MAP-JOIN-REDUCE: Towards Scalable and Efficient Data Analysis on Large Clusters

Dawei Jiang, Anthony K. H. TUNG, and Gang Chen

Abstract—Data analysis is an important functionality in cloud computing which allows a huge amount of data to be processed over very large clusters. MapReduce is recognized as a popular way to handle data in the cloud environment due to its excellent scalability and good fault tolerance. However, compared to parallel databases, the performance of MapReduce is slower when it is adopted to perform complex data analysis tasks that require the joining of multiple datasets in order to compute certain aggregates. A common concern is whether MapReduce can be improved to produce a system with both scalability and efficiency. In this paper, we introduce Map-Join-Reduce, a system that extends and improves MapReduce runtime framework to efficiently process complex data analysis tasks on large clusters. We first propose a filtering-join-aggregation programming model, a natural extension of MapReduce's filtering-aggregation programming model. Then, we present a new data processing strategy which performs filtering-join-aggregation tasks in two successive MapReduce jobs. The first job applies filtering logic to all the datasets in parallel, joins the qualified tuples, and pushes the join results to the reducers for partial aggregation. The second job combines all partial aggregation results and produces the final answer. The advantage of our approach is that we join multiple datasets in one go and thus avoid frequent checkpointing and shuffling of intermediate results, a major performance bottleneck in most of the current MapReduce based systems.

We benchmark our system against Hive, a state-of-the-art MapReduce based data warehouse on a 100-node cluster on Amazon EC2 using TPC-H benchmark. The results show that our approach significantly boosts the performance of complex analysis queries.

Index Terms—Cloud Computing, Parallel systems, Query processing.

1 Introduction

Cloud computing is a service through which a service provider delivers elastic computing resources (virtual compute nodes) to a number of users. This computing paradigm is attracting increasing interests since it enables users to scale their applications up and down seamlessly in a pay-as-you-go manner. To unleash the full power of cloud computing, it is well accepted that a cloud data processing system should provide a high degree of elasticity, scalability and fault tolerance.

MapReduce [1] is recognized as a possible means to perform elastic data processing in the cloud. There are three main reasons for this. First, the programming model of MapReduce is simple yet expressive. A large number of data analytical tasks can be expressed as a set of MapReduce jobs, including SQL query, data mining, machine learning and graph processing. Second, MapReduce achieves the desired elastic scalability through block-level scheduling and is proven to be highly scalable. Yahoo! has deployed MapReduce on a 4,000-node cluster [2]. Finally, MapReduce provides fine-grained fault tolerance whereby only tasks on failed nodes have to be restarted.

- *Dawei Jiang and Anthony K. H. Tung are with School of Computing, National University of Singapore, Singapore 117417.*
- *Gang Chen is with Computer Science College, Zhejiang University, Hangzhou 310027, China*

With the above features, MapReduce has become a popular tool for processing large-scale data analytical tasks. However, there are two problems when MapReduce is adopted for processing complex data analysis tasks which join multiple datasets for aggregation. First, MapReduce is mainly designed for performing a *filtering-aggregation* data analytical task on a single homogenous dataset [16]. It is not very convenient to express the join processing in the `map()` and `reduce()` functions [4].

Second, in certain cases, performing multi-way join using MapReduce is not efficient. The performance issue is mainly due to the fact that MapReduce employs a sequential data processing strategy which frequently checkpoints and shuffles intermediate results in data processing. Suppose we join three datasets, i.e., $R \bowtie S \bowtie T$, and conduct an aggregation on the join results. Most MapReduce based systems (e.g., Hive, Pig) will translate this query into four MapReduce jobs. The first job joins R and S, and writes the results U into a file system (e.g., Hadoop Distributed File System, HDFS). The second job joins U and T and produces V which will again be written to HDFS. The third job aggregates tuples on V. If more than one reducers are used in step three, a final job merges results from reducers of the third job, and writes the final query results into one HDFS file. Here, checkpointing U and V to HDFS, and shuffling them in the next MapReduce jobs incurs huge cost if U and V are large. Although one can achieve better performance

by allocating more nodes from the cloud, this "renting more nodes" solution is not really cost efficient in a pay-as-you-go environment like cloud. An ideal cloud data processing system should offer elastic data processing in the most *economical* way

This paper introduces Map-Join-Reduce, an extended and enhanced MapReduce system for simplifying and efficiently processing complex data analysis tasks. To solve the first problem described above, we introduce a filtering-join-aggregation programming model which is an extension of MapReduce's filtering-aggregation programming model. In addition to the mapper and reducer, we introduce a third operation join (called joiner) to the framework. To join multiple datasets for aggregation, users specify a set of `join()` functions and the join order. The runtime system automatically joins multiple datasets according to the join order and invoke `join()` functions to process the joined records. Like MapReduce, a Map-Join-Reduce job can be chained with an arbitrary number of MapReduce or Map-Join-Reduce jobs to form a complex data processing flow. Therefore, Map-Join-Reduce benefits both end users and high-level query engines built on top of MapReduce. For end users, Map-Join-Reduce removes the burden of presenting complex join algorithms to the system. For MapReduce based high-level query engines such as Hive [5] and Pig [6], Map-Join-Reduce provides a new building block for generating query plans.

To solve the second problem, we introduce a one-to-many shuffling strategy in Map-Join-Reduce. MapReduce adopts a one-to-one shuffling scheme which shuffles each intermediate key/value pair produced by a `map()` function to a unique reducer. In addition to this shuffling scheme, Map-Join-Reduce offers a one-to-many shuffling scheme which shuffles each intermediate key/value pair to many joiners at one time. We show that, with proper partition strategy, one can utilize the one-to-many shuffling scheme to join multiple datasets in one phase instead of a set of MapReduce jobs. This one-phase joining approach, in certain cases, is more efficient than the multi-phases joining approach employed by MapReduce in that it avoids checkpointing and shuffling intermediate join results in the next MapReduce jobs.

This paper makes the following contributions.

- We propose filtering-join-aggregation, a natural extension of MapReduce's filtering-aggregation programming model. This extended programming model covers more complex data analytical tasks which require to join multiple datasets for aggregation. The complexity of parallel join processing is handled by the runtime system, thus it is straightforward for both human and high-level query planner to generate data analytical programs.
- We introduce a one-to-many shuffling strategy and demonstrate the usage of such a shuffling strategy to perform filtering-join-aggregation data analytical tasks. This data processing scheme outperforms MapReduce's sequential data processing scheme since it avoids frequent checkpointing and shuffling intermediate results.
- We implement the proposed approach on Hadoop. We show that our technique is ready to be adopted. Although our solution is intrusive to Hadoop, our implementation is such that our system is binary compatible with Hadoop. Existing MapReduce programs can run directly on our system without modifications. This design makes it very easy for users to gradually migrate their legacy MapReduce programs to Map-Join-Reduce programs for better performance.
- We provide comprehensive performance study of our system. We benchmark our system against Hive using four TPC-H queries. The results show that the performance gap between our system and Hive grows as more joins are involved and more intermediate results are produced. For TPC-H Q9, our system runs thrice faster than Hive.

The rest of this paper is organized as follows: Section 2 presents the filter-join-aggregation model and describes MJR at high level. Section 3 discusses the implementation details on Hadoop. Section 4 presents optimization techniques. Section 5 reports experimental results. Section 6 reviews related work. We present our conclusions in Section 7.

2 MAP-JOIN-REDUCE

This section presents filtering-join-aggregation, a natural extension of MapReduce's filtering-aggregation programming model and describes the overall data processing flow in Map-Join-Reduce.

