Distributed Data Management Using MapReduce

FENG LI and BENG CHIN OOI, National University of Singapore
M. TAMER ÖZSU, University of Waterloo
SAI WU, Zhejiang University

MapReduce is a framework for processing and managing large-scale datasets in a distributed cluster, which has been used for applications such as generating search indexes, document clustering, access log analysis, and various other forms of data analytics. MapReduce adopts a flexible computation model with a simple interface consisting of *map* and *reduce* functions whose implementations can be customized by application developers. Since its introduction, a substantial amount of research effort has been directed toward making it more usable and efficient for supporting database-centric operations. In this article, we aim to provide a comprehensive review of a wide range of proposals and systems that focusing fundamentally on the support of distributed data management and processing using the MapReduce framework.

Categories and Subject Descriptors: H.2.4 [**Systems**]: Parallel Databases; H.2.4 [**Systems**]: Distributed Databases; C.2.4 [**Distributed Systems**]: Distributed Databases

General Terms: Algorithms, Performance

Additional Key Words and Phrases: MapReduce, scalability, Hadoop

ACM Reference Format:
Feng Li, Beng Chin Ooi, M. Tamer Özsu, and Sai Wu. 2014. Distributed data management using MapReduce. ACM Comput. Surv. 46, 3, Article 31 (January 2014), 42 pages.
DOI: http://dx.doi.org/10.1145/2503009

1. INTRODUCTION

Database Management Systems (DBMSs) have become a ubiquitous operational platform in managing huge amounts of business data. DBMSs have evolved over the last four decades and are now functionally rich. While enterprises are struggling with the problem of poor database scalability, a new challenge has emerged that has impacted the IT infrastructures of many modern enterprises. DBMSs have been criticized for their monolithic architecture that has been claimed to make them "heavyweight" and expensive to operate. It is sometimes argued that they are not efficient for many data management tasks despite their success in business data processing. This challenge has been labeled as the big data problem. In principle, while earlier DBMSs focused on modeling operational characteristics of enterprises, big data systems are now expected to model user behaviors by analyzing vast amounts of user interaction logs. There have been various proposals to restructure DBMSs (e.g., Chaudhuri and Weikum [2000] and Stonebraker et al. [2007]), but the basic architecture has not changed dramatically.

Authors' addresses: F. Li and B. C. Ooi, School of Computing, National University of Singapore, Singapore; M. T. Özsu, Cheriton School of Computer Science, University of Waterloo, Canada; S. Wu, Department of Computer Science, Zhejiang University, China; emails: li-feng@comp.nus.edu.sg; ooibc@comp.nus.edu.sg; tamer.ozsu@uwaterloo.ca; wusai@zju.edu.cn.

DOI: http://dx.doi.org/10.1145/2503009

With the increasing amount of data and the availability of high-performance and relatively low-cost hardware, database systems have been extended and parallelized to run on multiple hardware platforms to manage scalability [Özsu and Valduriez 2011]. Recently, a new distributed data processing framework called MapReduce was proposed [Dean and Ghemawat 2004] whose fundamental idea is to simplify the parallel processing using a distributed computing platform that offers only two interfaces: `map` and `reduce`. Programmers implement their own `map` and `reduce` functions, while the system is responsible for scheduling and synchronizing the map and reduce tasks. The MapReduce model can be used to solve the "embarrassingly parallel" problems,[1] where little or no effort is required to partition a task into a number of parallel but smaller tasks. MapReduce is being used increasingly in applications such as data mining, data analytics, and scientific computation. Its wide adoption and success lie in its distinguishing features, which can be summarized as follows.

(1) **Flexibility**. Since the code for `map` and `reduce` functions are written by the user, there is considerable flexibility in specifying the exact processing that is required over the data rather than specifying it using SQL. Programmers can write simple `map` and `reduce` functions to process petabytes of data on thousands of machines without the knowledge of how to parallelize the processing of a MapReduce job.
(2) **Scalability**. A major challenge in many existing applications is to be able to scale to increasing data volumes. In particular, *elastic scalability* is desired, which requires the system to be able to scale its performance up and down dynamically as the computation requirements change. Such a "pay-as-you-go" service model is now widely adopted by the cloud computing service providers, and MapReduce can support it seamlessly through data-parallel execution. MapReduce was successfully deployed on thousands of nodes and able to handle petabytes of data.
(3) **Efficiency**. MapReduce does not need to load data into a database, which typically incurs high cost. It is, therefore, very efficient for applications that require processing the data only once (or only a few times).
(4) **Fault tolerance**. In MapReduce, each job is divided into many small tasks that are assigned to different machines. Failure of a task or a machine is compensated by assigning the task to a machine that is able to handle the load. The input of a job is stored in a distributed file system where multiple replicas are kept to ensure high availability. Thus, the failed map task can be repeated correctly by reloading the replica. The failed reduce task can also be repeated by repulling the data from the completed map tasks.

The criticisms of MapReduce center on its reduced functionality, requiring considerable amount of programming effort, and its unsuitability for certain types of applications (e.g., those that require iterative computations) [DeWitt et al. 2008; Dewitt and Stonebraker 2009; Pavlo et al. 2009; Stonebraker et al. 2010]. MapReduce does not require the existence of a schema and does not provide a high-level language such as SQL. The flexibility advantage mentioned earlier comes at the expense of considerable (and usually sophisticated) programming on the part of the user. Consequently, a job that can be performed using relatively simple SQL commands may require considerable amount of programming in MapReduce, and this code is generally not reusable. Moreover, MapReduce does not have built-in indexing and query optimization support, always resorting to scans. The potential performance drawback of MapReduce has been reported [Pavlo et al. 2009] on the basis of experiments on two benchmarks: TPC-H and a customized benchmark tailored for search engines. In a 100-node cluster, a parallel database system and a column-wise data manage system

[1] http://parlab.eecs.berkeley.edu/wiki/media/patterns/map-reduce-pattern.doc.

called Vertica (http://www.vertica.com) show superior performance than Hadoop (http://hadoop.apache.org/) implementation of MapReduce for various workloads, including simple grep, join, and aggregation jobs.

Since the introduction of MapReduce, there has been a long stream of research that attempts to address the problems highlighted earlier, and this indeed remains an active area of research. Considerable effort has been spent on efficient implementation of the MapReduce framework. There have been proposals for more sophisticated scheduling algorithms [Zaharia et al. 2008] and parsing schemes [Jiang et al. 2010] to improve performance. There have also been a number of works to extend the framework to support more complex applications [Condie et al. 2010; Bu et al. 2010]. High-level declarative (Hive [Thusoo et al. 2009] and Pig [Olston et al. 2008]) and procedural languages (Sawzall [Pike et al. 2005]) as well as a Java library (FlumeJava [Chambers et al. 2010]) have also been proposed for the MapReduce framework to improve its ease of use (Section 3.2).

The focus of this survey is on large-scale data processing using MapReduce. The ease with which MapReduce programs can be parallelized has caused the migration of a number of applications to the MapReduce platform. There are two aspects of supporting DBMS functionality over MapReduce. The first aspect is the implementation of database operators, such as `select`, `project`, and so forth, as MapReduce jobs, and specific indexes [Dittrich et al. 2010] to support these implementations (Section 4). The second aspect is to combine these implementations to create a fully functional DBMS/data warehouse on MapReduce (Section 5). Example MapReduce-based DBMS implementations include HadoopDB [Abouzeid et al. 2009], Llama [Lin et al. 2011], MR-Share [Nykiel et al. 2010], and Cheetah [Chen 2010]. In addition, traditional database systems sometimes provide MapReduce as a built-in feature (e.g., Greenplum and Aster) [Friedman et al. 2009].

Within this context, the objectives of this survey are fourfold. First we introduce this new and increasingly widely deployed distributed computing paradigm (Section 2) and its current implementations (Section 3). Second, we present the current research on enhancing MapReduce to better address modern data-intensive applications without losing its fundamental advantages (Sections 4 and 5). Third, we discuss ongoing work in extending MapReduce to handle a richer set of workloads such as streaming data and iterative computations (Section 6). Finally, we briefly review a number of recent systems that may have been influenced by MapReduce (Section 7). We assume that the readers are familiar with basic data management terminology and concepts.

2. MAPREDUCE TECHNOLOGY

2.1. MapReduce Programming Model

MapReduce is a simplified parallel data processing approach for execution on a computer cluster [Dean and Ghemawat 2004]. Its programming model consists of two user-defined functions, `map` and `reduce`[2] (Table I).

The inputs of the `map` function are a set of key/value pairs. When a MapReduce job is submitted to the system, the map tasks (which are processes that are referred to as *mappers*) are started on the compute nodes and each map task applies the `map` function to every key/value pair ($k1$, $v1$) that is allocated to it. Zero or more intermediate key/value pairs ($list(k2, v2)$) can be generated for the same input key/value pair. These intermediate results are stored in the local file system and sorted by the keys. After all the map tasks complete, the MapReduce engine notifies the reduce tasks (which are also processes that are referred to as *reducers*) to start their processing. The reducers

[2] As a convention, we will use `courier` font when we refer to the specific function or interface and the regular font when we refer to the processing of the corresponding function.

Table I. `map` and `reduce` Functions

`map`	$(k1, v1) \rightarrow list(k2, v2)$
`reduce`	$(k2, list(v2)) \rightarrow list(v3)$

Table II. UserVisits Table

sourceIP	VARCHAR (16)
destURL	VARCHAR (100)
adRevenue	FLOAT
userAgent	VARCHAR (64)
countryCode	VARCHAR (3)
languageCode	VARCHAR (6)
searchWord	VARCHAR (32)
duration	INT

will pull the output files from the map tasks in parallel and merge-sort the files obtained from the map tasks to combine the key/value pairs into a set of new key/value pairs $(k2, list(v2))$, where all values with the same key $k2$ are grouped into a list and used as the input for the `reduce` function. The `reduce` function applies the user-defined processing logic to process the data. The results, normally a list of values, are written back to the storage system.

As an example, given the table UserVisits shown in Table II,[3] one typical job is to calculate the total adRevenue for each sourceIP. A possible MapReduce specification of the `map` and `reduce` functions is shown in Algorithms 1 and 2. The algorithms assume that the input dataset is in text format where the tuples are separated by lines and columns are separated by the character "|". Each mapper parses tuples assigned to it and generates a key/value pair (sourceIP, adRevenue). The results are first cached in the mapper's local disk and then copied (commonly called *shuffling*) to the reducers. In the reduce phase, key/value pairs are grouped into (sourceIP, (adRevenue1, adRevenue2, adRevenue3, ...)) based on their keys (i.e., sourceIP). Each reducer processes these pairs by summarizing the adRevenue for one sourceIP, and the result (sourceIP, sum(adRevenue)) is generated and returned.

ALGORITHM 1: Map Function for UserVisits

```
input: String key, String value
String[] array = value.split("|");
EmitIntermediate(array[0], ParseFloat(array[2]));
```

ALGORITHM 2: Reduce Function for UserVisits

```
input: String key, Iterator values
float totalRevenue = 0;
while values.hasNext() do
    totalRevenue + = values.next();
end
Emit(key, totalRevenue);
```

2.2. MapReduce Architecture

MapReduce adopts a loosely coupled design, where the processing engine is independent of the underlying storage system. This design allows the processing and the storage layers to scale up and down independently and as needed. The storage system typically makes use of an existing Distributed File System (DFS), such as Google File System (GFS) [Ghemawat et al. 2003]; Hadoop Distributed File System (HDFS),[4]

[3] The same schema was used as a benchmark dataset in Pavlo et al. [2009].

[4] http://hadoop.apache.org/hdfs/.

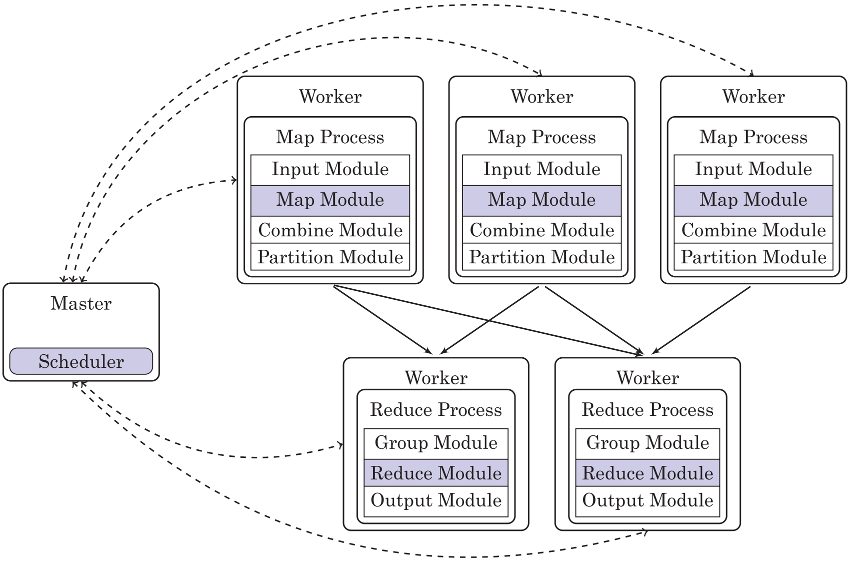


Fig. 1. Architecture of MapReduce.

which is a Java implementation of Google File System; RAMCloud [Ongaro et al. 2011]; or LogBase [Vo et al. 2012]. Based on the partitioning strategy employed by the DFS, data are split into equal-size chunks and distributed over the machines in a cluster. Each chunk is used as an input for a mapper. Therefore, if the dataset is partitioned into *k* chunks, MapReduce will create *k* mappers to process the data.

The MapReduce processing engine has two types of nodes, the *master* node and the *worker* nodes, as shown in Figure 1. The master node controls the execution flow of the tasks at the worker nodes via the *scheduler* module. Each worker node is responsible for a `map` or `reduce` process. The basic implementation of the MapReduce engine needs to include the following modules (which are marked as the gray boxes in Figure 1):

(1) *Scheduler*. The *scheduler* is responsible for assigning the map and reduce tasks to the worker nodes based on data locality, network state, and other statistics of the worker nodes. It also controls fault tolerance by rescheduling a failed process to other worker nodes (if possible). The design of the *scheduler* significantly affects the performance of the MapReduce system.
(2) *Map module*. The *map module* scans a data chunk and invokes the user-defined `map` function to process the input data. After generating the intermediate results (a set of key/value pairs), it groups the results based on the partition keys, sorts the tuples in each partition, and notifies the master node about the positions of the results.
(3) *Reduce module*. The *reduce module* pulls data from the mappers after receiving the notification from the master. Once all intermediate results are obtained from the mappers, the reducer merges the data by keys, and all values with the same key are grouped together. Finally, the user-defined function is applied to each key/value pair, and the results are output to DFS.

