3 shows the data flow of *IncreQuerying* alogrithm for an OLAP query on a two-dimensional cuboid (*mfgr*,*brand*). The query computes the summation of *price* for each brand produced by "M1". To ensure the freshness of the results, all the data of the queried table and the cuboid are scanned to process the real-time query. Note that the row key of the stored data cuboid is the combination of the dimension attributes. Therefore, if the filtering condition contains some attributes that could form a prefix of the row key, such as "Manufacturer#1" and "Brand#13", the range scan function of HBase-R can be used to avoid scanning the entire data cube. The min key for the range scan is "Manufacturer#1,Brand#13", and the max key is "Manufacturer#1,Brand#14".

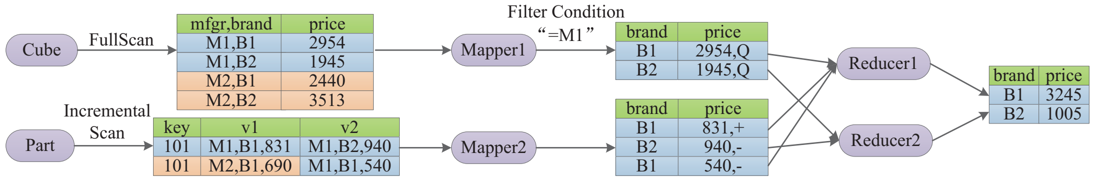


Fig. 3. Data Flow of IncreQuerying

```
Algorithm 5: Example Data Cube Query
  input: DataCube cub
1 cub.addFilter("mfgr", "=", "Manufacturer#1");
2 cub.addFilter("brand", "=", "Brand#13");
3 cub.addGroupBy("type");
4 cub.setNumericAttribute("retailprice");
5 cub.setAggregateFunc("sum");
6 cub.setOutputTable("resultTable");
7 SubmitQuery(cub);
```

To relieve users from having to merge the real-time data and the historic data cube, we define new data cube operators and automatically translate these operators into a MapReduce job. The processing of the real-time data is transparently encapsulated into the operators shown in Table I. Algorithm 5 shows an example that computes the summation of the *retailprice* for all the parts with "Brand#13" produced by "Manufacturer#1", grouped by *type*.

B. Correctness of Query Results

When an OLAP query is submitted to the system, a timestamp T_Q is acquired for this query from the *MetaStore*. To guarantee correctness, if the query needs to scan a table several times, the scan process on each node always returns the data before time T_Q. However, in a distributed system, although clocks can be synchronized to a certain extent, there might still be some difference between the clocks of different nodes. If the current timestamp T_k on a certain node k is smaller than T_Q, the next scan process on this node would return some data between T_k and T_Q, which leads to an inconsistent state. To avoid this inconsistency, if the timestamp T_Q is larger than T_k, the scan process is blocked for a while until T_Q is equal to or smaller than T_k. Since clock synchronization can achieve one millisecond accuracy in local area networks under ideal conditions, the delay of the scan process can be ignored compared to the processing time.

C. Cost Model

The *IncreQuerying* algorithm discussed above is not always better. The `IncrementalScan` scans not only the real-time table, but also the data cube, incurring a higher cost. In addition, it shuffles two versions for each updated key to MapReduce. When there are fewer OLTP transactions or the OLTP transactions access a small range of keys, *IncreQuerying* algorithm is better because `IncrementalScan` only transfers a small amount of data to the mappers. An alternative implementation of real-time querying is similar to re-computing the data cube: a `FullScan` operation is used to return one version for each key/value pair regardless of whether or not it has been updated. When the updates are uniformly distributed across all the keys, this baseline implementation could be more efficient. To be able to select a more efficient approach, we propose a cost model. Table II shows the parameters of the cost model. The most important one is $s(T)$, which is the percentage of the keys that are updated after refreshing the data cube: $s(T) = d(T)/|T|$.