2.1 Filtering-Join-Aggregation

As described before, MapReduce represents a two-phase filtering-aggregation data analysis framework with mappers performing filtering logic and reducers performing aggregation logic [16]. In [1], the signatures of map and reduce functions are defined as follows:

```
map     (k1,v1)          →  list(k2,v2)
reduce  (k2,list(v2))    →  list(v2)
```

This programming model is mainly designed for homogeneous datasets, namely the same filtering logic, represented by the `map()` function, is applied to each tuple in the dataset. We extend this model to filtering-join-aggregation in order to process multiple heterogenous datasets. In addition to `map()` and `reduce()` functions, we introduce a third `join()` function i.e., joiner. A filtering-join-aggregation data analytical task involves n dataset and $D_i, i \in \{1, \ldots, n\}$, $n-1$ join functions. The

signatures of map, join and reduce functions are as follows:

map_i (k1$_i$,v1$_i$) → (k2$_i$,list(v2$_i$))
join_j ((k2$_{j-1}$,list(v2$_{j-1}$)),(k2$_j$,list(v2$_j$))) → (k2$_{j+1}$,list(v2$_{j+1}$))
reduce (k2,list(v2)) → list(v2)

The signature of map in Map-Join-Reduce is similar to that of MapReduce except for the subscript i which denotes that the filtering logic defined by map_i is applied on dataset D_i. The join function $\text{join}_j, j \in \{1, \ldots, n-1\}$ defines the logic for processing the j-th joined tuples. If $j = 1$, the first input list of join_j comes from the mappers output. If $j > 1$ the first input list is from the $(j-1)$-th join results. The second input list of join_j must be from mapper output. From a database perspective, the join chain of Map-Join-Reduce is equivalent to a left-deep tree. Currently, we only support equal join. For each function join_j, the runtime system guarantees that the key of the first input list is equal to the key in the second input list, namely $k2_{j-1} = k2_j$. The reduce function's signature is the same as MapReduce, we shall not explain further. A Map-Join-Reduce job can be chained with an arbitrary number of MapReduce or Map-Join-Reduce jobs to form complex data processing flow by feeding its output to the next MapReduce or Map-Join-Reduce job. This chaining strategy is a standard technique in MapReduce based data processing systems [1]. Therefore, in this paper, we only focus on presenting the execution flow of a single Map-Join-Reduce task.

2.2 Example

We give a concrete example of the filtering-join-aggregation task here. The data analytical task in the example is a simplified TPC-H Q3 query [19]. This will be our running example for illustrating the features of Map-Join-Reduce. The TPC-H Q3 task, represented in SQL, is as follows:

```
select
  O.orderdate, sum(L.extendedprice)
from
  customer C, orders O, lineitem L
where
  C.mksegment='BUILDING' and
  C.custkey = O.custkey and
  L.orderkey = O.orderkey and
  O.orderdate < date '1995-03-15' and
  L.shipdate > date '1995-03-15'
group by
  O.orderdate
```

This data analytical task requires the system to apply filtering condition on all three datasets, i.e., `customer`, `orders`, and `lineitem`, join them and calculate the corresponding aggregations. Schemas of the datasets can be found in [19]. We intentionally omit them for saving space. The Map-Join-Reduce program that performs this analytical task is similar to the following pseudo-code:

```
map_C(long tid, Tuple t):
  // tid: tuple ID
  // t: tuple in customer
  if t.mksegment = 'BUILDING'
    emit(t.custkey, null)

map_O(long tid, Tuple t):
  if t.orderdate < date '1995-03-15'
    emit(t.custkey, (t.orderkey, t.orderdate))

map_L(long tid, Tuple t):
  if t.shipdate > date '1995-03-15'
    emit(t.orderkey, (t.extendedprice))

join_1(long lKey, Iterator lValues,
     long rKey, Iterator rValues):
  for each V in rValues
    emit(V.orderkey, (V.orderdate))

join_2(long lKey, Iterator lValues,
     long rKey, Iterator rValues):
  for each V1 in lValues
    for each V2 in rValues
      emit(V1.orderdate, (V2.extendedprice))

reduce(Date d, Iterator values):
  double price = 0.0
  for each V in values
    price += V
  emit(d, price)
```

To launch a Map-Join-Reduce job, in addition to the above pseudo-code, one also needs to specify a join order which defines the execution order of joiners. This is achieved by providing a Map-Join-Reduce job specification, an extension of original MapReduce's job specification, to the runtime system. Details of providing job specification can be found in Section 3.1. Here, we only focus on presenting the logic of `map()`, `join()` and `reduce()` functions.

To evaluate TPC-H Q3, three mappers, map_C, map_O, and map_L, are specified to process records in `customer`, `orders`, and `lineitem` respectively. The first joiner join_1 processes the results of $C \bowtie O$, i.e., `customer` and `orders`. For each joined record pair, it produces a key/value pair with the `orderkey` as the key and the `(orderdate)` as the value. The result pair are then passed to the second joiner. The second joiner joins the result tuple of join_1 with `lineitem` and emits `orderdate` as key and `(extendedprice)` as value. Finally, the reducer aggregates `extendedprice` on each possible date.

2.3 Execution Overview

To execute a Map-Join-Reduce job, the runtime system launches two kinds of processes: called *MapTask*, and *ReduceTask*. Mappers run inside the MapTask process while joiners and reducers are invoked inside the ReduceTask process. The MapTask process and ReduceTask process are semantically equivalent to map worker process and reduce worker process presented in [1]. Map-Join-Reduce's process model allows for pipelining intermediate results between joiners and reducers since joiners and reducers are run inside the

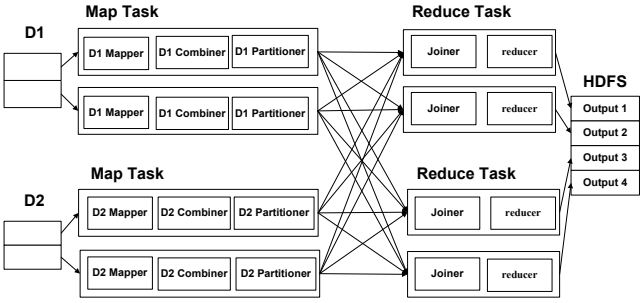


Fig. 1. Execution Flow of First MapReduce Job

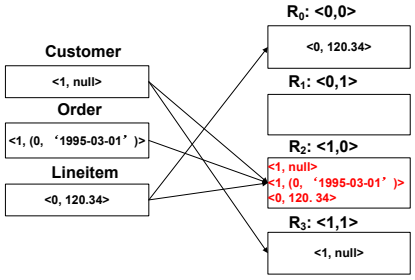


Fig. 2. Process of partition customer, orders, and lineitem

same ReduceTask process. The failure recovery strategy of Map-Join-Reduce is identical to MapReduce. In the presence of node failure, only MapTasks and uncompleted ReduceTasks need to be restarted. Completed ReduceTasks do not need to be re-executed. The process of task restarting in Map-Join-Reduce is also similar to MapReduce except for ReduceTask. In addition to re-run the `reduce()` function, when a ReduceTask is restarted, all the joiners are also re-executed.

Map-Join-Reduce is compatible with MapReduce. Therefore, a filtering-join-aggregation task can be evaluated by the standard sequential data processing strategy described in Section 1. In this case, for each MapReduce job, the ReduceTask process only invokes a unique `join()` to process an intermediate two-way join results. We shall omit the details of this data processing scheme. Alteratively, Map-Join-Reduce can also perform a filtering-join-aggregation task by two successive MapReduce jobs. The first job performs filtering, join and partial aggregation. The second job combines the partial aggregation results and writes the final aggregation results to HDFS [1].

In the first MapReduce job, the runtime system splits the input datasets into chunks in a per-dataset manner and then launches a set of MapTasks onto those chunks with one being allocated to each chunk. Each MapTask executes a corresponding map function to filter tuples, and emits intermediate key-value pairs. The output is then forwarded to the combiner, if a map-side partial aggregation is necessary, and to the partitioner in turn. The partitioner applies a user specified partitioning function on each map output and creates corresponding partitions for a set of reducers. We will see how Map-Join-Reduce partition the same intermediate pair to many reducers. For now, we simply state that the partition is to ensure that each reducer can independently perform all joins on the intermediate results that it receives. Details of partitioning will be presented later. Finally, the intermediate pairs are sorted by the key and then written to local disks.

When the MapTasks are completed, the runtime system launches a set of ReduceTasks. Each reducer builds a join list data structure which links all joiners as the user specified join order. Then, each ReduceTasks remotely reads (shuffles) partitions associated to it from all mappers. When a partition is successfully read, the ReduceTask checks whether the first joiner is ready to perform. A joiner is ready if and only if both its first and second input datasets are ready, either in memory or on local disk. When the joiner is ready, ReduceTask performs a merge-join algorithm on its input datasets and fires its join function on the joined results. ReduceTask buffers the output of the joiner in memory. If the memory buffer is full, it sorts the results and writes the sorted results to disk. ReduceTask repeats the whole loop until all the joiners are completed. Here, the shuffling and join operations overlap with each other. The output of the final joiner is then fed to the reducer for partial aggregation. Figure 1 depicts the execution flow of the first MapReduce job. In Figure 1, the datasets D_1 and D_2 are chopped into two chunks. For each chunk, a mapper is launched for filtering qualified tuples. The output of all mappers are then shuffled to joiners for join. Finally, the output of the final joiner is passed to reducer for partial aggregation.