Given its stated purpose of scaling over a large number of processing nodes, a MapReduce system needs to support fault tolerance efficiently. When a map or reduce task fails, another task on a different machine is created to re-execute the failed task. Since the mapper stores the results locally, even a completed map task needs to be re-executed in case of a node failure. In contrast, since the reducer stores the results in DFS, a completed reduce task does not need to be re-executed when node failure occurs.

2.3. Extensions and Optimizations of MapReduce Systems

To further improve its efficiency and usability, the basic MapReduce architecture discussed in Section 2.2 is usually extended, as illustrated in Figure 1, to include the following modules:

(1) *Input and output modules*. The *input module* is responsible for recognizing the input data with different input formats and splitting the input data into key/value pairs. This module allows the processing engine to work with different storage systems by allowing different input formats to be used to parse different data sources, such as text files, binary files, and even database files. The *output module* similarly specifies the output format of mappers and reducers.

(2) *Combine module*. The purpose of this module is to reduce the shuffling cost by performing a local reduce process for the key/value pairs generated by the mapper. Thus, it can be considered as a specific type of reducer. In our running example, to compute the sum of adRevenue for each sourceIP, we can use a combine function similar to the `reduce` function in Algorithm 1, so that for each sourceIP, the mapper only generates one result, which is the sum of the adRevenue in the corresponding data chunk. Therefore, only n tuples are shuffled to the reducers, where n is the number of unique sourceIPs.

(3) *Partition module*. This is used to specify how to shuffle the key/value pairs from mappers to reducers. The default partition function is defined as $f(key) = h(key) \% numOfReducer$, where $\%$ indicates the mod operator and $h(key)$ is the hash value of the key. A key/value pair (k, v) is sent to the $f(k)$th reducer. Users can define different partition functions to support more sophisticated behavior.

(4) *Group module*. *Group module* specifies how to merge the data received from different map processes into one sorted run in the reduce phase. By specifying the group function, which is a function of the map output key, the data can be merged more flexibly. For example, if the map output key is a composition of several attributes (sourceIP, destURL), the group function can only compare a subset of the attributes (sourceIP). As a result, in the reducer module, the reduce function is applied to the key/value pairs with the same sourceIP.

In this article, we base our discussion on the extended MapReduce architecture.

As a general processing framework, MapReduce does not specify implementation details. Therefore, techniques for efficient implementation of MapReduce systems have received a great deal of attention. The research on this topic can be classified into two categories. The first category addresses generic system optimizations for the MapReduce framework and focuses on the scheduler and input/output modules. The second category concentrates on efficient implementation for specific applications rather than generic system implementation—these involve the map, partition, group, and reduce modules. We focus on the generic system optimizations to the scheduler and input/output modules in this section.

2.3.1. Scheduler Optimizations. Scheduler is related to two important features of MapReduce: performance and fault tolerance. The scheduler makes task assignments based on data locality, which reduces the cost of network communication. Moreover, when a node crashes, the scheduler assigns the failed task to some other node. Two optimizations have been proposed for the scheduler; the first one improves performance in a heterogeneous environment where the nodes may have different computation capabilities, while the second improves performance over a multiuser cluster.

Scheduling in a heterogeneous environment. To improve performance in a heterogeneous environment, the scheduler performs a speculative execution of a task if the node is performing poorly. The task assigned to a poorly performing node is called a *straggler*. To detect stragglers, a progress score is maintained for each task. If a task has a significantly lower progress score than the other tasks that started at the same time, it is recognized as a straggler. The progress of a map task is the fraction of the data scanned, while the progress of the reduce task is computed with respect to three phases: the copy phase, the sort phase, and the reduce phase. To simplify the computation of the progress score, some MapReduce implementations (such as Hadoop,[5] an open source implementation of MapReduce) assume that these three phases have the same weight. However, in real applications, the execution speeds of the three phases are usually different. For example, the copy and the sort phases may be finished early if the data obtained from the mappers are small. Thus, the progress score of the reduce phase may not be accurate, causing false positives in recognizing stragglers.

To improve the accuracy of the progress score, a new progress indicator, Parallax [Morton et al. 2010b], considers both the execution speed of the current system and the parallelism of the cluster. At any point in time, to estimate the speed of the remaining work, Parallax utilizes the speed information of the tuples already processed. This speed changes over time, which improves the accuracy of the indicator. Furthermore, considering the parallelized environment of MapReduce, Parallax also estimates the pipeline width of the cluster for each job. This is estimated as $min(m, n)$, where m is the number of mappers required for the job and n is the number of free nodes available (taking into account other running jobs). By considering the pipeline width, Parallax has been shown to be able to estimate the remaining time accurately even when other jobs are running on the cluster concurrently. ParaTimer [Morton et al. 2010a] is an extension of Parallax that has been designed for estimating the remaining time of a MapReduce job even in the presence of failures or data skew.

To discover the stragglers more accurately based on the progress score, a new scheduling algorithm, Longest Approximate Time to End (LATE), was designed [Zaharia et al. 2008]. The (time) rate of progress of a task ($ProgressRate$) is estimated as $ProgressScore/T$, where T is the task execution time so far, which is used to estimate the execution speed of the task. The time to complete this task is then computed as $(1 - ProgressScore)/ProgressRate$. The task that will take the longest time to finish is treated as a straggler and is speculatively executed.

A system called Mantri [Ananthanarayanan et al. 2010] further classifies the causes of the stragglers according to three characteristics: bad and busy machines, cross-rack traffic, and data skew. Instead of simply running a speculative execution for a straggler, different strategies are adopted based on the cause. For example, to reduce the cross-rack traffic, a network-aware placement technique ensures that the reducers in each rack pull roughly the same amount of data from the mappers. Data skew can be avoided by collecting statistics on the input data and partitioning the data to different nodes by a load balancing algorithm [Gufler et al. 2012; Ramakrishnan et al. 2012]. Still another approach is to repartition the straggler's remaining work to other nodes when a straggler is detected that is due to data skew [Kwon et al. 2012].

Scheduling over a multiuser cluster. The original MapReduce scheduler adopts a FIFO strategy. Consequently, when multiple jobs are submitted to the clusters concurrently, the response times of the jobs are negatively affected. One solution is to partition

[5] http://hadoop.apache.org/.

the cluster into a number of subclusters, each to be used by a different user. This strategy is not optimal, since static partitioning may overprovision to users who may not have jobs to run and underprovision to others who may need more cluster nodes. The FAIR scheduler [Zaharia et al. 2009] was proposed to alleviate this problem.

The basic idea of FAIR is to divide the cluster into two levels. At the top level, the cluster is divided into different pools, each of which is assigned to a group of users and treated as private subclusters. At the second level, in each pool, several jobs share the computation nodes using a FIFO scheduling strategy. This strategy guarantees that all concurrent jobs will eventually get a certain number of nodes in their pools, making it fair. To fully utilize the cluster, if some pools are idle at some time, the nodes in these pools are allocated to the MapReduce jobs in other pools temporarily.

FAIR has introduced a number of scheduler optimizations. First, it adopts a delayed scheduling method: Instead of assigning a task immediately to a node, it waits for a certain amount of time before assignment is made to a nonlocal node. This method allows data movement to complete and improves the chances of assigning the task to the node that owns the data of that task. Second, the reduce phase needs to wait until all the mappers finish their tasks. Thus, reducers occupy the computation nodes for a long time, while some are idle most of the time. FAIR proposes a copy-compute splitting method that splits the reduce task into two tasks: copy task and compute task. When a reducer is still in its copy phase, the node can be used by other jobs, thereby improving the utilization of the whole cluster.

2.3.2. Optimizations to Input and Output Modules. The input module is responsible for (1) recognizing the format of the input data and (2) splitting the input data into raw data and parsing the raw data into key/value pairs for processing.

The input file can be stored in a local file system, a DFS, or a DBMS. Although the input/output module cannot directly affect the performance of the storage module, it can exploit the storage module more efficiently . For example, if a B^+ tree index is built on the input data, the filtering condition in the `map` function can be utilized in advance of the job submission: only the data that will pass the filtering condition are scanned. A static analysis technique can be employed to analyze the `map` function and discover the possible filters automatically [Jahani et al. 2011]. Based on the filters, the index is utilized to reduce the scan time. Moreover, since the filtering and projection information are obtained from the compiled code using static analysis, this optimization is hidden from the users and the submitted program does not need to be modified.

Before the input data can be processed by the `map` function, it must be converted into a set of key/value pairs, and each field in the key or the value part must be decoded to a certain type. Two record-decoding schemes have been proposed [Jiang et al. 2010]: immutable decoding and mutable decoding. The *immutable decoding* scheme transforms the raw data into immutable objects that are read-only and cannot be modified; their modification requires the creation of new objects. For example, if a MapReduce system is implemented in Java, the raw data are usually stored in the Java string object, which is immutable. To transform the string into a new value, the original string object needs to be replaced by a new string object. However, if the *mutable decoding* scheme is used, the original object can be modified directly without creating any new objects. It has been shown, experimentally, that the mutable decoding scheme is significantly faster than the immutable decoding scheme and improves the performance of MapReduce in the selection task by a factor of two [Jiang et al. 2010].

2.3.3. Configuration Parameters for MapReduce. In addition to the module optimizations discussed previously, it is also necessary to properly set various configuration parameters for a given MapReduce job, such as the number of map processes launched in each node, number of reduce processes, size of buffer in the map phase for sorting, and size of buffer in the reduce phase for merging. Proposals have been made to perform this configuration automatically as it is difficult to determine the optimal configuration even for sophisticated programmers. The process involves the following steps [Herodotou and Babu 2011]:

(1) *Statistics collection.* The statistics for estimating the execution time of a MapReduce job (e.g., I/O cost and map selectivity) need to be collected.
(2) *Job simulation.*After obtaining the statistics, the MapReduce job can be simulated to estimate the execution time. Several simulation tools, such as MRPerf [Wang et al. 2009], have been implemented to simulate the job at the task level.
(3) *Cost-based optimization.* Different settings of the configuration parameters result in different execution times. Finding the right combination of these parameter values can be treated as a cost-based optimization problem.

3. MAPREDUCE IMPLEMENTATIONS

In this section, we discuss some open-source implementations of the MapReduce framework. These implementations are currently dominated by Hadoop, but there are other alternatives that we discuss in Section 3.1. The basic MapReduce framework and its implementations do not provide a high-level language interface. However, there have been proposals for such languages that we present in Section 3.2.

3.1. MapReduce Implementations

Our focus in this section is a brief description of basic MapReduce implementations. There have been extensions to these systems to support a richer set of queries (e.g., streaming and iterative queries) that are not covered in this section. Those are discussed in Section 6.

3.1.1. Hadoop. Hadoop is currently the most popular open-source MapReduce implementation. It is written in Java and has been tested in Yahoo's cluster. Although Hadoop can be deployed on different storage systems, the released Hadoop package includes HDFS as the default storage system.

The two modules of Hadoop, namely, HDFS and the processing engine, are loosely connected. They can either share the same set of compute nodes or be deployed on different nodes. In HDFS, two types of nodes are created: the *name node* and *data node*. The name node records how data are partitioned and monitors the status of data nodes in HDFS. Data imported into HDFS are split into equal-size chunks and the name node distributes the data chunks to different data nodes, which store and manage the chunks assigned to them. The name node also acts as the dictionary server, providing partitioning information to applications that search for a specific chunk of data.

Hadoop creates two processes with specific functions: *job tracker* and *task tracker*. The job tracker is commonly hosted on the master node in the MapReduce framework and implements the scheduler module in Figure 1. Thus, it splits a submitted job into map and reduce tasks and schedules them to available task trackers. Each worker node in the cluster starts a task tracker process and accepts the request from the job tracker. If the request is a map task, the task tracker will process the data chunk specified by the job tracker. Otherwise, it initializes the reduce task and waits for the notification from the job tracker (that the mappers are done) to start processing.

In addition to writing the `map` and `reduce` functions, programmers can exert further control (e.g., input/output formats and partitioning function) by means of eight User-Defined Functions (UDFs) that Hadoop provides. The default functionality of these functions is given in Table III, but these can be overwritten by customized implementations. For instance, in the UserVisits table (Table II), the search engine can issue

Table III. UDF Functions in Hadoop

Phase	Name	Function
Map	InputFormat::getSplit	Partition the input data into different splits. Each split is processed by a mapper and may consist of several chunks.
	RecordReader::next	Define how a split is divided into items. Each item is a key/value pair and used as the input for the `map` function.
	Mapper::map	Users can customize the `map` function to process the input data. The input data are transformed into some intermediate key/value pairs.
	Job::setSortComparator WritableComparable::compareTo	Specify the class for comparing the key/value pairs. Define the default comparison function for user-defined key class.
	Job::setCombinerClass	Specify how the key/value pairs are aggregated locally.
Shuffle	Job::setPartitionerClass	Specify how the intermediate key/value pairs are shuffled to different reducers.
Reduce	Job::setGroupingComparatorClass	Specify how the key/value pairs are grouped in the reduce phase.
	Reducer::reduce	Users write their own `reduce` functions to perform the corresponding jobs.

a query, "select sum (adRevenue) from UserVisits group by countryCode, languageCode," to retrieve the total revenue of a country in a specific language. To process the query using MapReduce, a composite key consisting of countryCode and languageCode is created. The programmer needs to overwrite the default comparison and grouping functions to support comparison of composite key and partitioning. Moreover, if the UserVisits table is stored using different formats, the `RecordReader::next` function needs to be customized to parse the data split into key/value pairs.

Since Hadoop is the most popular MapReduce implementation, we use it as a reference implementation in discussing the other systems.

3.1.2. Cascading. Cascading (http://www.cascading.org/) is not strictly a MapReduce system implementation, but rather a query planner and scheduler implemented on top of Hadoop. It extends Hadoop by providing tools for defining complex processing logic using some atomic operators. Cascading aims to exploit the power of Hadoop without requiring users to know the programming details in order to reduce the overhead of migrating applications to Hadoop. It allows the user to link the atomic operators into a processing graph where each node represents an operator, and data flow is represented by the edges. A MapReduce job planner is applied to translate the processing graph into a set of MapReduce jobs. After one job completes, an event notification message is triggered to wake up the next job in the queue. To support fault tolerance, a failure trap can be created for the operators to back up the intermediate data. If an operation fails with exceptions and there is an associated trap, the offending data will be saved to the resource specified by the trap. In this way, the failed job can be resumed by another node by loading the data from the failure trap.