TABLE II. PARAMETERS

Parameter	Definition
$\|T\|$	number of tuples in table T
$d\|T\|$	number of distinct keys updated
$f(T)$	size of the tuple in table T
$s(T)$	percentage of the keys that are updated since the last data cube refresh
$\|C\|$	number of cells in the selected cuboid
$d(C)$	size of dimension attributes of cuboid
$n(C)$	size of numeric attribute of cuboid
$\|Q\|$	number of tuples in the query result
$s(Q)$	filtering selectivity of the query
$d(Q)$	size of query result key
$n(Q)$	size of query result value
sh_{HBase}	cost ratio of shuffling from HBase-R
w_{HBase}	cost ratio of HBase-R writes
sh_{MR}	cost ratio of shuffling in MapReduce
c_L	cost ratio of local I/Os
m_T	number of mappers for table T
m_C	number of mappers for the cuboid
B	block size

1) Cost Analysis of the Baseline Method: The baseline algorithm is a MapReduce job that is essentially similar to re-computing the entire data cube. First, we estimate the cost of the scan phase on the map side. The scan phase consists of two parts: scanning the local data on each HBase-R node (`FullScan` or adaptive `IncrementalScan` discussed in Section III-B1) and shuffling these data to mappers. The `FullScan` scans all the *storefiles* of the real-time table, while the adaptive `IncrementalScan` scans fewer *storefiles* when $d(T)$ is small and the $memstore$ has enough number of keys. However, whether the adaptive `IncrementalScan` is activated depends on the status of each HBase-R node and cannot be easily estimated. Thus, we assume that the cost of reading the local data on each HBase-R node are the same for `FullScan` and `IncrementalScan`. The difference is in the number of tuples transferred from HBase-R to mappers. Thus, we base our analysis on the network transfer cost. The `FullScan` operation shuffles one version for each key, and the cost of the scan phase of the MapReduce job is estimated as

$$C_{scan} = sh_{HBase} \times |T| \times f(T)$$

The mapper outputs one key/value pair for each tuple. Thus the size of the map output is

$$S_{MO} = s(Q) \times (|T|/m_T) \times (d(Q) + n(Q))$$

and the cost of external sorting the map output is:

$$C_{sort-map} = m_T \times 2c_L \times ((S_{MO} \times log_B(S_{MO}/(B+1)))$$

When the mappers complete, the reducers start to pull the sorted key/value pairs from the mappers. The cost of shuffling is:

$$C_{shuffling} = sh_{MR} \times s(Q) \times |T| \times (d(Q) + n(Q))$$

The pulled data are cached in the local file system, whose cost can be ignored since the shuffling process and the local writing process are pipelined. The data shuffled from the mappers are then merged into one sorted run using the multi-way merge-sort method, which only requires reading and writing the files once.

$$C_{reduce-merge} = 2c_L \times s(Q) \times |T| \times (d(Q) + n(Q))$$

Finally, the result table is written into HBase-R, whose cost is computed as

$$C_{reduce-write} = w_{HBase} \times |Q| \times (d(Q) + n(Q))$$

2) Cost Analysis of IncreQuerying Algorithm: The MapReduce job for *IncreQuerying* reads both the data cube and the updated data. Therefore, its cost is different from that of re-computation. At first, the mappers scan both the real-time data and the data cube. The cost of shuffling the real-time data and data cube to mappers is:

$$C_{scan-R} = sh_{HBase} \times 2|T| \times f(T) \times s(T)$$

while the cost of scanning the data cube is:

$$C_{scan-C} = sh_{HBase} \times |C| \times (d(C) + n(C))$$

After the scan phase, the real-time data and the data cube are sorted. The size of the map output for these two types of data is:

$$S_{MO-R} = 2 \times (s(Q) \times |T| \times s(T)/m_T) \times (d(Q) + n(Q))$$
$$S_{MO-C} = (|C|/m_C) \times s(Q) \times (d(Q) + n(Q))$$

and the cost of external sorting the map output is:

$$\begin{aligned} C_{sort-map-R} =& m_T \times 2c_L \times \\ & ((S_{MO-R} \times log_B(S_{MO-R}/(B+1))) \\ C_{sort-map-C} =& m_C \times 2c_L \times \\ & ((S_{MO-C} \times log_B(S_{MO-C}/(B+1))) \end{aligned}$$

The data on all the mappers are shuffled to the reducers after the mapper completes. The cost of shuffling is:

$$\begin{aligned} C_{shuffling} =& sh_{MR} \times s(Q) \times \\ & ((2 \times |T| \times s(T) + |C|) \times (d(Q) + n(Q)) \end{aligned}$$