When the first job is completed, the second MapReduce is launched to combine the partial results (typically via applying the same reduce function on the results) and present the final aggregation results to HDFS. The second job is a standard MapReduce job, and thus, we omit its execution details.

2.4 Partitioning

Obviously, to make the framework described above work, the important step is to properly partition the output of mappers so that each reducer can join all datasets locally.

This problem is fairly easy to solve if the analytical task only involves two datasets. Consider we join two datasets, $R \overset{R.a=S.b}{\bowtie} S$. To partition R and S to n_r reducers, we adopt a partition function $H(x) = h(x) \bmod n_r$, where $h(x)$ is a universal hash function to each tuple in R and S on the join column, and take the output of $H(x)$ as the partition signature

1. Strictly speaking, if only one reducer is used in the first job, the second merge job is unnecessary. However, in real-world workload, a number of reducers are required in the first job to speed up data processing. Therefore, the second job is needed to produce the final query results.

that is associated to a unique reducer for processing. Therefore, tuples that can be joined with each other will eventually go to the same reducer. This technique is equivalent to a standard parallel hash join algorithm [20] and is widely used in current MapReduce based systems. The scheme is also feasible for multiple datasets (more than two) join if each dataset has a unique join column. As an example, to perform $R \overset{R.a=S.b}{\bowtie} S \overset{S.b=T.c}{\bowtie} T$, we can also apply the same partition function $H(x)$ on the join columns $R.a$, $S.b$, and $T.c$, and partition R, S and T to the same n_r reducers to complete all joins in one MapReduce job.

However, if a data analytical task involves a dataset that has more than one join column, the above technique will not work. For example, if we perform $R \overset{R.a=S.a}{\bowtie} S \overset{S.b=T.c}{\bowtie} T$, it is impossible to use a single partition function to partition all three datasets to the reducers in one pass. Map-Join-Reduce solves this problem by utilizing k partition functions to partition the input datasets where k is the number of connected components in the derived join graph of the query.

We will first give a concrete example; general rules for data partitioning will be provided later. Recall the previous simplified TPC-H Q3 query. The query performs `C` $\bowtie$ `O` $\bowtie$ `L` for aggregation where `C`, `O` and `L` stand for `customer`, `orders` and `lineitem` respectively. The join condition is `C.custkey` = `O.custkey` and `O.orderkey` = `L.orderkey`.

To partition the three input datasets into $n_r = 4$ reducers, we use two partition functions $< H_1(x), H_2(x) >$ with $H_1(x)$ to partition columns `C.custkey` and `O.custkey` and $H_2(x)$ to partition columns `O.orderkey` and `L.orderkey`. Function $H_1(x)$ is defined as $H_1(x) = h(x) \bmod n_1$. Function $H_2(x)$ is defined as $H_2(x) = h(x) \bmod n_2$. To facilitate discussion, we assume that the universal hash function $h(x)$ is $h(x) = x$. The point is that the partition number n_1 and n_2 must satisfy the constraint $n_1 \cdot n_2 = n_r$. Suppose we set $n_1 = 2$ and $n_2 = 2$. Each reducer is then associated with a unique partition signature pair among all possible outcomes. In this example, reducer R_0 is associated with $< 0, 0 >$, R_1 is associated with $< 0, 1 >$, R_2 is associated with $< 1, 0 >$, and R_3 is associated with $< 1, 1 >$.

Now, we use $< H_1(x), H_2(x) >$ to partition the datasets. We begin with `customer` relation. Suppose the input key-value pair t is $< 1, null >$ where 1 is the `custkey`. The partition signature of this `custkey` is calculated as $H_1(1) = 1$. Since `customer` has no column that belongs to the partition function $H_2(x)$, all possible outcome of $H_2(x)$ is considered. Therefore, t is partitioned to reducers $R_2 :< 1, 0 >$ and $R_3 :< 1, 1 >$.

The same logic is applied to relations `orders` and `lineitem`. Suppose the input pair of `orders` o is $< 1, (0, \text{'1995-03-01'}) >$, then o is partitioned to $R_2 :< H_1(1) = 1, H_2(0) = 0 >$. The input pair of `lineitem` $l :< 0, 120.34 >$ is partitioned to $R_0 :< 0, H_2(0) = 0 >$ and $R_2 :< 1, H_2(0) = 0 >$. Now, all three tuples can be joined in R_2. Figure 2 shows the whole partitioning process.

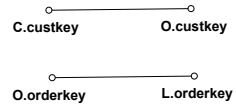


Fig. 3. Derived Join Graph for TPC-H Q3

The above algorithm is correct. Clearly, for any tuples from three datasets that can be joined, namely $\texttt{C}(x, null)$, $\texttt{O}(x, (y, date))$ and $\texttt{L}(y, price)$, these tuples will eventually be partitioned to the same reducer $R_i :< H_1(x), H_2(y) >$.

In general, to partition n datasets to n_r reducers for processing, we first build k partition functions:

$$\mathcal{H}(x) =< H_1(x), \ldots, H_k(x) >$$

Each partition function $H_i(x) = h(x) \bmod n_i, i \in \{1, \ldots, k\}$ is responsible for partitioning a set of join columns. We call the join columns that $H_i(x)$ operates on, the domain of $H_i(x)$ denoted by Dom$[H_i]$. The constraint is that parameters, i.e., n_i in all partition functions must satisfy $\Pi_{i=1}^{k} n_i = n_r$.

When k partition functions are built, the partitioning process is straightforward, as shown by the previous TPC-H Q3 example. First, we associate each reducer with a unique signature, a k-dimensional vector, from all possible partition outcomes. The reducer will then process intermediate mapper output that belongs to the assigned partition.

For each intermediate output pair, the k dimensional partition signature is calculated by applying all k partition functions in $\mathcal{H}$ on the pair in turn. For $H_i(x)$, if the intermediate result contains a join column c that falls in Dom$[H_i]$, the i-th value of the k dimensional signature is $H_i(c)$; otherwise all possible outcomes of $H_i(x)$ are considered.

The remaining questions of partitioning are: 1) how to build k partition functions for a query; 2) how to determine the domain for each partition function $H_i(x)$; and 3) given n_r, what are the optimal values for partition parameters $\{n_1, \ldots, n_k\}$. We solve the first two problems in this section. The optimization problems are discussed in Section 4.

We build a derived join graph for a data analytical task according to the following definition:

Definition 1: A graph G is called a derived join graph of task Q if each vertex in G is a unique join column involved in Q and each edge is a join condition that joins two datasets in Q.

Definition 2: A connected component G_c of a derived join graph is a subgraph in which any vertices are reachable by path.

Figure 3 shows the derived join graph of TPC-H Q3. We only support queries whose derived join graphs have no loops. Even with this restriction, we will see

this model covers many complex queries including those of TPC-H.

We build $\mathcal{H}(x)$ as follows: First, we enumerate all connected components in the derived join graph. Suppose k connected components are found, we build a partition function for each connected component. The domain of the partition function is the vertices (join columns) in the corresponding connected component. For example, the derived join graph of TPC-H Q3 has two connected components as shown in Figure 3. So, we build two partition functions $\mathcal{H}(x) =< H_1(x), H_2(x) >$. The domains of partition functions are vertices (join columns) in each connected component, namely:

$$\text{Dom}[H_1] = \{\texttt{C.custkey}, \texttt{O.custkey}\}$$

$$\text{Dom}[H_2] = \{\texttt{O.orderkey}, \texttt{L.orderkey}\}$$

Currently, we rely on users to manually build partition functions as we have done in the experiments. In the longer term, we plan to introduce an optimizer for automatically building derived join graph and partition functions.

2.5 Discussion

In Map-Join-Reduce, we only shuffle input datasets to reducers and do not checkpoint and shuffle intermediate join results. Instead, intermediate join results are pipelined between joiners and reducer either through in-memory buffer or local disk. In general, if join selectivity is low, a common case in real-world workload, it is less costly to shuffle input datasets than intermediate join results. Furthermore, shuffling and join operations overlap in the reducer side to speed up query processing. A joiner is launched immediately if both its input datasets are ready.