3.1.3. Sailfish. Sailfish [Rao et al. 2012] changes the transport layer between mappers and reducers in Hadoop. The main design contribution is improving the movement of data from mappers to reducers. The default method basically involves a sort-merge: the intermediate key/value pairs generated by each mapper are sorted locally, partitioned, sent to the corresponding reducers, and merged on the reduce side. In Sailfish, an abstraction called $\mathcal{I}$-file is introduced that facilitates batch transmission from mappers to reducers. The mappers directly append their map output key/values to $\mathcal{I}$-file, which will sort-merge them, and aggregate the pairs based on the partition key. Sorting the intermediate data becomes an independent phase and is optimized by $\mathcal{I}$-file itself using batching. In addition, Sailfish relieves users from the task of configuring the MapReduce parameters, because $\mathcal{I}$-file will automatically optimize the sorting of intermediate data.

3.1.4. Disco. Disco (http://discoproject.org/) is a distributed computing framework based on MapReduce developed by Nokia to handle massive numbers of moving objects in real-time systems. To track the locations of moving objects, the Disco Distributed File System (DDFS) is designed to support frequent updates. This is necessary since in other DFSs such as HDFS, files are never updated after insertion. DDFS is a tag-based file system that has no directory hierarchy; users can tag sets of objects with arbitrary labels and retrieve them via the tags. DDFS is optimized for storing small-size (4k) data items, such as user passwords or status indicators (for comparison, the size of data chunks in HDFS is always set to 64MB or 128MB). Disco supports a distributed index, called Discodex, which is distributed over the cluster nodes and stored in the DDFS. The index data are split into small files (called *ichunks*) that are replicated in the cluster and provide scalable and fault-tolerant indexing.

3.1.5. Skynet. Skynet (http://skynet.rubyforge.org/) is an open-source Ruby implementation of MapReduce. Besides the basic features of MapReduce systems, Skynet provides a peer recovery mechanism to reduce the overhead of the master node. The worker nodes monitor the status of each other; if one worker is down, another worker will detect it and take over the task. All workers can act as a master for any task at any time. In this way, Skynet can be configured as a fully distributed system with no permanent master node. This improves fault tolerance because the single point of failure is avoided.

3.1.6. FileMap. FileMap (http://mfisk.github.com/filemap/) is a lightweight implementation for Unix systems. It does not implement a DFS, and instead stores data in Unix files. It implements the processing logic via Unix scripts (written in Python and shell commands). Compared to the other MapReduce implementations, FileMap does not involve complex installation and configuration. It provides full functionality without requiring the user to install any special software. Users can write the `map` and `reduce` functions via scripts and submit to FileMap for processing. However, it also lacks some features available in other MapReduce systems, such as load balancing and fault tolerance. In FileMap, files are individually maintained by each node. If a job requires multiple files that reside on a number of nodes, the performance of the job is determined by those heavily loaded nodes. Moreover, the failure of a processing node results in the termination of the job, while in other MapReduce systems, a new node will take over the failed work.

3.1.7. Themis. Themis [Rasmussen et al. 2012] is a MapReduce implementation designed for small clusters where the probability of node failure is much lower. It achieves high efficiency by eliminating task-level fault tolerance. On the map side, the intermediate data are sorted and aggregated in memory. Once the memory is full, these data are directly sent to the reducers instead of being written to local disk. On the reduce side, the data shuffled from different mappers are partitioned and written to different files based on the partition key (the key/value pairs within the same partition are unsorted). Then, the reducer reads a whole partition, sorts the key/value pairs, and applies the reduce function to each key if the data of the partition can fit in memory. The performance of a MapReduce job can be significantly improved, but the entire job has to be re-executed in case of a node failure.

Table IV. Comparison of MapReduce Implementations

Name	Language	File System	Index	Master Server	Multiple Job Support
Hadoop	Java	HDFS	No	Name Node and Job Tracker	Yes
Cascading	Java	HDFS	No	Name Node and Job Tracker	Yes
Sailfish	Java	HDFS + $\mathcal{I}$-file	No	Name Node and Job Tracker	Yes
Disco	Python and Erlang	Distributed Index	Disco Server	No	No
Skynet	Ruby	MySQL or Unix File System	No	Any node in the cluster	No
FileMap	Shell and Perl Scripts	Unix File System	No	Any node in the cluster	No
Themis	Java	HDFS	No	Name Node and Job Tracker	Yes

3.1.8. Other Implementations. Oracle provides a MapReduce implementation by using its parallel pipelined table functions and parallel operations.[6] New DBMSs, such as Greenplum (http://www.greenplum.com), Aster (http://www.asterdata.com/), and MongoDB (http://www.mongodb.org), provide built-in MapReduce support. Some stream systems, such as IBM's SPADE, are also enhanced with the MapReduce processing model [Kumar et al. 2010].

3.1.9. System Comparison. Table IV shows the comparison of the various MapReduce implementations discussed in this section. We summarize these systems in different ways, including the language used to implement the system, the file system employed, the indexing strategies, the design of the master node, and the support for multiple jobs. Compared to Hadoop, other MapReduce implementations can be considered as lightweight MapReduce runtime engines. They are either built on top of Hadoop or implemented by script languages to simplify the development. They lack sophisticated administration tools, well-documented APIs, and many other Hadoop features (such as customized scheduler, various types of input/output format support, and compression techniques). Hadoop has become the most popular open-source MapReduce implementation and has been widely adopted.

3.2. High-Level Languages for MapReduce

The design philosophy of MapReduce is to provide a flexible framework that can be exploited to solve different problems. Therefore, MapReduce does not provide a query language, expecting the users to implement their customized `map` and `reduce` functions. While this provides considerable flexibility, it adds to the complexity of application development. To make MapReduce easier to use, a number of high-level languages have been developed, some of which are declarative (HiveQL [Thusoo et al. 2009], Tenzing [Chattopadhyay et al. 2011], JAQL [Beyer et al. 2009]); others are dataflow languages (Pig Latin [Olston et al. 2008]), procedural languages (Sawzall [Pike et al. 2005]), and Java library (FlumeJava [Chambers et al. 2010]); and still others are declarative machine learning languages (SystemML [Ghoting et al. 2011]). In this section, we review Pig Latin and HiveQL, the two most popular languages for MapReduce systems, and their corresponding systems, Pig and Hive, which automatically translate queries written in their respective languages into MapReduce jobs.

[6] http://blogs.oracle.com/datawarehousing/2009/10/in-database_map-reduce.html.

3.2.1. Pig Latin. Pig Latin [Olston et al. 2008] is a dataflow language that adopts a step-by-step specification method where each step refers to a data transformation operation. It supports a nested data model with user-defined functions and the ability to operate over plain files without any schema information. The details of these features are as follows:

(1) *Dataflow language.* Pig Latin is not declarative and the user is expected to specify the order of the MapReduce jobs. Pig Latin offers relational primitives such as LOAD, GENERATE, GROUP, FILTER, and JOIN, and users write a dataflow program consisting of these primitives. The order of the MapReduce jobs generated is the same as the user-specified dataflow, which helps users control query execution.

(2) *Operating over plain files.* Pig is designed to execute over plain files directly without any schema information, although a schema can also be optionally specified. The users can offer a user-defined parse function to Pig to specify the format of the input data. Similarly, the output format of Pig can also be flexibly specified by the user.

(3) *Nested data model.* Pig Latin supports a nested data model. The basic data type is Atom, such as an integer or string. Atoms can be combined into a Tuple, and several Tuples form a Bag. It also supports more complex data types such as Map⟨sourceIP, Bag(Tuple1, Tuple2, ...)⟩. This model is closer to the recursive data type in object-oriented programming languages and easier to use in user-defined functions.

(4) *UserDefined Functions (UDFs).* Due to the nested data model of Pig Latin, UDFs in Pig support nonatomic input parameters and can output nonatomic values. The UDF can be used in any context, while in SQL, the set-valued functions cannot be used in the SELECT clause. For example, the following script can be used to compute the top 10 adRevenue for each sourceIP:

```
groups = GROUP UserVisits BY sourceIP;
result = FOREACH groups GENERATE souceIP, top10 (adRevenue);
```

In this script, *top10* is a UDF that takes a set of adRevenue as input and outputs a set containing the top 10 adRevenues.

3.2.2. HiveQL. HiveQL is an SQL-like declarative language that is part of the Hive [Thusoo et al. 2009] system, which is an OLAP execution engine built on top of Hadoop. HiveQL features are the following:

(1) *SQL-like language*. HiveQL is an SQL-like query language that supports most of the traditional SQL operators such as SELECT, CREATE TABLE, UNION, GROUP BY, ORDER BY, and JOIN. In addition, Hive has three operators, MAP, CLUSTER BY, and REDUCE, which could integrate user-defined MapReduce programs into the SQL statement. As can be seen from the HiveQL query that follows, the CLUSTER BY operator is used to specify the intermediate key between `map` and `reduce`.

```
FROM (Map UserVisits USING 'python my_map.py'
        AS (sourceIP, adRevenue)
        From docs CLUSTER BY word) a
REDUCE sourceIP, totalRevenue USING 'python my_reduce.py'
```

HiveQL supports equijoin, semijoin, and outer join. Since Hive is a data warehouse system, the insert operation in HiveQL does not support inserting a tuple into an existing table; instead, it replaces the table by the output of a HiveQL statement.

(2) *Data model and type systems*. Hive supports the standard relational data model: the data are logically stored in tables, each of which consists of rows, and each row consists of columns, and a table may consist of several partitions. To implement

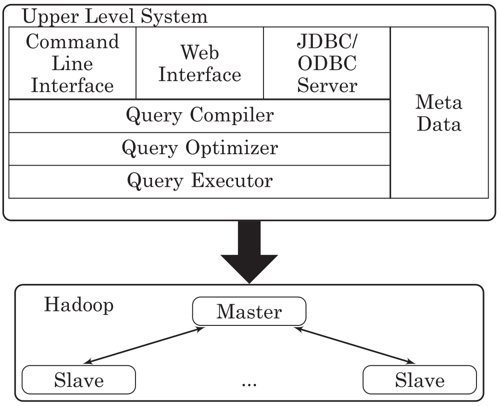


Fig. 2. Architecture of system like Pig and Hive.

this data model on top of MapReduce, the table is defined as a directory in DFS, while the partitions are subdirectories or files. Furthermore, the partition and table could also be divided into multiple buckets, each of which maps to a file in DFS. The bucket can be treated as the sample of the whole table and is usually used for debugging the Hive queries before they are fully run on the table. Hive provides Metastore for mapping the table to these partitions and buckets.

Columns in Hive can be either primitive types, such as integer, float, and string, or complex types, such as array, list, and struct. The complex types could be nested, for instance, list(map(string, string)). The schema information of the tables is also stored in Metastore. Hive has the flexibility to process plain files without transforming the data into tables. In this case, the user needs to implement a custom serializer and deserializer that will be used during the parsing of the data. Hive offers a default serializer that could determine the columns by user-specified delimiter or regular expressions.

3.2.3. Architecture of Hive and Pig. Though Pig and Hive support different languages, the architecture of these systems is similar, as shown in Figure 2. The upper level consists of multiple query interfaces such as command line interface, web interface, or JDBC/ODBC server. Currently, only Hive supports all these query interfaces. After a query is issued from one of the interfaces, the query compiler parses this query to generate a logical plan using the metadata. Then, the rule-based optimization, such as pushing projection down, is applied to optimize the logical plan. Finally, the plan is transformed into a DAG of MapReduce jobs, which are subsequently submitted to the execution engine one by one.

3.2.4. Comparison of Pig and Hive. The comparison of Pig and Hive is summarized in Table V. The most significant difference between Pig and Hive is the language. For example, the MapReduce job to compute the total adRevenue for each sourceIP in Algorithms 1 and 2 can be accomplished by the following Pig scripts:

```
groups = GROUP UserVisits BY sourceIP;
result = FOREACH groups GENERATE SUM(adRevenue), sourceIP;
```

In contrast, in HiveQL, this can be specified as follows:

```
SELECT sourceIP, sum(adRevenue)
FROM UserVisits
GROUPBY sourceIP;
```

Table V. Comparison between Pig and Hive

	Hive	Pig
Language	Declarative SQL-like	Dataflow
Data model	Nested	Nested
UDF	Supported	Supported
Data partition	Supported	Not supported
Interface	Command line, web, JDBC/ODBC server	Command line
Query optimization	Rule based	Rule based
Metastore	Supported	Not supported

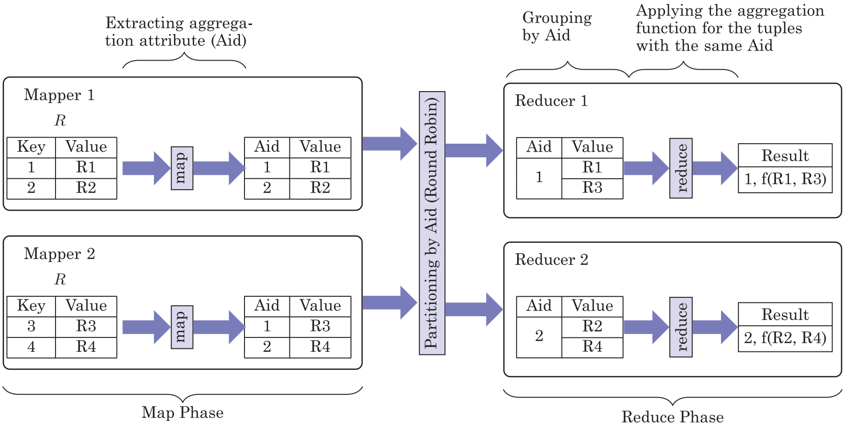


Fig. 3. Data flow of aggregation.

4. MAPREDUCE IMPLEMENTATION OF DATABASE OPERATORS

There have been a number of attempts to implement the typical database operators via MapReduce algorithms in order to improve the usefulness of these systems in data-intensive applications. Simple operators such as `select` and `project` can be easily supported in the `map` function, while complex ones, such as `theta-join` [Okcan and Riedewald 2011], `equijoin` [Blanas et al. 2010], `multiway join` [Wu et al. 2011; Jiang et al. 2011], and `similarity join` [Vernica et al. 2010; Metwally and Faloutsos 2012; Afrati et al. 2012], require significant effort. In this section, we discuss these proposals.

The projection and selection can be easily implemented by adding a few conditions to the `map` function to filter the unnecessary columns and tuples. The implementation of aggregation was discussed in the original MapReduce article. Figure 3 illustrates the data flow of the MapReduce job for the aggregation operator. The mapper extracts an aggregation key (Aid) for each incoming tuple (transformed into key/value pair). The tuples with the same aggregation key are shuffled to the same reducers, and the aggregation function (e.g., `sum`, `min`) is applied to these tuples.