In the reduce phase, the cost of sort merging process is:

$$\begin{aligned} C_{reduce-merge} =& 2c_L \times s(Q) \times \\ & (2 \times |T| \times s(T) + |C|) \times (d(Q) + n(Q)) \end{aligned}$$

and the cost of writing the data into HDFS is:

$$C_{reduce-write} = w_{HBase} \times |Q| \times (d(Q) + n(Q))$$

Based on the cost model discussed above, the more efficient approach is dynamically selected when a real-time query is submitted.

VI. Evaluation

In this section, we evaluate the R-Store on our in-house cluster of 144 nodes. Each node is equipped with Intel X3430 2.4 GHz processor, 8 GB of memory, 2x500 GB SATA disks, each of which is connected by a gigabit Ethernet and running CentOS 5.5. The cluster nodes are evenly placed onto three racks. We adopt TPC-H data for the experiments. However, TPC-H updates only append new keys, while we need online transactions that update the existing keys. Therefore, we write our own scripts to update the TPC-H data. The scripts can update the keys based on either a uniform distribution or Zipf distribution. In most of the following experiments, we use the TPC-H *part* table to build the data cube.

A. Performance of Maintaining Data Cube

In this experiment, we first measure the throughput of our real-time data cube maintenance algorithm to ensure that it has sufficiently high processing capacity to handle the update streams from HBase-R. As can be seen in Figure 4, when HStreaming is configured with 10 nodes, the algorithm can process more than 100K updates per second, which is even higher than the throughput of HBase-R with 40 nodes (the throughput of HBase-R will be discussed in Section VI-C).

We compare the two methods for refreshing the data cube: re-computation and incremental update. We deploy the system on 100 nodes, with 40 nodes for MapReduce, 40 nodes for HBase-R, and 20 nodes for HStreaming. The scale factor of the TPC-H data is set to 8000, so that there are 1,600,000,000 keys for *part* table. On each HBase-R node, there are 4.8 GB data. The data cube is built after the $part$ table is loaded into HBase-R.

Figure 5 shows the processing time of the two methods. The distribution of updated keys follows a Zipf distribution. We adjust the factor of the Zipf distribution so that about 1% keys are updated, while the number of updates is increased from 8 million to 1,600 million. Since HBase-R does not remove the previous version of the data, 0.024 GB to 4.8 GB of new data are inserted into each HBase-R node. The processing time of re-computation has two parts: the blue rectangle (ReCompScan) is the scan time of the real-time table, and the yellow rectangle (ReCompExe) is the execution time of the MapReduce job after the scan phase. As the number of updates increases, the data stored on each HBase-R node increases as well. Thus, more data are scanned at the HBase-R side for the re-computation approach, and the running time of the scan phase for re-computation is increased over time. However, as illustrated in Figure 5, the running time of the ReCompExe decreases as the number of updates increases, which is counterintuitive. We expected that the execution time of the MapReduce job should remain the same in different settings as they process the same number of key/value pairs. The reason for the decrease in ReCompExe is that ReCompScan and ReCompExe are pipelined. The more time ReCompScan takes, the more these two phases overlap, reducing the time ReCompScan takes.

In contrast, the processing time of incremental update consists of only one part (the red rectangle): the time it takes to write data cube into HBase-R. This is because our real-time data cube maintenance algorithm is fast enough to update the