Although Map-Join-Reduce is designed for multi-way join, it can also be used together with existing two-way join techniques. Assume we want to perform $S \bowtie R \bowtie T \bowtie U$. If S is quite small and can be fitted into the memory of any single machine, we can join R with S in the map-side by loading S into the memory of each mapper launched on R as described in [14]. In this case, mappers of R emit joined tuples to reducers and perform a normal Map-Join-Reduce procedure to join with T and U for aggregation. Also, if R and S are already partitioned on join columns among available nodes, we can also use a map-side join, and shuffle the results to reducers for further processing. In the future, we will introduce a map-side joiner, which runs a joiner inside MapTask process, and transparently integrate all these two-way join techniques in Map-Join-Reduce framework.

The potential problem of Map-Join-Reduce is that it may consume more memory and local disk space to process a query. Compared to the sequential processing strategy that we described in Section 1, reducers in Map-Join-Reduce receive more portions of input datasets than reducers in MapReduce. In MapReduce, datasets are partitioned with the full number of reducers, namely each dataset will be partitioned into n_r portions, but in Map-Join-Reduce, the dataset may be partitioned into a small number of partitions. In the TPC-H Q3 example, `customer` and `lineitem` are both partitioned into two partitions although the total number of available reducers is 4. Therefore, compared to sequential query processing, the reducers of Map-Join-Reduce may need a larger memory buffer and more disk space to hold input datasets. One possible solution to solve this problem is to allocate more nodes from the cloud and utilize those nodes to process the data analytical task.

Even in environments with a limited number of compute nodes, we still have some ways to solve the problem. First, given a fixed number of reducers, we can use the technique presented in Section 4 to tune the partition number n_i of each partition function to minimize the input dataset portions that each reducer receives. Second, we can compress the output of mappers and operate the data in compressed format, a technique which is widely used in column-wise database systems to reduce disk and memory cost [21]. Finally, we can adopt a hybrid query processing strategy. For example, suppose we need to join six datasets but the available computing resources only allow us to join four datasets once, we can first launch a MapReduce job to join four datasets, write the results to HDFS and then launch another MapReduce job to join the rest of the datasets for aggregation. This hybrid processing strategy is equivalent to the ZigZag processing in parallel database [22].

3 Implementation on Hadoop

We implement Map-Join-Reduce on Hadoop 0.19.2, an open source MapReduce implementation. Although large-scale data analysis is an emerging application of MapReduce, not all MapReduce programs are of this kind; some of them build inverted-indexing, perform large-scale machine learning, and conduct many other data processing tasks. Therefore, it is important that our modifications do not damage the interface and semantics of MapReduce to the extent that those non-data analysis tasks fail to work. That is, we must make sure that the resulting system should be binary compatible with Hadoop and existing MapReduce jobs can run on our system without any problems. Fortunately, based on the experiences from building this implementation, such a requirement does not introduce huge engineering efforts. This section describes these implementation details.

3.1 New APIs

We introduce new APIs to Hadoop for new features of Map-Join-Reduce. Since MapReduce is mainly designed for processing a single homogeneous dataset,

the API that Hadoop provides only supports specifying one mapper, combiner and partitioner for each MapReduce job

In Map-Join-Reduce, the mapper, combiner and partitioner is defined in a per-dataset manner. Users specify a mapper, combiner and partitioner for each dataset using the following API:

```
TableInputs.addTableInput(path, mapper, combiner,
                          partitioner)
```

In the above code, `path` points to the location in the HDFS that stores the dataset while `mapper`, `combiner` and `partitioner` define the Java class that the user creates to process the dataset. The return value of `addTableInput()` is an integer which specifies the dataset id (called table id in Map-Join-Reduce). Table id is used to specify the join input datasets, and is used for the system to perform various per-dataset operations, e.g., launching corresponding mappers and combiners, which will be discussed in the following subsections.

Following Hadoop's principle, joiner is also implemented as a Java interface in Map-Join-Reduce. Users specify join processing logic by creating a joiner class and implementing `join()` functions. Joiners are registered to the system as follows:

```
TableInputs.addJoiner(leftId, rightId,
                      joiner)
```

The input datasets of joiner are specified by `leftId` and `rightId`. Both `Id`s are integers either returned by `addTableInput()` or `addJoiner()`. The return value of `addJoiner()` represents the resulting table id. Therefore, joiners can be chained. As stated previously, to simplify implementation, the right input of a joiner must be a source input, namely the dataset added by `addTableInput()`; only the left input can be results of a joiner. As an example, the specifications of TPC-H Q3 query is described as follows:

```
C = TableInputs.addTableInput(CPath, CMap,
                              CPartitioner)
O = TableInputs.addTableInput(OPath, OMap,
                              CPartitioner)
L = TableInputs.addTableInput(LPath, LMap,
                              CPartitioner)
tmp = TableInputs.addJoiner(C, O, Join1)
TableInputs.addJoiner(tmp, L, Join2)
```

3.2 Data Partitioning

Before the MapReduce job can be launched, datasets need to be partitioned into chunks. Each chunk is called a `FileSplit` in Hadoop. We implement a `TableInputFormat` class to split multiple datasets for launching Map-Join-Reduce jobs. `TableInputFormat` walks through the dataset list produced by `addTableInput()`. For each dataset, it adopts conventional Hadoop code to split files of the dataset into `FileSplit`s. When all the `FileSplit`s for the processing dataset are collected, `TableInputFormat` rewrites each `FileSplit` into a `TableSplit` by appending additional information including table id, mapper class, combiner class, and partitioner class. When `TableSplit`s of all datasets are generated, `TableInputFormat` sorts these splits first by access order, then by split size. This is to ensure that datasets that will be joined first will have a bigger chance to scan first [2].

3.3 MapTask

When MapTask is launched, it first reads the `TableSplit` that is assigned to it, then parses the information from the `TableSplit`, and launches mapper, combiner and partitioner to process the dataset. Overall, the workflow of MapTask is the same as the original MapReduce except for partitioning. There are two problems here. First, in Map-Join-Reduce, the partition signature of an intermediate pair is a k-dimensional vector. However, Hadoop can only shuffle an intermediate pair based on a single value partition signature. Second, Map-Join-Reduce requires shuffling the same intermediate pair to many reducers. However, Hadoop can only shuffle the same intermediate pair to only one reducer.

To solve the first problem, we convert our k-dimensional signature into a single value. Given the k-dimensional partition signature $\mathcal{S} =< x_1, \ldots, x_k >$ and the k partition functions parameters $\{n_1, \ldots, n_k\}$, the single signature value s is calculated as follows:

$$s = x_1 + \sum_{i=2}^{k} x_i \cdot n_{i-1}$$

For the second problem, a naive method involves writing the same intermediate pair to disk several times. Suppose we need to shuffle an intermediate key-value pair I to m reducers, we can emit I for m times in map functions. Each I_i is associated with a different partition vector. Hadoop then is able to shuffle each I_i to a unique reducer. Unfortunately, this method dumps I to disk m times and introduces a huge I/O overhead at map-side.

We adopt an alternative solution to rectify this problem. First we extend Hadoop's `Partitioner` interface to `TablePartitioner` interface and add a new function that can return a set of reducers ids as shuffling targets. The new function is as follows:

```
int[] getTablePartitions(K key, V value)
```

MapTask then collects and sorts all the intermediate pairs. The sorting groups pairs that will be shuffled to the same set of reducers into partitions based on the returned information of `getTablePartition()` function, and orders pairs in each partition according to keys. Using this approach, the same intermediate

2. In MapReduce, there is no method to control dataset access order. Here, we only give a hint to the Hadoop scheduler for scheduling MapTask to process datasets.

pair is only written to disk once and thus will not introduce additional I/Os.

3.4 ReduceTask

The structure of ReduceTask is also similar to the original Hadoop version. The only difference is that it holds an array of counters for a multiple datasets job, one for each dataset. For each dataset, the counter is initially set to the total number of mappers that the reducer needs to connect for shuffling data. When a partition is successfully read from a certain mapper, ReduceTask decreases the corresponding counter. When all the needed partitions for a dataset are read, ReduceTask checks the joiner at the front of the joiner list. If the joiner is ready, ReduceTask performs the merge join algorithm and calls the joiner's join function to process the results.