Join operator implementations have attracted by far the most attention, as it is one of the more expensive operators, and a better implementation may potentially lead to significant performance improvement. Therefore, in this section, we focus our discussion on the join operator. We summarize the existing join algorithms in Figure 4.

4.1. Theta-Join

Theta-join (θ-join) [Zhang et al. 2012a] is a join operator where the join condition θ is one of $\{<, \leq, =, \geq, >, \neq\}$. As a baseline, let us first consider how a binary (natural) join of relations $R(A, B)$ and $S(B, C)$ can be performed using MapReduce. Relation R

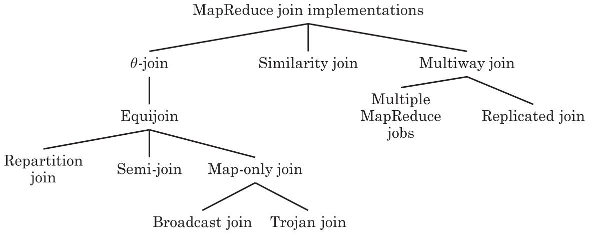


Fig. 4. Join implementations on MapReduce.

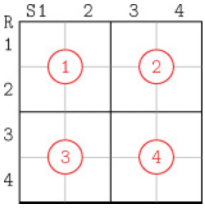


Fig. 5. Matrix-to-reducer mapping for cross-product.

is partitioned and each partition is assigned to a set of mappers. Each mapper takes tuples $\langle a, b\rangle$ and converts them to a list of key/value pairs of the form $(b, \langle a, R\rangle)$, where the key is the join attribute and the value includes the relation name R. These key/value pairs are shuffled and sent to the reducers so that all pairs with the same join key value are collected at the same reducer. The same process is applied to S. Each reducer then joins tuples of R with tuples of S (the inclusion of relation name in the value ensures that tuples of R or S are not joined with each other).

To efficiently implement theta-join on MapReduce, the $|R| \times |S|$ tuples should be evenly distributed on the r reducers, so that each reducer generates about the same number of results: $\frac{|R|\times|S|}{r}$. To achieve this goal, a randomized algorithm, 1-Bucket-Theta algorithm, was proposed [Okcan and Riedewald 2011] that evenly partitions the join matrix into buckets (Figure 5) and assigns each bucket to only one reducer to eliminate duplicate computation while also ensuring that all the reducers are assigned the same number of buckets to balance the load. In Figure 5, both tables R and S are evenly partitioned into four parts, resulting in a matrix with 16 buckets that are grouped into four regions. Each region is assigned to a reducer.

Figure 6 illustrates the dataflow of the theta-join when θ equals "$\neq$" for the case depicted in Figure 5. The map and reduce phases are implemented as follows:

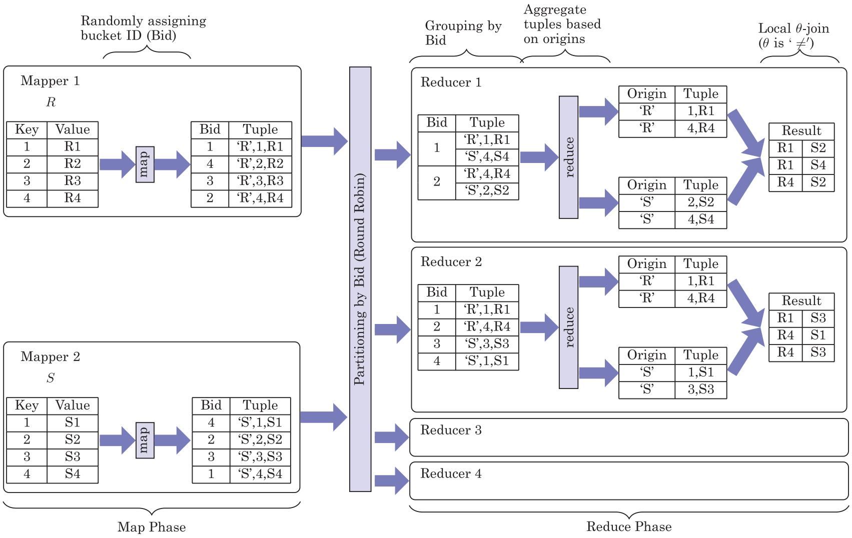
(1) *Map*. On the map side, for each tuple from R or S, a row id or column id (call it *Bid*) between 1 and the number of regions (4 in the previous example) is randomly selected as the map output key, and the tuple is concatenated with a tag indicating the origin of the tuple as the map output value. The *Bid* specifies which row or column in the matrix (of Figure 5) the tuple belongs to, and the output tuples of the `map` function are shuffled to all the reducers (each reducer corresponds to one region) that intersect with the row or column.

(2) *Reduce*. On the reduce side, the tuples from the same table are grouped together based on the tags. The local theta-join computation is then applied to the two partitions. The qualified results ($R.key \neq S.key$) are output to storage. Since each bucket is assigned to only one reducer, no redundant results are generated.

   In Figure 5, there are 16 buckets organized into four regions; there are four reducers in Figure 6, each responsible for one region. Since Reducer 1 is in charge of region 1, all R tuples where Bid $=$ 1 or 2 and S tuples with Bid $=$ 1 or 2 are sent to it. Similarly, Reducer 2 gets R tuples with Bid $=$ 1 or 2 and S tuples with Bid $=$ 3 or 4. Each reducer partitions the tuples it receives into two parts based on the origins and joins these parts.

4.2. Equijoin Operator

Equijoin is a special case of θ-join where θ is "$=$". The strategies for MapReduce implementations of the equijoin operator follow earlier parallel database implementations



Key	Value
1	R1
2	R2
3	R3
4	R4

Bid	Tuple
1	'R',1,R1
4	'R',2,R2
3	'R',3,R3
2	'R',4,R4

Key	Value
1	S1
2	S2
3	S3
4	S4

Bid	Tuple
4	'S',1,S1
2	'S',2,S2
3	'S',3,S3
1	'S',4,S4

Bid	Tuple
1	'R',1,R1 'S',4,S4
2	'R',4,R4 'S',2,S2

Origin	Tuple
'R'	1,R1
'R'	4,R4

Origin	Tuple
'S'	2,S2
'S'	4,S4

Result	
R1	S2
R1	S4
R4	S2

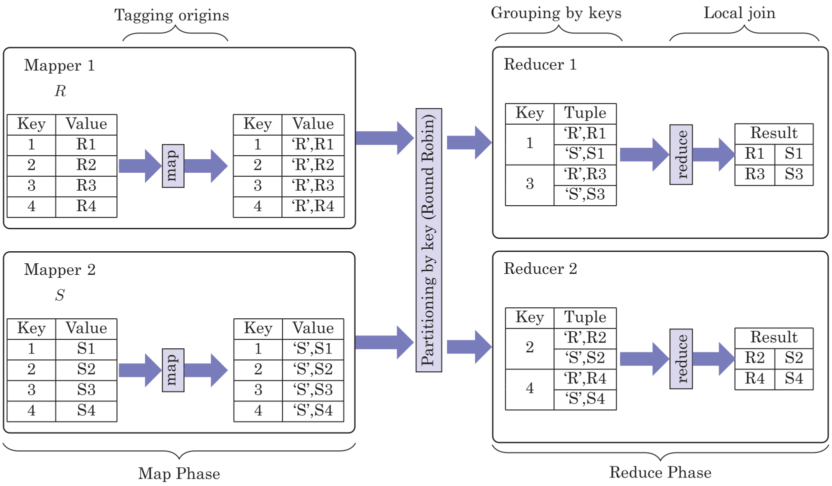
Bid	Tuple
1	'R',1,R1
2	'R',4,R4
3	'S',3,S3
4	'S',1,S1

Origin	Tuple
'R'	1,R1
'R'	4,R4

Origin	Tuple
'S'	1,S1
'S'	3,S3

Result	
R1	S3
R4	S1
R4	S3

Fig. 6. Data flow of theta-join (theta equals "≠").



Key	Value
1	R1
2	R2
3	R3
4	R4

Key	Value
1	'R',R1
2	'R',R2
3	'R',R3
4	'R',R4

Key	Value
1	S1
2	S2
3	S3
4	S4

Key	Value
1	'S',S1
2	'S',S2
3	'S',S3
4	'S',S4

Key	Tuple
1	'R',R1 'S',S1
3	'R',R3 'S',S3

Result	
R1	S1
R3	S3

Key	Tuple
2	'R',R2 'S',S2
4	'R',R4 'S',S4

Result	
R2	S2
R4	S4

Fig. 7. Data flow of repartition join.

[Schneider and Dewitt 1989]. Given tables R and S, the equijoin operator creates a new result table by combining the columns of R and S based on the equality comparisons over one or more column values. There are three variations of equijoin implementations (Figure 4): repartition join, semijoin-based join, and map-only join (joins that only require map side processing).

4.2.1. Repartition Join. Repartition Join [Blanas et al. 2010] is the default join algorithm for MapReduce in Hadoop. The two tables are partitioned in the map phase, followed by shuffling the tuples with the same key to the same reducer that joins the tuples. As shown in Figure 7, repartition join can be implemented as one MapReduce job. Since this is the most basic equijoin implementation in MapReduce, we discuss it in some

Table VI. Notations for Cost Analysis

Symbols	Definition
$c_{hdfsRead}$	The I/O cost of reading files in HDFS
c_{local}	The I/O cost of processing the data locally
$c_{shuffle}$	The cost of network shuffling per byte
$c_{hdfsWrite}$	The I/O cost of writing files to HDFS
$\|R\|$	Number of tuples in table R
$\|Tuple_R\|$	Average size of each tuple in table R
$KeyValue_R$	The size of key/value pair generated by the mapper for table R
Sel_R	The selectivity of R in the map function
Sel_{join}	The join selectivity
m	Number of mappers
B	Chunk size of multiway sorting
$Size_{output}$	Average size of join output tuple

detail and analyze the total I/O cost of each step (in the cost analysis, we assume that the data are uniformly distributed on the join key, and the CPU costs are ignored):

(1) *Map*. Two types of mappers are created in the map phase, each of which is responsible for processing one of the tables. For each tuple of the table, the mapper outputs a key/value pair $(K, \langle T, V\rangle)$, where K is the join attribute value, V is the entire tuple, and T is the tag indicating the source relation of the key/value pair. More specifically, the map phase consists of the following steps:

(a) Scanning the data from HDFS and generating the key/value pair. Suppose the I/O cost of reading files in HDFS is $c_{hdfsRead}$ per byte, the number of tuples in table R is $|R|$, and the size of each tuple is $|Tuple_R|$ (Table VI shows the detailed notations used in this section). Then the cost of scanning table R is

$$C_{scan} = c_{hdfsRead} \times |R| \times |Tuple_R|.$$

(b) Sorting the map output (i.e., set of key/value pairs). On the map side, the output of each mapper needs to be sorted before being shuffled to the reducers. Let the chunk size of multiway sorting be B and the selectivity of R in the map function be Sel_R. Suppose the I/O cost of processing the data locally is c_{local} per byte. Let $KeyValue_R$ denote the size of key/value pairs generated by the mapper. If m mappers are employed, the size of the data generated by each map task for R is:

$$Size_{map-output} = (|R|/m) \times Sel_R \times KeyValue_{R.}$$

Thus, the cost of this phase is:

$$C_{sort-map} = c_{local} \times (m \times Size_{map-output} \times log_B(Size_{map-output}/(B+1))).$$

(2) *Shuffle*. After the map tasks are finished, the generated data are shuffled to the reduce tasks. If the cost of network shuffling is $c_{shuffle}$ per byte, the cost of the shuffling phase is:

$$C_{shuffle} = m \times Size_{map-output} \times c_{shuffle.}$$

(3) *Reduce*. The reduce phase includes the following steps:

(a) *Merge*. Each reducer merges the data that it receives using the sort-merge algorithm. Assume that the memory is sufficient for processing all sorted runs together. Then the reducer only needs to read and write data into local file systems once. Thus, the cost of merging is:

$$C_{merge-reduce} = 2 \times c_{local} \times (m \times Size_{map-output}).$$

Table VII. Summarization of Cost Analysis

Phases	I/O Cost
Map	$C_{scan} = \times \lvert R \rvert \times \lvert Tuple_R \rvert$
	$Size_{map-output} = (\lvert R \rvert / m) \times Sel_R \times KeyValue_R$ $C_{sort-map} = c_{local} \times (m \times Size_{map-output} \times log_B(Size_{map-output}/(B+1)))$
Shuffling	$C_{shuffle} = m \times R_{map-output} \times c_{shuffle}$
Reduce	$C_{merge-reduce} = 2 \times c_{local} \times (m \times Size_{map-output})$
	$C_{reduce-join} = 0$
	$C_{reduce-output} = c_{hdfsWrite} \times (\lvert R \rvert \times Sel_R \times \lvert S \rvert \times Sel_S \times Sel_{join} \times Size_{output})$

Table VIII. Map-Reduce-Merge Functions

Map	$(k1, v1)_r \rightarrow list(k2, v2)_r$
Reduce	$(k2, list(v2))_r \rightarrow (k2, list(v3))_r$
Merge	$(k2, list(v3))_r, (k2, list(v3))_s \rightarrow list(v4)$

(b) *Join*. After the sorted runs are merged, the reducer needs two phases to complete the join. First, the tuples with the same key are split into two parts based on the tag indicating its source relation. Second, the two parts are joined locally. Assuming that the number of tuples for the same key are small and can fit in memory, this step only needs to scan the sorted run once. Since the data generated from the preceding merge phase can be directly fed to this join phase, the cost of scanning this sorted run is ignored:

$$C_{reduce-join} = 0.$$

(c) *Write to HDFS*. Finally, the results generated by the reducer should be written back to the HDFS. If the I/O cost of writing files to HDFS is $c_{hdfsWrite}$ per byte and the join selectivity is Sel_{join}, the cost of this final stage is:

$$C_{reduce-output} = c_{hdfsWrite} \times (|R| \times Sel_R \times |S| \times Sel_S \times Sel_{join} \times Size_{output}),$$

where $Size_{output}$ denotes the average size of join output tuple. The cost analysis is summarized in Table VII. More detailed analysis can be found in Afrati and Ullman [2010], Li et al. [2011], and Wu et al. [2011].

The repartition join algorithm discussed earlier relies on a tag that is generated and inserted into each key/value pair to indicate its source. This ensures that once the tuples are partitioned based on the keys, those with the same key from different sources can be joined. However, it also degrades the performance of MapReduce since more data (tags) are shuffled to the reducers. An alternative approach, called Map-Reduce-Merge [Yang et al. 2007], has been proposed to eliminate this drawback.