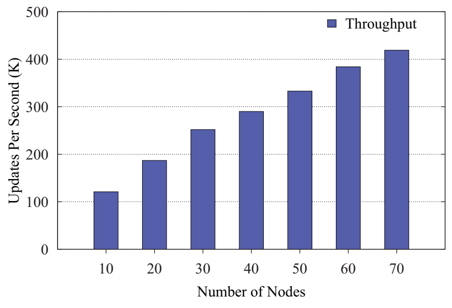


Fig. 4. Throughput of Real-Time Data Cube Maintenance Algorithm

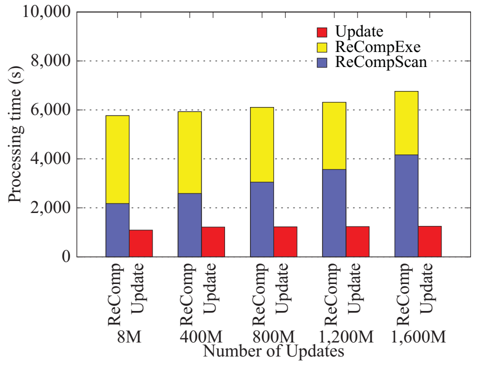


Fig. 5. Performance of Data Cube Refresh

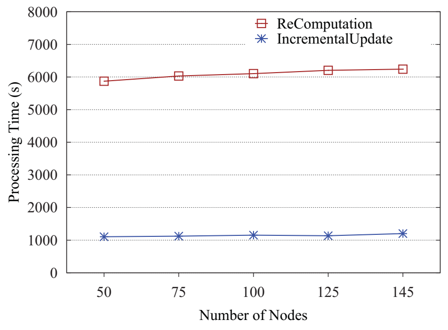


Fig. 6. Scalability

real-time data cube with the data streams from HBase-R. Thus the latency of periodically refreshing the data cube in HBase-R equals to the time of writing the real-time data cube into HBase-R. This time is related to the the size of the data cube and does not change as the number of updates increases.

We also evaluate the scalability of R-Store. In this experiment, the number of nodes and the data size increase with the same ratio. The percentage of updates is set to 1% for different scalability settings. As can be seen in Figure 6, the running time of both re-computation (the brown line) and incremental update (blue line) do not change much as the number of nodes increase, which demonstrates the scalability of R-Store.

B. *Performance of Real-Time Querying*

In this experiment, we investigate the performance of real-time querying. First, we compare the *IncreQuerying* algorithm, which optimizes the real-time query using the data cube, with the *Baseline* algorithm implemented with the `FullScan` operation. The cluster settings are the same as those of Figure 5, except that we fix the number of updates to 8,000 million and vary the percentage of the keys updated.

Figure 7(a) shows the processing time of both algorithms for a typical data cube slice query:

$$\begin{aligned}&SELECT\ sum(prices)\ FROM\ part\\&WHERE\ mft\ =\ \text{“}Manufacture\#1''\\&GROUPBY\ brand, type, size, container\end{aligned}$$

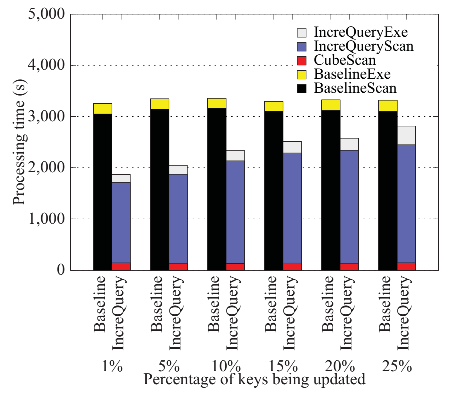
The processing time of the *Baseline* algorithm consists of two parts: the black rectangle (ReCompScan) is the time to scan the real-time table, and the yellow rectangle (ReCompExe) is the execution time of the MapReduce job after the scan phase. In contrast, the processing time of *IncreQuerying* consists of three parts: the red rectangle (CubeScan) is the time to scan the data cube, the blue rectangle (UpdateScan) is the time to scan the $part$ table in HBase-R, and the grey rectangle (UpdateExe) is the execution time of the MapReduce job after the scan phase.

When only a small range of keys are updated, *IncreQuerying* performs much better than *Baseline*. It outperforms the *Baseline* approach for two reasons: (1) by using adaptive incremental scan, it scans fewer data in HBase-R and shuffles fewer data to MapReduce; (2) its MapReduce job processes fewer data than that of re-computation. However, as the percentage of updated keys increases, more data are shuffled from HBase-R to MapReduce. Thus, both the scan time and the execution time increase. In contrast, for *Baseline*, since the `FullScan` always shuffles one version for each key to MapReduce, the amount of data shuffled from HBase-R is constant. As a result, the running time of *Baseline* is almost constant. Due to the existence of the filtering condition on attribute *mft*, most tuples of the table are filtered, and fewer data are sorted and shuffled during the execution of the MapReduce job. As a result, the difference between the execution times is not so significant. In general, *IncreQuerying* algorithm outperforms *Baseline* algorithm when the percentage of keys being updated is low.