4 OPTIMIZATION

In addition to the modifications described in the previous section, three optimization strategies are also adopted. We present them here.

4.1 Speedup Parsing

Following MapReduce, Map-Join-Reduce is designed to be storage independent. As a result, users have to decode the record stored in the value part of the input key/value pair in `map()` and `reduce()` functions. Previous studies show that this runtime data decoding process introduces considerable overhead [12], [10], [11].

There are two kinds of decoding schemes: immutable decoding and mutable decoding. The immutable decoding scheme transforms raw data into immutable objects, i.e., read-only objects. Using this approach, decoding 4 millions records results in 4 millions immutable objects and thus introduces huge CPU overhead. We found that the poor performance of data parsing reported in previous studies is due to the fact that all these studies adopt the immutable scheme.

To reduce the record parsing problem, we adopt a mutable decoding scheme in Map-Join-Reduce. The idea is straightforward, to decode records from a dataset D, we create a mutable object according to the schema of D and use that object to decode all records belonging to D. Therefore, no matter how many records will be decoded, only one mutable object is created. Our benchmarking results show that the performance of mutable decoding outperforms immutable decoding by a factor of four.

4.2 Tuning Partition Functions

In Map-Join-Reduce, an intermediate pair could be shuffled to many reducers for join. To save network bandwidth and computation cost, it is important to ensure that each reducer receives only a minimal number of intermediate pairs to process. This section discusses this problem.

Suppose a filtering-join-aggregation task Q involves n datasets $D = \{D_1, \ldots, D_n\}$. The derived join graph includes k connected components and the corresponding partition functions are $\mathcal{H} = \{H_1(x), \ldots, H_k(x)\}$, with partition numbers $\{n_1, \ldots, n_k\}$. For each dataset D_i, m_i partition functions $\mathcal{H}_i = \{H_{m_1}(x), \ldots, H_{m_i}(x)\}$ are used to partition the join columns. Furthermore, we assume that there is no data distribution skew for the time being. Data skew is our future work. The optimization problem is to minimize the intermediate pairs that each reducer receives:

$$\begin{array}{ll} \text{minimize} & F(x) = \sum_{i=1}^{n} \frac{|D_i|}{\Pi_{j=1}^{m_i} n_j} \\ \text{subject to} & \Pi_{i=1}^{k} n_i = n_r \\ & n_i \geq 1 \text{ is an integer} \end{array}$$

In the above problem formulation, n_r is the number of ReduceTasks specified by the user. Like MapReduce, the number n_r is often set to a small multiple of the number of slave nodes [1]. The optimization problem is equivalent to a non-linear integer programming program. In general, non-linear integer programming is an NP-hard problem [23], and there is no efficient algorithm to solve it. However, in this case, if the number of reducers n_r is small, we can use a brute-force approach which enumerates all possible feasible solutions to minimize the object function $F(x)$. However, if n_r is large, say 10,000, finding optimal partition numbers for four partition functions requires $O(10^{16})$ computations, which makes the brute-force approach infeasible.

If n_r is large, we use a heuristic approach to solve the optimization problem and produce an approximation solution that works reasonably well. For a very large n_r, we first round it to a number $n'_r \leq n_r$ with $n'_r = 2^d$. Then we replace n_r with n'_r and rewrite the constraint as:

$$\Pi_{i=1}^{k} n_i = n'_r$$

It is easy to see that after constraint rewriting, each n_i must be of the form $n_i = 2^{j_i}$, where j_i is an integer. So, the constraint can further be written as:

$$\sum_{i=1}^{k} j_i = d$$

Now, the brute-force approach can be used to find optimal $j_i, i \in \{1, \ldots, k\}$ to minimize the object function. The computation cost is reduced to $O(d^k)$.

We now build a cost model and analyze the I/O costs of evaluating a filtering-join-aggregation task by: 1) a standard sequential data processing strategy employed by original MapReduce and 2) the alternative data processing strategy introduced by Map-Join-Reduce. We regard the whole cluster as a single

computer and estimate the total I/O costs for both approaches. The difference between the two approaches lies in the method of joining multiple datasets. The final aggregation step is the same. Therefore, we only consider the joining phase.

For the sequential data processing, the multi-way join is evaluated by a set of MapReduce jobs. The I/O cost C_s is the sum of I/O costs of all `map()` and `reduce()` functions, which is equivalent to scanning and shuffling the input datasets and intermediate join results:

$$C_m = 2\left(\sum_{i=1}^{n}|D_i| + \sum_{j=1}^{n-1}|J_j|\right)$$

where $|J_j|$ is the size of j-th join results. The coefficient is two since both the input datasets and intermediate results are first read from HDFS by mappers and then shuffled to reducers for processing. Thus, two I/Os are introduced.

For the one-phase join processing, the input datasets are first read by mappers and then replicated to multiple reducers for joining. Thus the total I/O cost C_p is

$$C_p = \sum_{i=1}^{n}|D_i| + \sum_{i=1}^{n}\prod^{H_j \notin \mathcal{H}_i}(n_j \cdot |D_i|)$$

Comparing C_s and C_p, it is obvious that if the intermediate join results is huge and thus the I/O cost of checkpointing and shuffling those intermediate results is higher than replicating input datasets on multiple reducers, one-phase join processing is more efficient than sequential data processing.

4.3 Speeding Up the Final Merge

In Map-Join-Reduce, in order to calculate the final aggregates, the second MaReduce job often needs to process a large number of small files. This is because the first MapReduce job launches a huge number of reducers for processing join and partial aggregation and produces a partial aggregation results file for each reducer.

Currently, Hadoop schedules mappers in per-file manner, one mapper for each file. If there are 400 files to process, there will be at least 400 mappers to launch. This per-file assignment scheme is quite inefficient for the second merging job. After join and partial aggregation, the partial results files produced by the first job are quite small, several KBs typically. However, we observe that the start-up cost of a mapper in a 100-node cluster is around 7∼10 seconds, thousands times larger than actual data processing time.

To speed up the final merging process, we adopt another scheduling strategy for the second MapReduce job. Instead of scheduling mappers in a per-file manner, we schedule a mapper to process multiple files in order to enlarge the payload. Particularly, we schedule a mapper to process 128MB data consolidated from multiple files. Using this approach, the number of mappers needed in the second job is significantly reduced. The typical merging time, based on our experiments on TPC-H queries, is around 15 seconds, which approaches the minimal cost of launching a MapReduce job.

5 Experiments

In this Section, we study the performance of Map-Join-Reduce. Our benchmark consists of five tasks. In the first task, we evaluate the performance of our tuple parsing technique and study whether our approach could reduce the CPU cost in runtime parsing. Then, we benchmark Map-Join-Reduce against Hive with four analytical tasks drawn from the TPC-H benchmark.

5.1 Benchmark Environment

We run all benchmarks on Amazon EC2 Cloud with large instances. Each instance has 7.5 GB memory, 4 EC2 Compute Units (2 virtual cores), 420 GB instance storage, and runs 64-bit platform Linux Fedora 8 OS. Interestingly, in the data sheet, Amazon claims that a large instance has 850GB instance storage (2 $\times$ 420 GB plus 10 GB root partition). However, when we login to the instance and check the system with `df -h`, we found out only one 420 GB disk was installed. The raw disk speed of a large instance is roughly 120MB/s, and the network bandwidth is about 100MB/s. For analysis tasks, we benchmark the performance with cluster sizes of 10, 50, and 100 nodes [3]. We implemented Map-Join-Reduce on Hadoop v0.19.2 and use the enhanced Hadoop to run all benchmarks. The Java system we used is 1.6.0_16.

5.1.1 Hive Settings

There are two important reasons that we choose Hive as the system to benchmark against. First, Hive represents state-of-the-art MapReduce based system that processes complex analytical workloads. Second, but more importantly, Hive has already benchmarked itself using the TPC-H benchmark and released its HiveQL, an SQL like query declaration language, scripts and Hadoop configurations [25]. This simplifies our effort in setting up Hive to run and tune the parameters for better performance. We assume that the HiveQL scripts that Hive provides are well tuned and use them without modifications.