Map-Reduce-Merge adds a *merge* phase after the reduce phase. The functions of each phase in Map-Reduce-Merge are shown in Table VIII and discussed next.

Suppose there are two data sources, R and S. The map phase and reduce phase in Map-Reduce-Merge are similar to the original algorithm described earlier. The mapper's output data are shuffled to different reducers according to the partition key and sorted, hashed, or grouped by the reducers. However, reducers do not write the output to HDFS; instead, they store the output locally. Then, at merge phase, according to the partition selector, the reducer outputs are shuffled to the *mergers*. The data from different reducers are merged and written to HDFS.

In the merge phase, each data source can be individually preprocessed prior to merging in order to implement a specific function. For example, if hash-join is preferred, the processor for the smaller relation, say, S, can build a hash table on its data (in this case, no processing is required for R). The merger can use this hash table to join R with S.

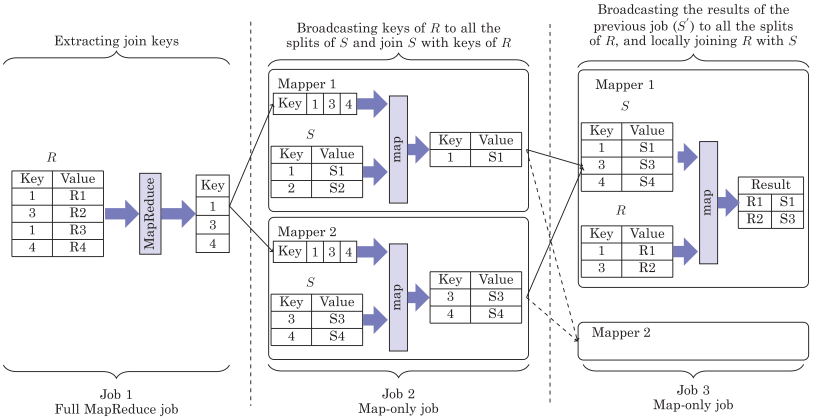


Fig. 8. Data flow of semijoin-based join.

Adding the merge phase makes it easier to program relational operator implementations. Repartition join using Map-Reduce-Merge can simply be implemented as follows:

(1) *Map*. This is identical to the original repartition join discussed previously.
(2) *Reduce*. In the reduce phase, data pulled from the *mappers* for the same table are automatically merged into one sorted run.
(3) *Merge*. The *merger* reads two sorted runs from the two tables in the same key range and joins them.

Map-Reduce-Merge works better for more complex queries (such as the queries that first need a group-by over a table, followed by a join of the two tables) because it has one more phase than MapReduce. The group-by is accomplished by the reducer, while the join is accomplished by the merger.

4.2.2. Semijoin-Based Join. Semijoin-based join has been well studied in parallel database systems (e.g., Bernstein and Chiu [1981]), and it is natural to implement it on MapReduce [Blanas et al. 2010]. The semijoin operator implementation consists of three MapReduce jobs (Figure 8). The first is a full MapReduce job that extracts the unique join keys from one of the relations, say, R, where the map task extracts the join key of each tuple and shuffles the identical keys to the same reducer, and the reduce task eliminates the duplicate keys and stores the results in DFS as a set of files $(u_0, u_1, \ldots, u_k)$. The second job is a map-only job that produces the semijoin results $S' = S \ltimes R$. In this job, since the files that store the unique keys of R are small, they are broadcast to each mapper and locally joined with the part of S (called *data chunk*) assigned to that mapper. The third job is also a map-only job where S' is broadcast to all the mappers and locally joined with R. This algorithm can be further improved. In the third phase, the results of $(S \ltimes R)$ are joined with every chunk of R. However, only a small portion of $(S \ltimes R)$ has common join keys with a specific data chunk of R, say, R_i. Thus, most comparisons are unnecessary and can be eliminated. Consequently, in the first job, the mappers would partition the join keys of R_i into separate files $(R_i.u_0, R_i.u_1, \ldots, R_i.u_k)$. In the second job, S would be joined with each $R_i.u_j$, and all the joined results between S and $R_i.u_j$ would be written to the same file named S_{R_i}, producing $(S \ltimes R_i)$. In the final job, each data chunk R_i would only need to be joined with S_{R_i}.

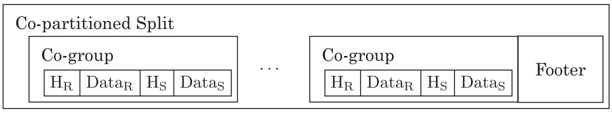


Fig. 9. Trojan join indexed split.

Table IX. Comparison of Equijoin Algorithms

Algorithm	Advantage	Constraint
Repartition Join	It is the most general join method	It is not efficient enough in some circumstances
Semijoin-Based Join	It is efficient when the semijoin result is small	It requires several MapReduce jobs. The semijoin result should be computed first
Broadcast Join	It only involves a map-only job	One table should be small enough to fit into memory
Trojan Join	It only involves a map-only job	The relations should be copartitioned in advance

4.2.3. Map-Only Join. In the aforementioned join algorithms, the input data first need to be partitioned based on the join key and then shuffled to the reducers according to the partition function. If the inner relation is much smaller than the outer relation, then shuffling can be avoided as proposed in Broadcast Join [Blanas et al. 2010]. The algorithm has only the map phase similar to the third job of the semijoin-based algorithm (Figure 8). Assuming S is the inner and R is the outer relation, each mapper loads the full S table to build an in-memory hash and scans its assigned data chunk of R (i.e., R_i). The local hash-join is performed between S and R_i.

Map-only join can also be used if the relations are already copartitioned based on the join key. In Hadoop++ [Dittrich et al. 2010], Trojan join is implemented to join the copartitioned tables. In this case, for a specific join key, all tuples of R and S are colocated on the same node. The scheduler loads the copartitioned data chunks of R and S in the same mapper to perform a local join, and the join can be processed entirely on the map side without shuffling the data to the reducers. This copartitioning strategy has been adopted in many systems such as Hadoop++ [Dittrich et al. 2010].

Copartitioning prior to join is implemented by one MapReduce Job: the `map` function implements the same logic as the repartition join, while the `reduce` function takes these key-list pairs as input and transforms each of these key-list pairs into a cogroup by merging the tuples from both tables R and S into one list if they have the same key. As shown in Figure 9, each cogroup contains the tuples with the same key from both tables, and the tuples from different tables are distinguished by their tags. Finally, the copartitioned splits, each of which consists of several cogroups, are written to HDFS. At this point, each copartitioned split contains all the tuples from both R and S for a specific join key, and the join query can be directly processed by the mappers.

Table IX shows the comparison of the equijoin algorithms. Each algorithm has its advantages and constraints. To obtain the best implementation, the statistics of the data must be collected, and the cost of each algorithm should be estimated using a similar analysis method as Section 4.2.1.

4.3. Similarity Join

In many applications, the join condition may be inexact, and it is sufficient for the attribute values to be "similar." For this purpose, the similarity of two join attributes ($R.A$ and $S.B$) is computed by a function, $sim(R.A, S.B)$, and a threshold δ is defined. Typically, $sim(R.A, S.B)$ simply computes the difference between the values of the

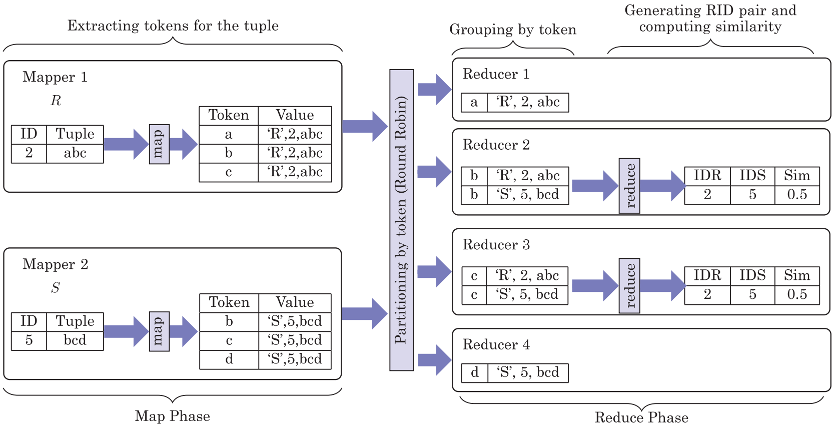


Fig. 10. Data flow for stage 2 of similarity join.

two attributes, but other more sophisticated similarity functions are also possible. Similarity join is then defined as a join whose predicate is $sim(R.A, S.B) \geq \delta$.

Similarity calculation on numeric attributes is straightforward. In the case of string-valued attributes, the step is to convert strings into a set of tokens in order to simplify the similarity computation. A number of tokenization approaches can be adopted, such as dividing the string into words (e.g., string "similarity join work" would be tokenized into set {similarity, join, work}) or computing the q-grams, which are overlapping strings of length q (e.g., 3-gram set for the string "attribute" is {att, ttr, tri, rib,ibu, but, ute}). In the following, "token" refers to the result of this phase.

The naive method to compute similarities is to carry out the computation between all the tuples in R and all the tuples in S, but this incurs many unnecessary computations. A prefix filtering method [Bayardo et al. 2007; Chaudhuri et al. 2006; Xiao et al. 2008] can be applied to reduce the number of involved tuples by pruning some pairs without computing their similarity. For example, after tokenization, if tuple r_1 consists of tokens {a,b,c,d}, tuple s_1 consists of {e,g,f}, and the length of prefix is set to 2, the prefix for r_1 is then {a,b} and the prefix of s_1 is {e,g}. Since these two records do not share a common token in their prefixes, they are not similar and can be pruned.

Similarity join can be parallelized as a sequence of MapReduce jobs [Vernica et al. 2010; Metwally and Faloutsos 2012; Afrati et al. 2012]. We review the first of these [Vernica et al. 2010] as one example to demonstrate the idea. The parallel similarity join consists of three stages:

(1) *Token ordering*. The first stage is to sort the tokens of the join attribute across all the tuples in tables R and S. Sorting removes the duplicate tuples, which have the same set of tokens but in a different sequence (e.g., tuples {a,b} and {b,a} need to be recognized as the same tuple in the remainder of the algorithm). Frequencies of token occurrences in each table are computed, starting with the smaller relation, say, S. This process is implemented as a word count MapReduce job that computes the count of each token for the table and sorts the results based on the count. The result of the first job is used to prune table R: The tuples in R that do not have any tokens appearing in table S are deleted. The tokens are then sorted based on the frequencies in the remaining tuples of both R and S.
(2) *Row id-pair generation*. The second stage is the core of similarity join, which is implemented as a MapReduce job illustrated in Figure 10. The `map` function scans the tables and creates a key/value pair for every token in the tuple's prefix. The

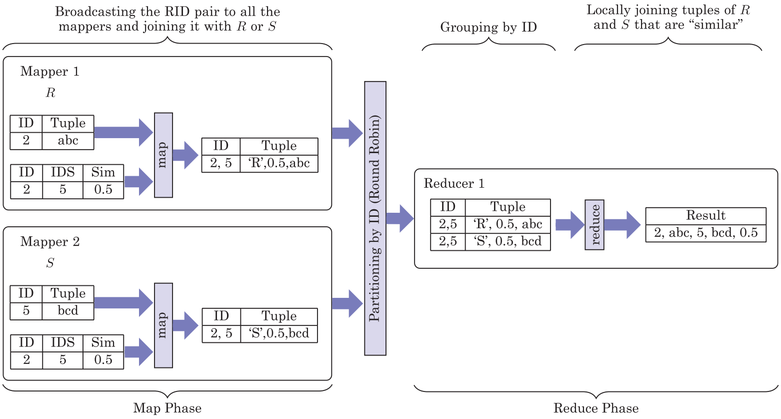


Fig. 11. Data flow for stage 3 of similarity join.

output key is the token, and the value is the same as the input, namely, (*tag, RID, tuple*), where RID is the row id. The intermediate key/value pairs are sorted based on the key and shuffled to the reducers. The tuples that share common tokens in the prefix are sent to the same reducers, which join the tuples with the same common token and compute the similarity of each tuple pair. The output is a triple (ID_R, ID_S, α), where ID_R and ID_S are RIDs of the two corresponding tuples and α is their similarity. The triples where $\alpha < \delta$ are pruned. Finally, duplicate result pairs are eliminated.

(3) *Tuple join*. The final stage is to join the tuples in R and S using the triples generated in the second stage. This stage can be implemented by one MapReduce job as shown in Figure 11. First, the set of generated triples are broadcast to all the mappers. The mappers scan R and S tables and the triples from the previous stage and use the RID pair as the map output key to partition the data. The reducer uses the RID pair to join the original tuples from R and S. For example, for pair (2, 5, 0.5) in Figure 11, the mapper generates two tuples, ⟨(2, 5), (‘R’,0.5, abc)⟩ and ⟨((2, 5), (‘*S*’,0.5, bcd)⟩. These two tuples are shuffled to the same reducer and the result is computed as (2, abc, 5, bcd, 0.5).

In addition, algorithms have been proposed for finding the top-*k* most similar pairs [Kim and Shim 2012] and kNN join [Zhang et al. 2012b; Lu et al. 2012], in which the similarity between the tuples is defined as their distance in an $N-$dimensional space, and only the *k*-nearest (the *k*-most similar) tuples in S for each tuple in R are returned.

4.4. Multiway Join

The implementation of a multiway join is more complex than a binary join. It can be implemented using either multiple MapReduce jobs (one for each join) or only one MapReduce job (the replicated join). In this section, we review these two processing strategies for multiway equijoins. We then discuss a generalization to multiway θ-join.