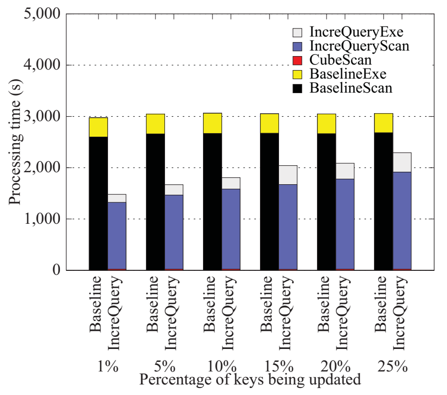
In addition to the data cube slice query, we also evaluate TPC-H Q1 (Figure 7(b)) with the same experimental settings. We did not illustrate other benchmark queries as they involve multiple tables, which will not be able to illustrate as clearly the effectiveness of the basic operators supported in R-Store. The parameter *shipdate* of TPC-H Q1 is set to “365” days, and only about 15% of the tuples are filtered ($shipdate$ larger than “12-01-1998”). Thus, the execution time of the MapReduce job after the scan phase is longer than that of Figure 7(a). For the *Lineitem* table, since we only build the data cube on attributes *shipdate*, *returnflag* and *linestatus*, the data cube is much smaller than the real-time table, and the time to scan the data cube is around 20 seconds. Overall, Figure 7(b) demonstrates that the performance of *IncreQuerying* is significantly better than that of *Baseline*.

To select the better querying method among the two, we use the cost model (Section V-C) to estimate the number of I/Os. Figure 8 shows the running time of *IncreQuerying*, and the I/Os estimated for both *Baseline* and *IncreQuerying* algorithms. The *y*-axis on the left is the processing time of the query, while the *y*-axis on the right is the estimated I/Os. The estimated number of I/Os for *IncreQuerying* (the blue line) increases linearly with almost the same slope (the histogram) as the processing time of the query, while the estimated number of I/Os for the Baseline (the brown line) is constant, which is around 2.52×10^{11}. This result hence verifies the accuracy of our cost model.

Compared to querying only the data cube, RTOLAP queries require two additional steps, which incur additional cost: scanning the real-time data from HBase-R, and merging the real-time data with the data cube on demand in MapReduce.



(a) Data Cube Slice Query



(b) TPC-H Q1

Fig. 7. Performance of Querying

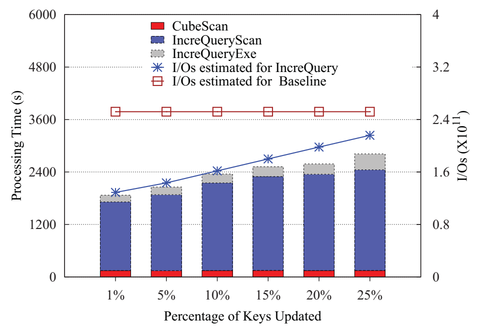


Fig. 8. Accuracy of Cost Model

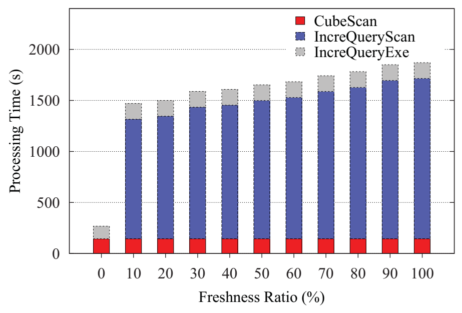


Fig. 9. Performance vs. Freshness

On each HBase-R node, the key/values are stored in *storefile* format. Though only one or two versions of the same key are returned to MapReduce, HBase-R has to scan all the *storefiles* of the *part* table.