We carefully follow the Hadoop configurations used by Hive for the TPC-H benchmarking. We only make a few small modifications. First, we set each

3. These nodes are slave nodes. To make Hadoop run, we use an additional master node to run NameNode and JobTracker.

slave node to run two MapTasks and two ReduceTasks concurrently instead of four since we only have two cores in each slave node. Second, we set the sort buffer to 500MB to ensure that MapTask could hold all intermediate pairs in memory. This setting makes both systems, Map-Join-Reduce and Hive, run a little faster. Third, we set the HDFS block size to 512MB instead of 128MB that Hive used in the TPC-H benchmarking. This is because we observe that although Hive set block size to 128MB, it manually sets the minimal chunk split size to 512MB in each query. This setting is in line with our observation that MapTask should process reasonable sized data chunk to amortize the startup cost. So, we directly use 512MB block size. Hive enables map output compression in its benchmark. At present, we do not support compression and therefore we disable it. Disabling compression will not significantly affect the performance. According to another benchmarking results published by Hive, enabling compression only improves the performance by less than 4% [26]. The final modification is to enable JVM task reuse.

5.1.2 Map-Join-Reduce Settings

Map-Join-Reduce shares the same common Hadoop settings with Hive. Furthermore, we set joiner output buffer to 150MB.

5.2 Performance Study of Tuple Parsing

This benchmark studies whether MapReduce's runtime parsing cost can be reduced. Since Hive is a system, we have no method to only test its parsing component. Therefore, we compare our parsing library (called MJR approach) with the code that is used in [10] (called the Java approach). The differences between the two approaches is that our code does not create temporary objects in splitting and parsing while Java code does.

We create a one-node cluster and populate HDFS with 725MB `lineitem` dataset. We run two MapReduce jobs to test the performance. The first job extracts the tuple structure by reading a line from the input as a tuple and then splitting it into fields according to the delimiter, '—'. The second job splits the tuple into fields and parses two date columns, i.e., `l_commitdate` and `l_receiptdate`, to compares which date is early. Here, the computation is merely for testing purpose. We are interested in whether the parsing cost is acceptable. Both jobs only have map functions and do not produce output into HDFS. The minimal file split size is set to 1GB so that the mapper will take the whole data set as input. We only report the execution time of the mapper and ignore the startup cost of the job. Figure 4 plots the results. In Figure 4, the left two columns represents the time MJR code used to split and parse the tuple. The right two columns records the execution time of Java code. We can see that our split approach runs about four timers faster than Java code. Furthermore, parsing two columns actually only introduces very little overhead (less than one second). The results confirm our claim made in Section 4.1. The cost in the runtime parsing is mainly due to the creation of temporary java objects. By proper and careful coding, much of the cost can be removed.

5.3 Analytical Tasks

We benchmark Map-Join-Reduce against Hive. The original Hive's TPC-H benchmark [25] runs on a 11-node cluster with ten slaves to process a TPC-H 100GB dataset, with 10GB data per node. Each slave node has 4 cores, 8GB memory, 4 hard disks with 1.6TB space. However, our EC2 instance only has 2 cores, 7.5GB memory, and 1 hard disk. Therefore, to enable the benchmark to be completed within a reasonable time frame, we process 5GB data in each node. With the use of 10, 50, 100-node clusters, we subsequently have three datasets of 50GB, 250GB and 500GB. Unfortunately, Hive fails to perform all four analytical queries using the 500GB dataset. The JVM throws various runtime exceptions, such as "GC overhead limits exceeded" during query processing. This problem is not due to our modifications to the Hadoop since the same problem occurred when we ran Hive on the standard Hadoop v0.19.2 release. Therefore, for the 500GB dataset, we only report results of Map-Join-Reduce. Instead, for the 100-node cluster, we reduce the dataset size to 350GB so that Hive can complete all four queries.

We choose four TPC-H queries for benchmarking, namely Q3, Q4, Q7 and Q9. Each query is executed three times and the average of three runs is reported. For Hive, we use the latest 0.4.0 release. HiveQL scripts are also used for query submission. For Map-Join-Reduce, all the programs are hand coded. For each query, we specify the same join order with Hive. We set HDFS replication factor $r = 3$, when the dataset and results are replicated twice. That is, three copies of the data are stored. The effect of replication on performance is studied in Section 5.3.5 when no replication is used. Data are generated by TPC-H DBGEN tool and are loaded to HDFS as text files. We do not report loading time since both systems directly perform queries on files.

It is useful to list each query's SQL and HiveQL script. However, the full list will run out of our space limit. Therefore, we only present each query's execution flow in Hive. The details of the SQL query and HiveQL scripts can be found in [19] and [25]. Hive is able to dynamically determine the number of reducers to use based on the input size. Map-Join-Reduce, however, gives the user more freedom to specify the number of reducers to be used. To make a fair comparison, we set the number of reducers to be no more than the total number of reducers that Hive

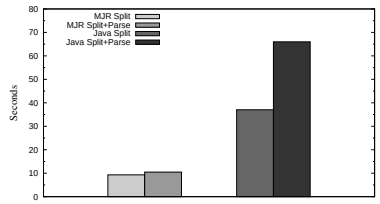


Fig. 4. Tuple Parsing Results

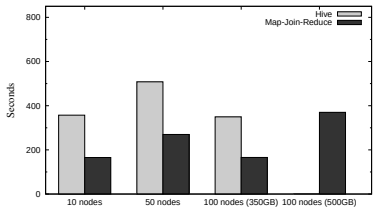


Fig. 5. TPC-H Q4 (r=3)

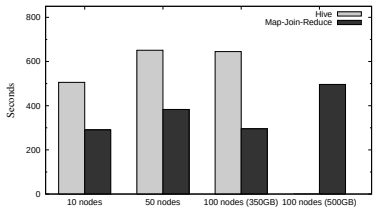


Fig. 6. TPC-H Q3 (r=3)

used in processing the same query. For example, if Hive uses 50 reducers to process a query, we will set the number of reducers to no more than 50 in Map-Join-Reduce. We could not set the number of reducers to be same with Hive since some reducer number, e.g., a prime number, makes us fail to build partition functions.

5.3.1 TPC-H Q4

This query joins `lineitem` with `orders` and counts the number of `orders` based on order priority. Hive uses four MapReduce jobs to evaluate this query. The first job writes the unique `l_orderkey`s in `lineitem` to HDFS. The second job joins the unique `l_orderkey`s with `orders` and writes the join results to HDFS. The third job aggregates joined tuples and writes the aggregation results to HDFS. The forth job merges results into one HDFS file.

Map-Join-Reduce launches two MapReduce jobs to evaluate the query. The first job performs the ordinary filtering, joining and partial aggregation tasks on `lineitem` and `orders`. The second job combines all partial aggregation results and produce the final answers to HDFS. The partition function for this query is straight forward. Since there are only two datasets involved in the query, one partition function is sufficient. It partitions tuples from `lineitem` and `orders` to all available reducers. We set the number of reducers to the be sum of reducers in the first and second jobs that Hive launched.

Figure 5 presents the performance of each system. In general, Map-Join-Reduce runs twice faster than Hive. The main reason for Hive to be slower than Map-Join-Reduce is because Hive uses two MapReduce jobs to join `lineitem` and `orders`. This plan causes the intermediate results, namely the unique keys of `lineitem` produced by J_1, to be shuffled again, for joining, in J_2. Actually, to speed up writing unique `l_orderkeys` to HDFS, Hive already partitions and shuffles these keys to all reducers in J_1. If this shuffling can also be applied to `orders` in J_1 and thus shuffles all qualified `orders` tuples to reducers for joining, we think Hive is able to deliver the same performance as Map-Join-Reduce.

5.3.2 TPC-H Q3

Hive runs Q3 using five MapReduce jobs. The first job J_1 joins the qualified tuples in `customer` and `orders` and produce the join results I_1 to HDFS. The second job J_2 joins I_1 and `lineitem`, writes join results I_2 to HDFS. The third job J_3 aggregates I_2 on the group keys. The fourth job J_4 sorts the aggregation results in decreasing order of `revenue`. The final job J_5 limits the results to top ten orders with the largest revenues and writes these ten result tuples into HDFS.