4.4.1. Multiple MapReduce Jobs. Multiway join can be executed as a sequence of equijoins; for example, $R \bowtie S \bowtie T$ can be implemented as $(R \bowtie S) \bowtie T$. Each of these joins is performed by one MapReduce job. The result of each MapReduce job (e.g., $R \bowtie S$) is treated as input for the next MapReduce job. As usual, different join orders lead to different query plans with significantly different performance. To find the best join order, we need to collect the statistics of the data (e.g., histograms appropriate for MapReduce [Jestes et al. 2011]) and estimate the processing cost of each possible plan using a cost

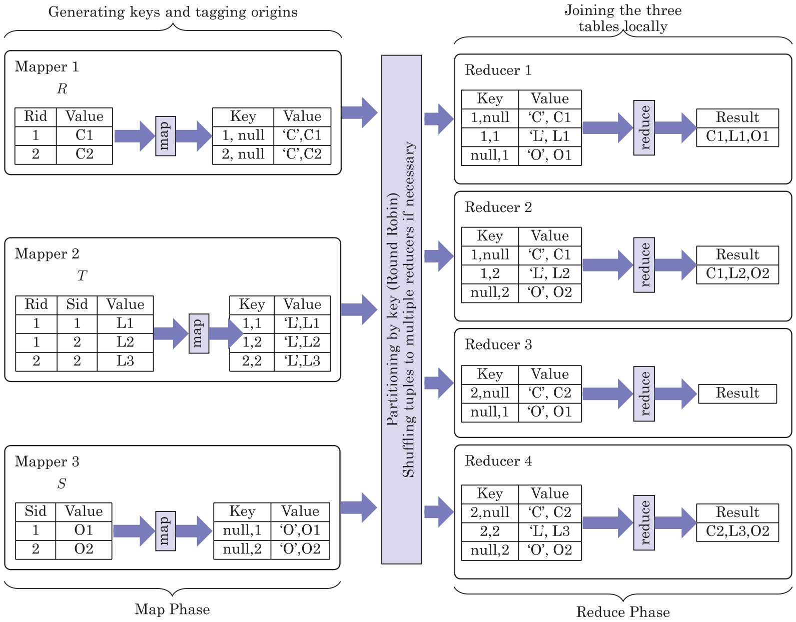


Fig. 12. Dataflow for replicated join.

model. In Section 4.2.1, we analyzed the cost of binary join using the repartition approach. The same estimation process can be applied to the other join approaches (e.g., semijoin). Using the model for binary join, we can calculate the cost of the multiway join step by step.

Many plan generation and selection algorithms that were developed for relational DBMSs can be directly applied here to find the optimal plan. These optimization algorithms can be further improved in a MapReduce system [Wu et al. 2011]; in particular, more elaborate algorithms may be deployed for two reasons. First, the relational optimization algorithms are designed to run fast to balance query optimization time versus query execution time. MapReduce jobs usually run longer than relational queries, justifying more elaborate algorithms (i.e., longer query optimization time) if they can reduce query execution time. Second, in most relational DBMSs, only left-deep plans are typically considered to reduce the plan search space and to pipeline the data between operators. There is no pipeline between the operators in the original MapReduce, and, as we indicated previously, query execution time is more important. Thus, the bushy plans are often considered for their efficiency.

4.4.2. Replicated Join. Replicated join [Jiang et al. 2011; Afrati and Ullman 2010] performs multiway join as a single job. We demonstrate the operation of replicated join using the three-way join $R \bowtie S \bowtie T$. The dataflow of the replicated join for this query is shown in Figure 12.

To support replicated join, in the map phase, the *Partitioner* module is modified. Suppose the *Sids* (the keys of S) are partitioned into N_s groups, *Rids* (the keys of R) are partitioned into N_r groups, and the number of reducers in the system is $N_r \times N_s$. The partition key of the MapReduce job is the pair $(Rid\%N_r, Sid\%N_s)$. Each tuple of T is shuffled to a specific reducer since it contains both *Rid* and *Sid*. However, the partition keys of tuples in S have the format $(null, Sid\%N_s)$, so each such tuple is shuffled to N_r reducers (namely, they are replicated at multiple reducers). Similarly, the tuples of table

R are shuffled to N_s reducers. In this way, each reducer can receive all the necessary tuples for a (Rid, Sid) pair and join tuples from three tables locally. Compared to the implementation with multiple MapReduce jobs, replicated join reduces the cost of writing the intermediate results to HDFS by reducing the number of jobs. Using a cost analysis similar to the one described in Section 4.2.1, we compute the cost saved by the replicated join as:

$$C_{less\_written} = |S| \times |R| \times Sel_S \times Sel_R \times Sel_{join} \times Size_{output} \times c_{hdfsWrite.}$$

However, the replicated join shuffles and sorts more data compared to the basic strategy, so its cost is

$$C_{more\_shuffled} = ((N_s - 1) \times |R| \times KeyValue_R + (N_r - 1) \times |S| \times KeyValue_S) \times (c_{shuffle} + c_{sort}).$$

Thus, replicated join is preferred only if $C_{less\_written} > C_{more\_shuffled}$.

In replicated join, given the number of computation nodes, a proper allocation of N_r and N_s can minimize the shuffled data. For instance, in the previous example query, if R is much smaller than S, then N_r is set to 1. As a result, the data of R are replicated to all the reducers, while the data shuffled for S are the same as join using multiple jobs.

In many MapReduce implementations, jobs are scheduled using a FIFO strategy. Although the join implementation with multiple jobs scans the same amount of data as replicated join, the second job of the multiple jobs method has to wait until the first job is finished, which reduces the response time between the submission of the query and the collection of the final result. For example, suppose that each table needs 10 mappers, all the mappers run for the same time, and there are 15 nodes in the system. The normal join method needs three rounds to finish the scan, while the replicated join only needs two rounds $((10 + 10 + 10)/15)$. In addition, since the reducers start to pull the data once a mapper is finished, the data generated from the mappers in the first round are shuffled to the reducers concurrently with the execution of the mappers in the second round. Thus, although replicated join shuffles more data, it improves the join performance by reducing the intermediate results and better exploiting the parallelism.

A special case of replicated join is *star join,* where all join conditions are on the same attributes (e.g., tables $R(x, y)$, $S(x, z)$, and $T(x, u)$ are joined on attribute x). By setting the map output key to be the join attribute, the star join can be implemented by one MapReduce job. In case the data are skewed, a load-balanced algorithm can be used [Koutris and Suciu 2011]. The algorithm first estimates the frequent values in R, S, and T. If a value (e.g., a) is frequent in R, the tuples $(x = a)$ for tables S and T are broadcast to all the reducers, but the tuples $(x = a)$ for table R are partitioned to the reducers by a different strategy (such as round robin). The join results for $x = a$ are the combinations of the partial results of all the reducers.

4.4.3. Generalization to θ-Join. In the previous two subsections, we discussed the multiway equijoin where the join condition is restricted to "=". An improvement to the these works is multiway θ-join where $\theta \in \{<, \leq, =, \geq, >, \neq\}$. Similar to multiway equijoin, the multiway θ-join can also be implemented in two ways [Zhang et al. 2012a]: (1) each θ-join is processed by one MapReduce job (as discussed in cross-product), and a cost model is proposed to decide the best join order; or (2) the joins are implemented in one MapReduce job. To implement the multiway θ-join in one MapReduce job, an extended version of the one-bucket join [Okcan and Riedewald 2011] discussed earlier is adopted. In the one-bucket join, the join matrix has only two dimensions, which are partitioned to k reducers. In contrast, here the join matrix has r dimensions corresponding to the number of relations involved in the multiway join. The r-dimensional matrix is evenly partitioned to k reducers, and the data of each relation are shuffled to a reducer if it overlaps with the partitions assigned to that reducer.

In addition, a hybrid method has also been proposed: A multiway θ-join is implemented by a few MapReduce jobs, and one MapReduce job may process the θ-join for more than two relations. For example, the implementation of a multiway θ-join $R \bowtie_\theta S \bowtie_\theta T \bowtie_\theta W$ may consist of two MapReduce jobs, where the first one computes $Result_1 = R \bowtie_\theta S \bowtie_\theta T$, and the second one computes $Result_1 \bowtie_\theta W$. The search space of this hybrid method is much larger than the case when each MapReduce job executes one θ-join, and the proposed techniques explore a majority of the possible plans to achieve the near-optimal performance.

5. DBMS IMPLEMENTATIONS ON MAPREDUCE

The implementation of database operators using the MapReduce framework facilitates the development of efficient full-fledged MapReduce-based DBMSs. In their simplest form, these consist of only an SQL parser, which transforms the SQL queries into a set of MapReduce jobs. Examples include Hive [Thusoo et al. 2009] and Google's SQL translator [Chattopadhyay et al. 2011]. In a more complete form, a MapReduce-based DBMS natively incorporates existing database technologies to improve performance and usability, such as indexing, data compression, and data partitioning. Examples include HadoopDB [Abouzeid et al. 2009], Llama [Lin et al. 2011], and Cheetah [Chen 2010]. Some of these systems follow the traditional relational DBMS approach of storing data row-wise (e.g., HadoopDB) and are, therefore, called *row stores*. Others (e.g., Llama) store data column-wise and are called *column stores*. It is now generally accepted that the column-wise storage model is preferable for analytical applications that involve aggregation queries because (1) the values in each column are stored together and a specific compression scheme can be applied for each column, which makes data compression much more effective, and (2) it speeds up the scanning of the table by avoiding access to the columns that are not involved in the query [Stonebraker et al. 2005]. Column-oriented storage has been widely applied to many large-scale distributed systems (such as Dremel [Melnik et al. 2011]), and they can significantly improve the performance of MapReduce [Floratou et al. 2011]. In addition to pure row stores and column stores, some systems adopt a hybrid storage format (e.g., Cheetah): the columns of the same row are stored in the same data chunk, but the format of each data chunk is column oriented. An improvement to this format is to combine a few columns that are frequently accessed together into a column group, which reduces the CPU cost of reconstructing the tuples. Furthermore, since MapReduce usually replicates each data chunk, different replicas can have different combinations of column groups so that the query can access the best suitable replica [Jindal et al. 2011].

A full DBMS implementation over MapReduce usually supports the following functions: (1) a high-level language, (2) storage management, (3) data compression, (4) data partitioning, (5) indexing, and (6) query optimization. In this section, we review the recent research work on building MapReduce-based DBMSs focusing on these functions.

5.1. HadoopDB

HadoopDB [Abouzeid et al. 2009] introduces the partitioning and indexing strategies of parallel DBMSs into the MapReduce framework. Its architecture consists of three layers. The top layer extends Hive to transform the queries into MapReduce jobs. The middle layer implements the MapReduce infrastructure and DFS and deals with caching the intermediate files, shuffling the data between nodes, and fault tolerance. The bottom layer is distributed across a set of computing nodes, each of which runs an instance of PostgreSQL DBMS to store the data. The following summarizes how HadoopDB supports the functions listed previously.

(1) *High-level language.* HadoopDB adopts HiveQL as its high-level query language. It transforms HiveQL queries into local plans executed on each MapReduce node. Since HadoopDB stores the data in DBMS instances on each node, the local plans are different than those generated by Hive (see earlier discussion). For instance, the local plan in HadoopDB might be a "select" query instead of the file scan that Hive performs.

(2) *Data partitioning.* HadoopDB horizontally partitions a table into *chunks* based on a given key. Even in the presence of skewed data distribution across keys, HadoopDB attempts to ensure uniform-sized chunks for better load balancing during query execution, which is different than MapReduce's default hash-partitioning that ignores skew. Furthermore, it can copartition several tables together based on the join keys, so that the tables can be joined by a map-only job like the Trojan join.

(3) *Storage management.* Each chunk obtained as a result of partitioning is assigned to one node and stored in the PostgreSQL instance at that node.

(4) *Data compression.* Data compression is not used in HadoopDB since it uses a traditional row-oriented database system for storage, while data compression is better suited for column stores (in the latest demo by Abouzied et al. [2010], the data compression can be achieved when they use a column-oriented database instead of PostgreSQL, but in this section we only discuss the case when the row-oriented database is used).

(5) *Indexing.* Building of local indexes for each data chunk is left to PostgreSQL; these indexes can later be used during query processing.

(6) *Query optimization.* HadoopDB performs both global optimization and local optimization of submitted queries. For global optimization, it inherits the rule-based strategies from Hive, such as pushing predicates closer to table scans and pruning unused columns early. The map phase utilizes PostgreSQL for query processing, while the shuffle and the reduce phases are handled by MapReduce infrastructure. To fully utilize the PostgreSQL instances on the nodes, as much of the work as possible is performed in the map phase. For example, if the tables are copartitioned on the join key, the join queries are transformed to a map-only job. Each database instance joins the tables stored on that node without shuffling any data over the network. When the tables are not copartitioned, HadoopDB has two implementations for the join query. The first implementation is similar to the repartition join: each mapper scans the part of the local table in the DBMSs at that node and shuffles the data to reducers, while the reducers join the tables without using the DBMS instance. The second implementation is similar to Trojan join: one MapReduce job is launched to repartition the two tables on the join key, and another map-only job is launched to join the two tables.

    Local optimization of plans on each node is performed by the PostgreSQL query optimizer. The local indexes can be used for index scan, index join, and so forth.

HadoopDB combines the advantages of both MapReduce and conventional DBMSs. It scales well for large datasets and its performance is not affected by node failures due to the fault tolerance of MapReduce. By adopting the copartitioning strategy, the join operator can be processed as a map-only job. Moreover, at each node, local query processing automatically exploits the functionality of PostgreSQL.

5.2. Llama

Llama [Lin et al. 2011] is a column store implementation on top of MapReduce with the following functionality:

(1) *High-level language.* Llama does not have an SQL-like query language and expects users to write the `map` and `reduce` functions. However, it changes the input format

of these functions to be column based. It also implements a specific interface called `LlamaInputs` to facilitate easy composition of join queries using column-wise storage.

(2) *Storage management.* Llama uses a specific file structure, CFile, as the storage unit. Each CFile is a DFS file that contains only the data of a single column. The CFile is split into several data chunks and each data chunk contains a fixed number of records, say, k. The location of each chunk is recorded so that it can be directly located. By default, the data of a CFile is sorted by the primary key of the table. To retrieve the column value of the n^{th} tuple, the $(n\%k)^{\text{th}}$ value in the $(n/k)^{\text{th}}$ chunk is returned.

(3) *Data partitioning.* Each table is vertically partitioned into several groups, and each group consists of multiple CFiles corresponding to the columns of that group. In a vertical group, all the CFiles are sorted by the primary key of the group. There are two types of vertical groups. The *basic group* includes the CFiles of all the columns in the table. The *primary-foreign key* (PF) group is defined to include the CFiles of the foreign key, primary key, and some other columns in the query predicates. CFiles in a PF group are sorted by the join key. This strategy enables a map-only join that greatly reduces the processing overhead. To facilitate join processing, PF groups are dynamically created based on the statistics of the query patterns. However, PF groups replicate the data of basic groups and incur additional storage costs. If the groups are not used for a long time, they are automatically discarded to reduce storage cost.

(4) *Data compression.* Llama compresses each chunk of the CFile. As noted earlier, since the values of a column may typically contain similar patterns, compression is effective. Furthermore, each column can have a specific data compression scheme that is best suited for the data in that column.

(5) *Indexing.* In each CFile, a chunk index is added to the end of each chunk, which maintains the summary information of data in the chunk.
An alternative compression and indexing technique in Llama is BIDS [Lu et al. 2013], which builds a bitmap index for each column. The advantage of BIDS is its compact size, especially for the columns with limited unique values. Since data can be recovered directly from BIDS, the index can be considered as a compression technique as well.