Since the *memstore* is materialized to HDFS when it is full, these files are sorted by time. Thus, instead of scanning all the *storefiles* and *memstore* between T_{DC} and T_Q, only the *storefiles* between T_{DC} and a user specified timestamp T_i ($T_i < T_Q$) are scanned. The value of T_i decides the freshness of the result. There is a trade-off between the performance of the query and the freshness of the result: the smaller T_i is, the fewer real-time data are scanned. Figure 9 shows the query processing time with different freshness ratios, which is defined as the percentage of the real-time data we have to scan for the query. In this experiment, $|part|$ = 1600 million, and 800 million updates on 1% distinct keys are submitted to HBase-R. When the freshness ratio is 0, the input of the query is only the data cube. Thus, the cost of scanning the real-time data is 0. When the freshness ratio increases to 10%, the cost of scanning the real-time data is around 1500 seconds because the cost of scanning the real-time table dominates the OLAP query. As the freshness ratio increases, the running time of *IncreQuerying* method increases slightly, which is due to two reasons: (1) the data before T_{DC} still need to be scanned; and (2) the amount of data shuffled to mappers are roughly the same with different ratios.

Figure 10 depicts the effectiveness of our compaction scheme. In this experiment, we measure the processing time of the data cube slice query when the compaction scheme is applied (*Baseline* and *IncreQuerying*) and when it is not (*Baseline-NC* and *IncreQuerying-NC*). We submit 800 million updates to the server each day, and the percentage of keys updated is fixed to 1%. The data cube is refreshed at the beginning of each day, and the OLAP query is submitted to the server at the end of the day. Since the data are compacted after the data cube refresh, the amount of data stored in the real-time table are almost the same at the same time of each day. The processing time of *Baseline* and *IncreQuerying* are thus almost constant. In contrast, when the compaction scheme is turned off, HBase-R stores much more data, and the cost of locally scanning these data becomes larger than the cost of shuffling the data to MapReduce. As a result, the processing time of *Baseline-NC* and *IncreQuerying-NC* increases over time.

C. Performance of OLTP

In this experiment, we investigate the performance of OLTP queries when OLAP queries are running. The workload is update-only, and the keys being updated are uniformly distributed. We launch ten clients to concurrently submit the updates when the system is deployed on 100 nodes. Each client starts ten threads, each of which submits one million updates (100 updates in batch). Another client is launched to submit the data cube slice query. That is, one OLAP query and approximately 50,000 updates are concurrently processed in R-Store. The system reaches its maximum usage in this setting based on our observation. When the system is deployed on other number of nodes, the number of clients submitting updates is adjusted accordingly.

Figure 11(a) shows the throughput of the system. The throughput increases as the number of nodes increases, which demonstrates the scalability of the system. However, when OLAP queries are running, the update performance is lower than running only OLTP queries. This result is expected, because the OLAP queries compete for resources with the OLTP queries. We also evaluate the latency of updates when the system is approximately fully used. As shown in Figure 11(b), the aggregated response time for 1000 updates are similar with respect to varying scales.

VII. Conclusion

MapReduce is a parallel execution framework, which has been widely adopted due to its scalability and suitability in

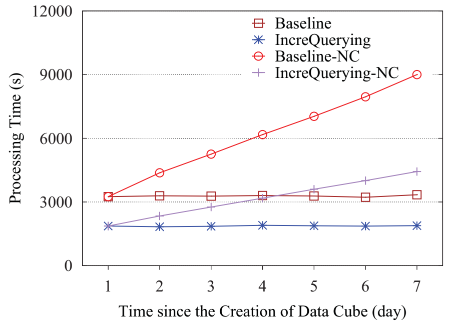
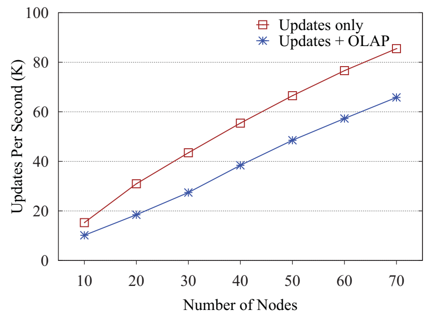
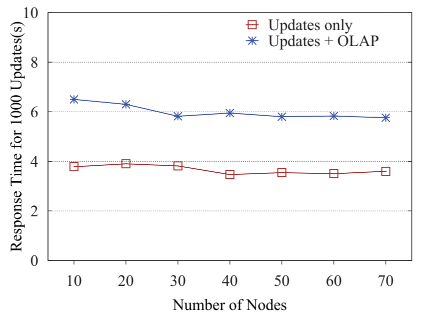