Map-Join-Reduce can process the query with two jobs. The first job scans all three datasets and shuffles qualified tuples to reducers. At the reducer side, two joiners are chained to join all tuples. The join order is the same as Hive's, namely first joining `orders` and `customer`, followed by joining the results with `lineitem`. In partial aggregation, the reducers in the first job maintain a heap to hold the top ten tuples with the largest revenues. Finally, the second job combines partial aggregations and produces the final query answers. Two partition functions are built for the first job to partition intermediate pairs. The domains of the partition functions are $\text{Dom}[H_1] = \{\texttt{c\_custkey}, \texttt{o\_custkey}\}$ and $\text{Dom}[H_2] = \{\texttt{l\_orderkey}, \texttt{o\_orderkey}\}$. We set the number of reducers employed in the first job to be close to the sum of reducers Hive used in J_1, J_2, and J_3. The partition numbers in each partition function are tuned by the brute-force search algorithm described in Section 4.2.

Figure 6 illustrates the results of this benchmark task. Although Map-Join-Reduce can perform all joins within one job, it only runs twice faster than Hive. This is because the intermediate join results are small (relative to the input). Therefore checkpointing intermediate join results and shuffling those results in the complete job does not introduce too much overhead. In 350GB dataset, the first job of Hive only writes 1.7GB intermediate results into HDFS which is much smaller than the input datasets. Since Map-Join-Reduce may need more memory and disk space in each processing node, this observation suggests that if the computation resources are inadequate and the intermediate results are small, sequential processing should be used instead without too much performance degradation.

5.3.3 TPC-H Q7

Hive compiles this query to ten MapReduce jobs. The first four jobs perform a self-join on `nation` to find out desired `nation` key pairs with the given `nation` names. The key pairs that have been found are written to HDFS as temporary results I_1. Then another four jobs are launched to join `lineitem`,

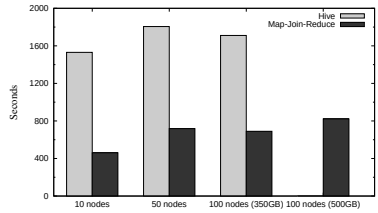


Fig. 7. TPC-H Q7 (r=3)

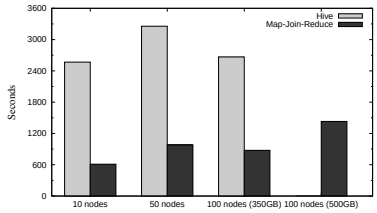


Fig. 8. TPC-H Q9 (r=3)

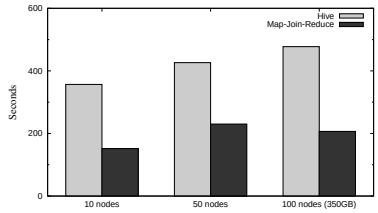


Fig. 9. TPC-H Q4 (r=1)

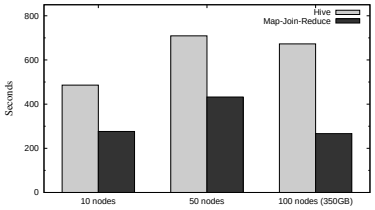


Fig. 10. TPC-H Q3(r=1)

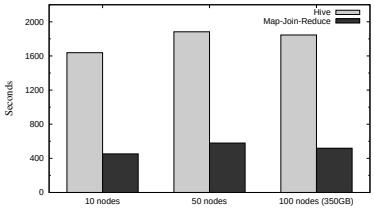


Fig. 11. TPC-H Q7 (r=1)

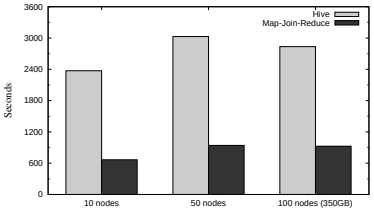


Fig. 12. TPC-H Q9 (r=1)

orders, customer, supplier, and I_1. Finally two additional jobs are launched to compute aggregations, order result tuples and store them into HDFS.

In contrast to the complexity of Hive, Map-Join-Reduce only needs two MapReduce jobs to evaluate the query. The simplicity of Map-Join-Reduce is its strong point and as expected, Map-Join-Reduce is much easier to use than MapReduce to represent complex analysis logic. This feature becomes important if users prefer to write programs to analyze the data. In the first MapReduce job, mappers scan the datasets in parallel. We load nation into the memory of mappers which scan supplier and customer for map-side join. The logic of reducer side is as per normal. Three joiners, linked as Hive's join order, join the tuples and push the results to reducer for aggregation. Three partition functions are also built. Their domains are Dom$[H_1] = \{$s_suppkey, l_suppkey$\}$, Dom$[H_2] = \{$c_custkey, o_custkey$\}$, and Dom$[H_3] = \{$o_orderkey, l_orderkey$\}$. The partial aggregates ordered by group key are combined in the second job to generate the final query result. We also use the brute-force search method for parameters tuning.

Figure 7 presents the performance of both systems. On average, Map-Join-Reduce runs nearly three times faster than Hive. We attribute this significant performance boost to the benefit of avoiding checkpointing and shuffling. In the test with 350GB dataset, Hive needs to write more than 88GB data to HDFS. All these data need to be shuffled again in the next MapReduce job. Frequent shuffling of such large volume of data incurs huge performance overhead. Thus, Hive is significantly slower than Map-Join-Reduce.

5.3.4 TPC-H Q9

Hive compiles this query into seven MapReduce jobs. Five jobs are to join lineitem, supplier, partsupp, part, and orders. Hive also joins nation with supplier in the map side. Then, two additional jobs are launched for aggregation and ordering. Map-Join-Reduce still performs the query over two MapReduce jobs. The procedure is similar to the previous queries. Therefore, we only list partition functions and omit other details. The domain of partition functions are Dom$[H_1] = \{$s_suppkey, l_suppkey, ps_suppkey$\}$, Dom$[H_2] = \{$l_partkey, ps_partkey, p_partkey$\}$, and Dom$[H_3] = \{$o_orderkey, l_orderkey$\}$. We also join supplier with nation on the map-side.

Figure 8 plots the result of this benchmark. This query shows the largest performance gap between the two systems. Map-Join-Reduce runs nearly four times faster than Hive. This is because Hive needs to checkpoint and shuffle a huge amount of intermediate results. In the test with 350GB dataset with 100 nodes, Hive needs to checkpoint and shuffle more than 300GB intermediate join results. Although Map-Join-Reduce also incurs more than 400GB disk I/O accesses in the first job's reducers for holding input dataset and join results, it is less costly to read and write data to the local disks than to shuffle those results over the network. Therefore, the significant performance boost is mainly due to the ability of Map-Join-Reduce in joining all datasets locally, which is its another strong point.

5.3.5 Effect of Replication

We also set the replication factor $r = 1$ to study how replication affects the performance. In this setting, the datasets and intermediate results are not replicated by HDFS and thus have only one copy. Since Hive failed to process 500GB dataset, we only conduct this test on 350GB dataset in a 100-node cluster.

Figures 9, 10, 11, and 12 present the results of this benchmark. We do not see significant performance improvements for both systems. It is reasonable for the performance of Map-Join-Reduce to be insensitive to replication since replication only affects the data volume that are written to HDFS, and Map-Join-Reduce only writes small partial aggregations to

HDFS. We also observe that in some settings, Map-Reduce-Join runs a little slower than the version that runs with three replications setting. This is because fewer replications increase the chance of JobTracker to schedule non-data local map tasks.

For the tasks that produce small intermediate results, i.e., Q3 and Q4, there is also no performance improvement in Hive. However, for queries producing large intermediate results, i.e., Q9, setting replication to one improves the performance of Hive by 10%.

6 Related Work

There are two kinds of systems that are able to perform large-scale data analysis tasks on a shared-nothing cluster: 1) Parallel Databases, and 2) MapReduce based systems.

The research on parallel databases started in the late 1980s [7]. Pioneering research systems include Gamma [8], and Grace [9]. A full comparison between parallel databases and MapReduce are presented in [10] and [11]. The comparison shows that main differences between the two systems are performance and scalability. The scalability issues of the two systems are further studied in [12].