(6) *Query optimization.* Llama uses several optimization strategies. First, using an index, it filters CFiles and only scans those that are relevant to the query, thus reducing I/O costs. The second optimization relates to the column-wise storage model that it uses. In column store systems, it is necessary to employ a materialization process to concatenate the column values of the same tuple together. This can be achieved by Early Materialization (EM) or Late Materialization (LM). EM materializes the columns of a tuple at the beginning of the MapReduce job, while LM materializes the columns on demand. For instance, to process a join query, EM materializes all the necessary columns in the map phase, while LM only materializes the join keys and the columns that appear in selection predicates. In the reduce phase, LM retrieves the other columns for the tuples appearing in the query results. Compared to EM, LM scans less data in the map phase but incurs higher random read cost when retrieving the columns in the reduce phase. Llama uses a cost function to choose the appropriate strategy for a given query.

The use of the column-wise storage model allows the implementation of a multiway join in a single MapReduce job, thereby reducing the number of jobs and the total processing cost. For example, to process a join over a star schema that contains a high cardinality fact table (i.e., a large number of tuples) and a few lower cardinality but higher degree (i.e., high number of attributes) dimension tables, a map-only join is used to join the fact table with the dimension tables using the corresponding PF groups. Then, the intermediate results of the join are sent to the reducers to merge. In the reduce phase, the join results are combined based on the fact table's primary key.

5.3. Cheetah

Cheetah [Chen 2010] is a system specially designed for analysis applications that use a star schema or snowflake schema. Star schema characteristics were discussed earlier; in contrast, in a snowflake schema, one attribute can be represented by several dimension tables with parent-children relationships. Cheetah's functionality includes the following:

(1) *High-level language*. Cheetah defines a single-block SQL-like language; hence, nested queries are not supported. There is no support for join queries either. Instead, a view is defined on top of the fact table and dimension tables that hides the schema and the necessity for joins; users can write queries on top of the views.

(2) *Storage management*. Cheetah adopts a hybrid storage method that uses both row-based and column-based storage. The data are first horizontally partitioned into different chunks such that each chunk contains a set of rows. Each chunk is then stored in a column-oriented format.

(3) *Data compression*. Each column in a chunk can be compressed by one of the following methods: dictionary encoding [Abadi et al. 2006], run length encoding [Graefe and Shapiro 1991], and default value decoding [Raman and Swart 2006]. The compression type is determined by the type and statistics of the column. After each column is compressed, the entire chunk is further compressed.

(4) *Data partitioning*. The fact table and the big dimension tables are copartitioned so that the join query between the fact table and the dimension tables can be processed by a map-only job (similar to Trojan join). The remaining dimension tables are relatively small and can be directly loaded into memory at each compute node.

(5) *Query optimization*. Two unique aspects of Cheetah query optimization are the exploitation of materialized views and multiquery optimization. The former is straightforward, so we only discuss the latter. If a number of queries access the same input table, they use the same mappers to scan the table and use a query ID to indicate where the tuple is used. Then the tuple is partitioned according to the query ID and processed by different reducers in different queries.

A typical query consists of three parts: the source table, the filters, and the aggregation attributes [Nykiel et al. 2010]. Thus, three levels of sharing is possible: (1) *sharing scans only*—the queries have the same source table while the filters and the aggregation attributes are different; (2) *sharing map output*—the queries have the same source table and the same aggregation attributes (including both the grouping keys and the aggregated columns), while the filters are different; and (3) *sharing* `map` *functions*—the main purpose of the `map` function here is to filter the tuples based on the predicates. If the queries have the same source table and the same filters, the `map` functions can be shared between these queries. However, sharing scan or the map output is not always better than the original method that shares nothing. The cost analysis discussed in Section 4.2 can be applied here to choose the better alternative. In addition, to share the scan operation, queries have to be processed in batches, which delays some queries and increases their response times. Instead of merely sharing scans to reduce I/O, response time of the queries

Table X. Comparison of MapReduce DBMS Implementations

	HadoopDB	Llama	Cheetah
Language	SQL-like	Simple interface	SQL
Storage	Row store	Column store	Hybrid store
Data compression	No	Yes	Yes
Data partition	Horizontally partitioned	Vertically partitioned	Horizontally partitioned at chunk level
Indexing	Local index in each database instance	Local index + Bitmap index	Local index for each data chunk
Query optimization	Rule-based optimization plus local optimization by PostgreSQL	Column-based optimization, late materialization, and processing multiway join in one job	Multiquery optimization, materialized views

can also be considered: those requiring quick response are processed immediately after they are submitted to the system [Wang et al. 2011].

5.4. Comparisons of the Systems

The detailed comparison of the three systems is shown in Table X. In systems that support an SQL-like query language, user queries are transformed into a set of MapReduce jobs. These systems adopt different techniques to optimize query performance, and many of these techniques are adaptations of well-known methods incorporated into many relational DBMSs. The storage scheme of HadoopDB is row oriented, while Llama is a pure column-oriented system. Cheetah adopts a hybrid storage model where each chunk contains a set of rows that are vertically partitioned. This “first horizontally partition, then vertically partition” technique has been adopted by other systems such as RCFile [He et al. 2011]. Both Llama and Cheetah take advantage of superior data compression that is possible with column storage.

6. MAPREDUCE EXTENSIONS FOR DATA-INTENSIVE APPLICATIONS

As discussed earlier, MapReduce is capable of processing certain data-intensive workloads efficiently. These workloads can be implemented by a MapReduce job that consists of a `map` and a `reduce` function, followed by writing the data back to the distributed file system once the job is finished. However, this model is not well suited for a class of emerging data-intensive applications that are characterized by iterative computation requiring a chain of (i.e., multiple) MapReduce jobs (e.g., data analysis applications such as PageRank [Page et al. 1999]) or online aggregation. Extending MapReduce to cover this class of applications is currently an active research topic. Hence, we discuss them in this section.

6.1. Iterative Computation

Figure 13 shows an iterative task with three iterations that have two features: (1) the data source of each iteration consists of a variant part and an invariant part—the variant part consists of the files generated from the previous MapReduce jobs (the gray arrows in Figure 13), and the invariant part is the original input file (the black arrows in Figure 13); (2) a progress check might be needed at the end of each iteration to detect whether a fixed point has been reached. The fixed point has different meanings in different applications, such as whether the PageRank value has converged, or whether the within-cluster sum-of-squares is minimized in k-means clustering. As can be seen in Figure 13, an additional job is needed at the end of each iteration to compare the results generated between the current job and the previous one. The algorithm for the job controller of a simpler iterative computation with the fixed point is given

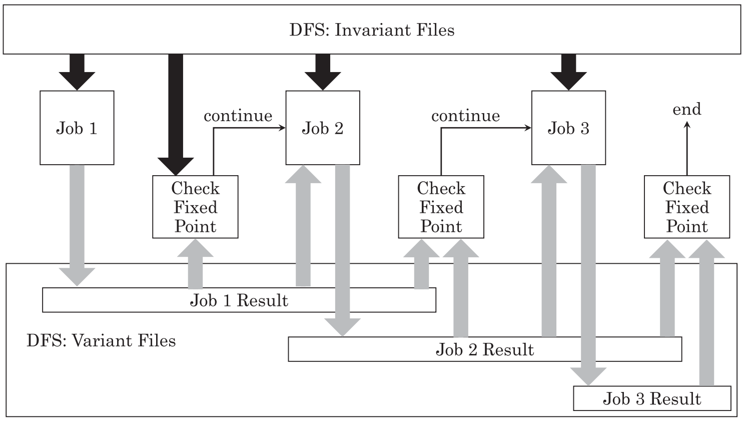


Fig. 13. MapReduce processing for iterative computation.

in Algorithm 3. Recall that the job controller sets the input path, input format, class containing `map` and `reduce` function, and so forth. In the algorithm, the function `addInputPath` sets the path of the input file for the MapReduce job, `setOutputPath` sets the path of the output file, and `waitForCompletion` waits for the job to complete.

ALGORITHM 3: Job Controller for Iterative Computation in MapReduce

```
input: String $InvariantPath$, String $IterativePath$
begin
    $i \leftarrow 0$;
    while fixed point not reached do
        $job1 \leftarrow$ new Job("Iterative Computation");
        set mapper class, reducer class, input format, output format, etc for $job1$;
        if $i \neq 0$ then $job1$.setIterativePath($IterativePath$ + (i - 1));
        $job1$.setInvariantPath($InvariantPath$);
        $job1$.setOutputPath($IterativePath$ + i);
        $job1$.execute();
        $job2 \leftarrow$ new Job("Checking Fixed Point");
        set mapper class, reducer class, input format, output format, etc for $job2$;
        $job2$.execute();
        $i \leftarrow i + 1$;
    end
end
```

HaLoop [Bu et al. 2010] is a MapReduce variant developed specifically for iterative computation and aggregation. In addition to the `map` and `reduce` functions, HaLoop introduces `AddMap` and `AddReduce` functions that express the iterative computation. To test the termination condition, the functions `SetFixedPointThreshold`, `ResultDistance`, and `SetMaxNumOfIterations` are defined. To distinguish the loop-variant and loop-invariant data, `AddStepInput` and `AddInvariantTable` are introduced. Using these functions, the job controller of HaLoop for the iterative application in Figure 13 is expressed in Algorithm 4.

To avoid unnecessary scans of invariant data, HaLoop maintains a reducer input cache storing the intermediate results of the invariant table. As a result, in each iteration, the mappers only need to scan the variant data generated from the previous jobs. The reducer pulls the variant data from the mappers and reads the invariant

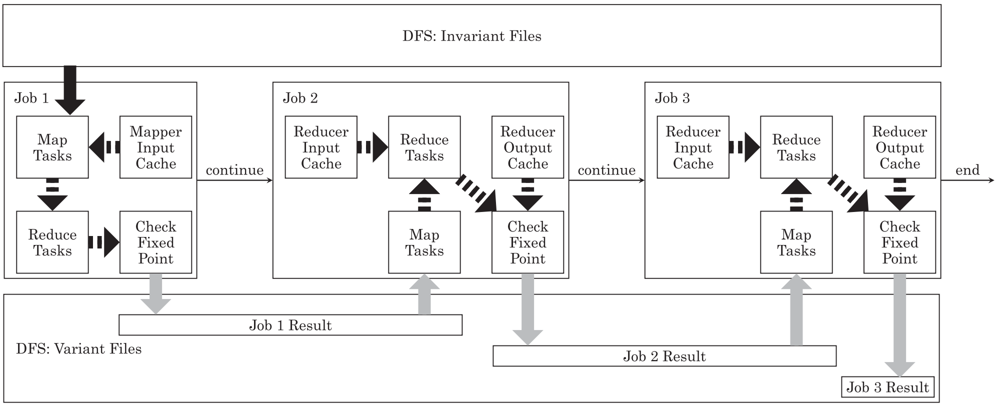


Fig. 14. Iterative computation in HaLoop.

ALGORITHM 4: Job Controller for Iterative Computation in HaLoop

```
input: String InvariantPath, String IterativePath, Func IterativeMapFunc, Func
       IterativeReduceFunc, Func ResultDistanceFunc
begin
    job ← new Job();
    job.AddMap(IterativeMapFunc, 1);
    /* set the map function for the first MapReduce job in each iteration. There
    might be several jobs in each iteration depending on the application */;
    job.AddReduce(IterativeReduceFunc, 1);
                    /* set the reduce function for the first job */;
    job.SetDistanceMeasure(ResultDistanceFunc);
    job.AddInvariantTable(InvariantPath);
    job.SetInput(IterativePath);
    job.SetReducerInputCache(true)        /* enable reducer input cache */;
    job.SetReducerOutputCache(true)       /* enable reducer output cache */;
    job.Submit();
end
```

data locally. To reduce the time to reach a fixed point, the reducer output cache is used to store the output locally in the reducers. At the end of each iteration, the output that has been cached from the previous iteration is compared with the newly generated results to detect whether the fixed point is reached. As can be seen, both of these caches need the mappers to shuffle the intermediate results with the same key to the same reducer running on the same machine in different iterations. To assign the reducers to the same machine in different iterations, the MapReduce scheduler is also modified. However, when a node fails, the reducer input cache contents contained in this node are lost. To solve this problem, the reducer input cache in the first iteration is stored in DFS, so that the cached data can be reloaded to a new node from the DFS. HaLoop also implements a mapper input cache that stores the input split of the mapper in the local file system. The HaLoop scheduler can recognize these caches and assign a map task to the node that contains the input cache of the task. This strategy can enhance data locality in HaLoop.

Figure 14 shows the revised execution flow in HaLoop for the iterative computation of Figure 13. In MapReduce Job1, the map tasks can read the data from either DFS or the mapper input cache, depending on whether the mapper input cache exists. In

MapReduce Job2, since the reduce tasks read data directly from reducer input cache, the map tasks do not need to read the invariant files from DFS. The reducer output cache is utilized to check the fixed point.

The experimental results show that HaLoop's running time of iterative computations are better than MapReduce by a factor of 1.85 [Bu et al. 2010]. However, the performance impact of cache reloading during the failure of the hosting node has not been studied.

Spark [Zaharia et al. 2010] is another system for performing iterative computation over a cluster. It introduces a storage abstraction called *Resilient Distributed Dataset* (RDD), which is a collection of tuples across a set of machines. When an RDD is constructed from HDFS as the input of the `map` function, the processing of the Spark job is similar to that of MapReduce: the `map` function is used to process each tuple in each partition of the RDD, and the `reduce` function is used for aggregation or other computations. Similar to HaLoop, Spark can cache the intermediate RDD and reuse it in subsequent iterations. In addition, this intermediate RDD can be cached in the local memory of each node, making the processing even faster.

Twister [Ekanayake et al. 2010] is a similar system that provides support for iterative processing. Users can write their `map` and `reduce` functions as in Hadoop, while the iterative MapReduce jobs are automatically linked and processed transparent to the users. It optimizes the iterative jobs by caching the resources (e.g., process stacks and status) for the mappers and reducers. Since creating new mappers and reducers incurs high overhead, these processes are reused throughout the computation. This strategy is also adopted by many Message Passing Interface (MPI) applications. The reused mappers and reducers can be configured to avoid loading static data (fixed input data for each iteration) repeatedly.