Fig. 10. Effectiveness of Compaction



(a) Throughput



(b) Latency

Fig. 11. Performance of OLTP Queries

a large scale distributed environment. However, most existing works only focus on optimizing the OLAP queries and assume that the data scanned by MapReduce are unchanged during the execution of a MapReduce job. In reality, the real-time results from the most recently updated data are more meaningful for decision making. In this paper, we propose R-Store for supporting real-time OLAP on MapReduce. R-Store leverages stable technology (HBase and HStreaming) and extends them to achieve high performance and scalability. The storage system of R-Store adopts multi-version concurrency control to support real-time OLAP. To reduce the storage requirement, it periodically materializes the real-time data into a data cube and compacts the historical versions into one version. During query processing, the proposed adaptive incremental scan operation shuffles the real-time data to MapReduce efficiently. The data cube and the newly updated data are combined in MapReduce to return the real-time results. In addition, based on our proposed cost model, the more efficient query processing method is selected. To evaluate the performance of R-Store, we have conducted extensive experimental study using the TPC-H data. The experimental results show that our system can support real-time OLAP queries much more efficiently than the baseline methods. Though the performance of OLTP degrades slightly due to the competition for resources with OLAP, the response time and throughput remain good and acceptable.

Acknowledgment

The work described in this paper was in part supported by the Singapore Ministry of Education Grant No. R-252-000-454-112 under the epiC project, and M.T.Özsu's work was partially supported by Natural Sciences and Engineering Research Council (NSERC) of Canada. We would also like to thank the anonymous reviewers for their insightful comments.

References

[1] http://hbase.apache.org/.

[2] http://hstreaming.com/.

[3] http://www.comp.nus.edu.sg/∼epic/.

[4] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. Masm: efficient online updates in data warehouses. In *SIGMOD*, pages 865–876, 2011.

[5] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. Es2: A cloud data storage system for supporting both oltp and olap. ICDE, pages 291–302, 2011.

[6] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, pages 577–589, 1991.

[7] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, pages 313–328, 2010.

[8] J. Dean, S. Ghemawat, and G. Inc. Mapreduce: simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[9] L. Golab, T. Johnson, and V. Shkapenyuk. Scheduling updates in a real-time stream warehouse. ICDE, pages 1207–1210, 2009.

[10] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. Hyrise: a main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, Nov. 2010.

[11] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally (extended abstract). In *SIGMOD*, pages 157–166, 1993.

[12] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *SIGMOD*, pages 543–554, 2010.

[13] D. Jiang, G. Chen, B. C. Ooi, and K.-L. Tan. epic: an extensible and scalable system for processing big data. 2014.

[14] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: an in-depth study. *Proc. VLDB Endow.*, 3(1-2):472–483, Sept. 2010.

[15] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. PODS '10, pages 41–52.

[16] A. Kemper, T. Neumann, F. F. Informatik, T. U. Mnchen, and D-Garching. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *In ICDE*, 2011.

[17] T.-W. Kuo, Y.-T. Kao, and C.-F. Kuo. Two-version based concurrency control and recovery in real-time client/server databases. *IEEE Trans. Comput.*, 52(4):506–524, Apr. 2003.

[18] K. Y. Lee and M. H. Kim. Efficient incremental maintenance of data cubes. In *VLDB*, pages 823–833, 2006.

[19] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using mapreduce. In *ACM Computing Survey*, 2014.

[20] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD*, pages 100–111, 1997.

[21] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan. Distributed cube materialization on holistic measures. In *ICDE*, pages 183–194, 2011.

[22] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, pages 170–177, 2010.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[24] K. Sergey and K. Yury. Applying map-reduce paradigm for parallel closed cube computation. In *DBKDA*, pages 62–67, 2009.

[25] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[26] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[27] P. Vassiliadis and A. Simitsis. Near real time ETL. In *Annals of Information Systems*, volume 3, pages 1–31. 2009.

[28] C. White. Intelligent business strategies: Real-time data warehousing heats up. *DM Review*, 2012.