Efficient join processing is also extensively studied in parallel database systems. The proposed work can be categorized as two classes: 1) two-way join algorithms and 2) schemes for evaluating multi-way joins based on two-way joins. Work in the first category include parallel nested loop join, parallel sort-merge join, parallel hash join, and parallel partition join [8]. All these join algorithms have been implemented in MapReduce based systems in one form or another [13], [14], [12]. Although Map-Join-Reduce targets a multi-way join, these two-way joins techniques can also be integrated in our Map-Join-Reduce framework. We discuss this problem in Section 2.5.

Parallel database systems evaluate multi-join queries through a pipelined processing strategy. Suppose we are to perform a three-way join $R_1 \bowtie R_2 \bowtie R_3$. Typical pipelined processing works as follows: First, two nodes N_1 and N_2 scan R_2 and R_3 in parallel and load them into an in-memory hash table if the tables can fit in memory. Then, the third node N_3 reads tuples from R_1 and pipelines the read tuples to N_2 and N_3 to probe R_2 and R_3 in turn and produce the final query results. Pipelined processing is proven to be superior to sequential processing [15]. However, pipelined processing suffers from node failure since it introduces dependencies between processing nodes. When a node (say N_2) fails, the data flow is broken and thus the whole query needs to be resubmitted. Therefore, all MapReduce based systems adopt the sequential processing strategy.

MapReduce is introduced by Dean et al. for simplifying construction of inverted indexes [1]. But it was quickly found that the framework is also able to perform filtering-aggregation data analysis tasks [16]. More complex data analytical tasks can also be evaluated by a set of MapReduce jobs [1], [10]. Although join processing can be implemented in a MapReduce framework, processing heterogenous datasets and manually writing a join algorithm is not straightforward. In [3], Map-Reduce-Merge is proposed for simplifying join processing by introducing a Merge operation. Compared to this work, Map-Join-Reduce aims to not only alleviate development effort but also improve the performance of multi-way join processing. There are also a number of query processing systems that are built on top of MapReduce, including Pig [6], Hive [5] and Cascading [17]. These systems provide a high-level query language and associated optimizer for efficient evaluating complex queries which may involve multiple joins. Compared to this work, Map-Join-Reduce provides built-in support for multi-way join processing and processes join in system level rather than application level. Therefore, Map-Join-Reduce can be used as a new building block (in addition to MapReduce) for these systems to generate an efficient query plan.

During preparation of the paper, we noticed that the work presented in a newly accepted paper [18] is closely related to ours. In [18], the authors propose an efficient multi-way join processing strategy for MapReduce system. The basic idea of the join processing strategy is similar to ours. Our work is an independent work. Compared to [18], our work not only targets at efficient join processing but also aims to simplify developing complex data analytical tasks. For the join processing technique, the differences between their work with ours lies in technical details. First, we adopt a derived join graph approach for generating k-partition function and consider every join column. Contrary to our approach, their work do not provide a concrete method for constructing partition functions. The join columns that are included in partition functions are determined by Lagrangean optimization process. Not every join column will be included. Second, to optimize parameters in partition functions, we adopt a heuristic solution to solve an integer programming problem. This heuristic solution guarantees feasible parameters (satisfying all constraints) can be always be found although the parameters may not be optimal. Contrary to our approach, their Lagrangean optimization technique dost not guarantee feasible parameters can always be found. In cases where no feasible parameters are returned, they employ alternative strategies for deriving parameters. Finally, we provide a working system and a comprehensive benchmarking performance study of the proposed approach.

7 Conclusion

In this paper, we present Map-Join-Reduce, a system that extends and improves the MapReduce runtime

system to efficiently process complex data analytical tasks on large clusters. The novelty of Map-Join-Reduce is that it introduces a filtering-join-aggregation programming model. This programming model allows users to specify data analytical tasks that require joining multiple datasets for aggregate computation with a relatively simple interface offering three functions: `map()`, `join()` and `reduce()`. We have designed a one-to-many shuffling strategy and demonstrated the usage of such a shuffling strategy in efficiently processing filtering-join-aggregation tasks. We have conducted a benchmarking study against Hive on Amazon EC2 using the TPC-H benchmark. The benchmarking results show that our approach significantly improves the performance of complex data analytical tasks, and confirm the potential of the Map-Join-Reduce approach.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," pp. 137–150. [Online]. Available: http://www.usenix.org/events/osdi04/tech/dean.html

[2] "http://developer.yahoo.net/blogs/hadoop/2008/09/."

[3] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, "Map-Reduce-Merge: simplified relational data processing on large clusters," in *SIGMOD*, 2007.

[4] D. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, and A. Krioukov, "Clustera: an integrated computation and data management system," *VLDB*, 2008.

[5] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wychoff, and R. Murthy, "Hive - a warehousing solution over a map-reduce framework," in *VLDB*, 2009.

[6] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD*, 2008.

[7] D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Commun. ACM*, 1992.

[8] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna, "Gamma - a high performance dataflow database machine," in *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 228–237.

[9] S. Fushimi, M. Kitsuregawa, and H. Tanaka, "An overview of the system software of a parallel relational database machine grace," in *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 209–219.

[10] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD*. ACM, June 2009. [Online]. Available: http://database.cs.brown.edu/sigmod09/benchmarks-sigmod09.pdf

[11] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "Mapreduce and parallel dbmss: friends or foes?" *Communications of the ACM*, vol. 53, no. 1, pp. 64–71, 2010.

[12] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads," in *VLDB*, Lyon, France, 2009.

[13] "http://hadoop.apache.org."

[14] K. S. Beyer, V. Ercegovac, R. Krishnamurthy, S. Raghavan, J. Rao, F. Reiss, E. J. Shekita, D. E. Simmen, S. Tata, S. Vaithyanathan, and H. Zhu, "Towards a scalable enterprise content analytics platform," *IEEE Data Eng. Bull.*, vol. 32, no. 1, pp. 28–35, 2009.

[15] D. A. Schneider and D. J. DeWitt, "Tradeoffs in processing complex join queries via hashing in multiprocessor database machines," in *VLDB '90: Proceedings of the 16th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 469–480.

[16] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Sci. Program.*, vol. 13, no. 4, pp. 277–298, 2005.

[17] "http://www.cascading.org."

[18] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," *to appear in EDBT*, 2010.

[19] "http://www.tpc.org/tpch/."

[20] D. A. Schneider and D. J. DeWitt, "A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment," *SIGMOD Rec.*, vol. 18, no. 2, 1989.

[21] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, "C-store: A column-oriented dbms," in *VLDB*, Trondheim, Norway, 2005, pp. 553–564.

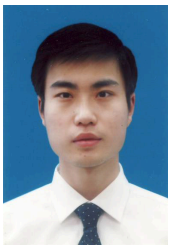
[22] M. Ziane, M. Zaït, and P. Borla-Salamet, "Parallel query processing with zigzag trees," *The VLDB Journal*, vol. 2, no. 3, pp. 277–302, 1993.

[23] A. Weintraub, "Integer programming in forestry," *Annals OR*, vol. 149, no. 1, pp. 209–216, 2007.

[24] "http://www.cloudera.com/blog/2009/05/07/what's-new-in-hadoop-core-020/."

[25] "http://issues.apache.org/jira/browse/hive-600."

[26] "http://issues.apache.org/jira/browse/hive-396."



Dawei Jiang is currently a Research Fellow at the school of computing, National University of Singapore. He received both his B.Sc. and Ph.D. in computer science from the Southeast University in 2001 and 2008 respectively. His research interests includes Cloud computing, database systems and large-scale distributed systems.

Anthony K. H. TUNG is currently an Associate Professor in the School of Computing, National University of Singapore (NUS) and a junior faculty member in the NUS Graduate School for Integrative Sciences and Engineering and a SINGA supervisor. He received both his B.Sc.(2nd Class Honour) and M.Sc. in computer sciences from the National University of Singapore in 1997 and 1998 respectively. In 2001, he received the Ph.D. in computer sciences from Simon Fraser University (SFU). His research interests span across the whole process of converting data into intelligence. Prof. Anthony is also affiliated with the Database Lab, Computational Biology Lab and the NUS Bioinformatics Programme.

Gang Chen is currently a Professor at the College of Computer Science, Zhejiang University. He received his B.Sc., M.Sc. and Ph.D. in computer science and engineering from Zhejiang University in 1993, 1995 and 1998 respectively. His research interests include database, information retrieval, information security and computer supported cooperative work. Prof. Gang is also the executive director of Zhejiang University – Netease Joint Lab on Internet Technology.