6.2. Streams and Continuous Query Processing

Another extension to MapReduce has been to address continuous processing such as stream processing [Stephens 1997; Golab and Özsu 2010] or online aggregation [Hellerstein et al. 1997; Wu et al. 2010]. Recall that a sort-merge process is accomplished by the mapper and reducer modules. Given the key/value pairs, the mapper generates new key/value pairs using the `map` function, groups them according to the partition key, and sorts the pairs in each partition. Then, the reducers pull the data from the mappers, and the sorted runs from different mappers are merged to one sorted file. In this step, the reducers are blocked by the mappers: they have to start the merge process after all the mappers finish their tasks. This blocking behavior between the mapper and the reducer incurs some overhead and needs to be eliminated to support continuous query processing. MapReduce Online [Condie et al. 2010] addresses this by pipelining intermediate data between operators. MapReduce Online supports both the pipeline within one MapReduce job and the pipeline between consecutive jobs.

(1) **Pipeline within a job.** To support pipelining between the mappers and reducers, MapReduce Online replaces pulling by the reducers with pushing by the mappers. That is, each mapper directly sends the data to the reducers once it processes a key/value pair, without sorting the data locally. As a result, the reducers have to sort the data when they receive them. In this case, the combiner module cannot be used since the data are shuffled to the reducers immediately. Furthermore, due to the limited number of the TCP connections per node, in a large cluster, the mappers may not connect to all the reducers directly. Thus, in MapReduce Online, a refined pipeline method is adopted. If the mapper cannot push the records to the specified reducer immediately due to the TCP limitation, it stores these records

locally (called *spill files*). Then, the combiner module can be utilized to perform a local aggregation to the spills, reducing the size of the intermediate data shuffled to reducers.

(2) **Pipeline between jobs.** If an application needs several MapReduce jobs, the results of the intermediate jobs are typically written into the DFS. Instead, MapReduce Online supports data transfer from the reducers directly to the mappers of the next job. As a result, pipelining is extended across multiple jobs, and more complex continuous queries that incur several MapReduce jobs can be implemented.

One advantage of the pull model is fault tolerance: since a mapper shuffles its data to reducers only if it completes the map task, the reducers will not receive any data from this failed mapper. In contrast, when push is used, as in MapReduce Online, reducers may receive *some* data from the failed mapper. To recover from the failure of a mapper, the mapper state indicating whether it is completed needs to be maintained. If the mapper has not completed its task, the reducers do not combine the spill files shuffled from this mapper with others; they only store these spill files temporarily. If the mapper finishes, the reducer merges the data with the others. Thus, when an incomplete map task fails, the reducer simply ignores the data produced by this mapper, and a new mapper is launched to execute this map task. To address the failure of the reducer, the mappers have to store all the spill files locally until the entire job is finished, and these local files are shuffled to the new launched reducer in the event of a reducer failure.

By supporting online aggregation through pipelining, the reducer can get the data from the mapper continuously, and consequently, the user only requires to specify when a snapshot of the aggregation needs to be computed in the reducers. For example, the snapshot could be computed and returned to the user when a certain percentage of the input has been received by the reducer. Alternatively, rather than modifying the pull model of MapReduce, each data chunk assigned to the mapper can be treated as one unit in online aggregation [Pansare et al. 2011]. The snapshot is taken on the data shuffled from the mappers that have finished processing the assigned data chunks.

An alternative method to support continuous query processing is to implement a purely hash-based framework to replace the sort-merge process in MapReduce [Li et al. 2011]. The idea is to "group data by key, then apply the `reduce` function to each group" while avoiding the overhead of sorting on the map side and blocking on the reducer side. First, a hash function h_1 is used to partition the map output into different groups for different reducers. The key/value pairs in each group do not need to be sorted, which eliminates the sort time during the map process. The reducer builds an in-memory hash table H using hash function h_2. For each key/value pair that comes from the mapper, if the hash key exists in H, the state of the hash key is updated by a function of the old state and the new value. Otherwise, the key/value pair is inserted into H if it is not already full. If it is full, another hash function h_3 is used to hash the key/value pairs into some bucket in the local file system. Once the state of the key in H satisfies some application-defined threshold, the computation for this key can be stopped. For example, if a user who visits a website 10 times a day is considered a frequent user, the computation for counting the visits of that user can be stopped when this threshold (10 times) is satisfied. To generate the results for frequent keys faster, the h_2 function can be modified to hash the frequent keys into H.

In addition to these systems that extend MapReduce for continuous and iterative processing, there are other distributed stream processing systems that are inspired by MapReduce but that go beyond the MapReduce framework. We shall briefly review them in the next section.

7. MAPREDUCE-INFLUENCED DATA PROCESSING PLATFORMS

There are many other distributed data processing systems that have been inspired by MapReduce but that go beyond the MapReduce framework. These systems have been designed to address various problems, such as iterative processing over the same dataset, that are not well handled by MapReduce, and many are still ongoing. In this section, for the purpose of highlighting deviations from the MapReduce framework, we shall briefly discuss some representative systems. In particular, we shall discuss general-purpose distributed processing platforms that have a more flexible programming model (e.g., Dryad [Isard et al. 2007] and epiC [Chen et al. 2010]), large-scale graph processing systems (e.g., Pregel [Malewicz et al. 2010] and Distributed GraphLab [Low et al. 2012]), and continuous data processing systems (e.g., S4 [Neumeyer et al. 2010] and Storm[7]).

7.1. Generic Data Processing Platforms

An interesting line of research has been to develop parallel processing platforms that have MapReduce flavor, but are more general. Two examples of this line of work are Dryad [Isard et al. 2007] and epiC [Chen et al. 2010].

Dryad [Isard et al. 2007] represents each job as a directed acyclic graph whose vertices correspond to processes and whose edges represent communication channels. Dryad jobs (graphs) consist of several stages such that vertices in the same stage execute the same user-written functions for processing their input data. Consequently, the MapReduce programming model can be viewed as a special case of Dryad's where the graph consists of two stages: the vertices of the map stage shuffle their data to the vertices of the reduce stage.

Driven by the limitations of MapReduce-based systems in dealing with "varieties" in cloud data management, epiC [Chen et al. 2010] was designed to handle a variety of data (e.g., structured and unstructured), a variety of storage (e.g., database and file systems), and a variety of processing (e.g., SQL and proprietary APIs). Its execution engine is similar to Dryad's to some extent. The important characteristic of epiC, from a MapReduce or data management perspective, is that it simultaneously supports both data-intensive analytical workloads (OLAP) and online transactional workloads (OLTP). Traditionally, these two modes of processing are supported by different engines. The system consists of the Query Interface, OLAP/OLTP controller, Elastic Execution Engine (E3), and Elastic Storage System (ES2) [Cao et al. 2011]. SQL-like OLAP queries and OLTP queries are submitted to the OLAP/OLTP controller through the Query Interface. E3 is responsible for the large-scale analytical jobs, and ES2, the underlying distributed storage system that adopts the relational data model and supports various indexing mechanisms [Chen et al. 2011; Wang et al. 2010; Wu et al. 2010], handles the OLTP queries.

7.2. Graph Processing Platforms

A growing class of applications use graph data (e.g., social network analysis, RDF), and the analysis of graphs has been a topic of considerable interest. Graph processing is of an iterative nature, and the same dataset may have to be revisited many times; this calls for a design that deviates from the phase-based stateless processing framework.

Pregel [Malewicz et al. 2010] operates on a large directed graph. A Pregel job is executed in several *supersteps* (iterations), and it is terminated when a certain condition for the graph is satisfied. In each superstep, a user-written `compute` function is applied to the vertices. The `compute` function operates on each vertex and may update the

[7] http://storm-project.net/.

vertex state based on its previous state and the message passed to it from the preceding vertices. As discussed in Section 6.1, an iterative query like PageRank computation can also be implemented by a chain of MapReduce jobs, in which case the state and the graph information must be reloaded every time a MapReduce job is started. In contrast, in Pregel, the edges and the state of each vertex are stored across supersteps during the lifetime of the Pregel job. Fault tolerance is achieved through periodic checkpointing. Once a worker failure occurs, the state of the vertex values and edges of the entire graph are recovered from the latest checkpoint (which may be a number of supersteps before the failure). Checkpointing and synchronization are costly, which may be reduced by trading between checkpointing and recovery.

The architecture of Distributed GraphLab [Low et al. 2012] is similar to Pregel, while the processing model is different: a user-defined `update` function modifies the the state of a vertex based on its previous state, the current state of all of its adjacent vertices, and the value of its adjacent edges. An important difference between GraphLab and Pregel is in terms of their synchronization models, which defines how the vertices collaborate during processing. Pregel only supports the Bulk Synchronization Model (BSM) [Valiant 1990], where, after the vertices complete their processing of a superstep, all vertices should reach a global synchronization status; GraphLab allows three choices: synchronized, asynchronized, and partially synchronized.

7.3. Continuous Data Processing Platforms

S4 [Neumeyer et al. 2010] is a distributed stream processing system that follows the Actor programming model. Each keyed tuple in the data stream is treated as an event and is the unit of communication between processing elements (PEs). PEs form a directed acyclic graph, which can also be grouped into several stages. At each stage, all the PEs share the same computation function, and each PE processes the events with certain keys. The architecture of S4 is different from the MapReduce-based systems: it adopts a decentralized and symmetric architecture. In S4, there is no master node that schedules the entire cluster. The cluster has many Processing Nodes (PNs) that contain several PEs for processing the events. Since the data are streaming between PEs, there's no on-disk checkpoint for the PEs. Thus, the partial fault tolerance is achieved in S4: if a PN failure occurs, its processes are moved to a standby server, but the state of these processes is lost and cannot be recovered.

Storm is another stream processing system in this category that shares many features with S4. A Storm job is also represented by a directed acyclic graph, and its fault tolerance is partially due to the streaming channel between vertices. The difference is the architecture: Storm is a master-slave system like MapReduce. A Storm cluster has a master node (called Nimbus) and worker nodes (called supervisors).

7.4. Discussion

Table XI shows a quick summary on some common features of the systems discussed in this section. Most of the systems use the Directed Acyclic Graph (DAG) as their programming model. Compared to MapReduce's simplified programming model, the DAG is more flexible and can express more complex queries. The features of Dryad and epiC are mostly the same except that epiC also supports OLTP queries. The architecture of these systems is usually master-slaves (influenced by MapReduce), but S4 adopts the decentralized architecture. The fault tolerance strategy is quite different for different systems. For Storm and S4, when node failure occurs, the processes on the failed node are moved to standby nodes. However, as the data streamed between the vertices are not cached, only the functionality of the failed node is recovered, while the states are lost. Pregel and GraphLab use checkpointing for fault tolerance, which is invoked

Table XI. Summary of Non-MapReduce-Based Data Processing Platforms

Name	Application	Programming Model	Architecture	Fault Tolerance
Dryad	General-purpose parallel execution engine	Directed acyclic graph	Master-Slaves	Node-level fault tolerance
epiC	Large-scale OLAP + OLTP	Directed acyclic graph	Master-Slaves	Node-level fault tolerance
Pregel	Large-scale graph processing engine	Directed graph	Master-Slaves	Checkpointing
GraphLab	Large-scale machine learning and data mining framework	Directed graph	Master-Slaves	Checkpointing
Storm	Distributed streaming processing engine	Directed acyclic graph	Master-Slaves	Partial fault tolerance
S4	Distributed streaming processing engine	Directed acyclic graph	Decentralized and symmetric	Partial fault tolerance

at the beginning of some iterations. The iterations following a checkpoint need to be repeated (when some vertices fail).

While MapReduce could not support all applications efficiently, it has generated huge interest in designing scalable, elastic, and efficient distributed large-scale processing platforms. Issues such as the right tradeoff between consistency and availability, efficient processing with quick recovery, and extensibility remain to be solved.

8. CONCLUSION

As a massively parallel processing framework, MapReduce is well recognized for its scalability, flexibility, fault tolerance, and a number of other attractive features. In particular, it facilitates parallelization of a class of applications, commonly referred to as embarrassingly parallelizable. However, as has been commonly acknowledged, MapReduce has not been designed for large-scale complex data management tasks. For example, the original framework does not provide high-level language support that is familiar to and expected by database users; consequently, users have to individually develop various processing logics and programs. It also does not have built-in indexing and query optimization support required for database queries. This has naturally led to a long stream of research that attempts to address the lack of database functionality.

In this survey, our focus is on the enhancement and extension of the MapReduce framework for database applications. We reviewed the basic MapReduce framework and its various implementations. Since MapReduce is a framework with no concrete specification of how each component should be implemented, there have been different approaches to the implementation. We compared the design and feature set of the various well-known implementations.

There have also been a number of proposals for extending the basic framework with new functionality (e.g., languages) and mechanisms of optimizing its performance for this class of jobs. Many state-of-the-art database techniques have been adapted to enhance MapReduce systems to improve their performance and usability. We reviewed the relevant work on these topics, focusing on the implementation alternatives for basic database operators in MapReduce and efforts in extending MapReduce as the database engine in a distributed environment.

In this article we also discussed systems that extend MapReduce for more data-intensive applications, and that go beyond MapReduce while retaining its flavor and design principles. Most of this research is ongoing, and there will likely be significant improvements to the MapReduce framework as well as extensions to the MapReduce paradigm for distributed and parallel computing. Distributed data management proposals over MapReduce are typically based on the more mature technology of distributed and parallel database systems. It has been widely acknowledged that traditional parallel DBMSs have been designed with a full set of functionality but with limited scalability, while the MapReduce systems have been designed for massive scalability but trade off a considerable amount of functionality. Both systems have their own strengths and weaknesses and cater to somewhat different users and applications. Notwithstanding these differences, parallel database systems and MapReduce-based systems do share a very important common ground of exploiting data partitioning and partitioned execution for achieving parallelism, and surely, we will see more fusion of technologies where applicable. This article provides a comprehensive analysis of the technology and the work to date and serves as a basis for further experimentation and research.

ACKNOWLEDGMENTS

This research is funded by the Singapore National Research Foundation and the publication is supported under the Competitive Research Programme (CRP) Grant NRF-CRP8-2011-08, and by the Natural Sciences and Engineering Research Council of Canada under the Discovery Grants Program.

REFERENCES

Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 671–682.

Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. 2009. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 922–933.

Azza Abouzied, Kamil Bajda-Pawlikowski, Jiewen Huang, Daniel J. Abadi, and Avi Silberschatz. 2010. HadoopDB in action: building real world applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, 1111–1114. DOI:http://dx.doi.org/10.1145/1807167.1807294

Foto N. Afrati, Anish Das Sarma, David Menestrina, Aditya G. Parameswaran, and Jeffrey D. Ullman. 2012. Fuzzy joins using MapReduce. In *Proceedings of the 28th International Conference on Data Engineering*. 498–509.

Foto N. Afrati and Jeffrey D. Ullman. 2010. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*. 99–110.

Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the outliers in map-reduce clusters using Mantri. In *Proceedings of the 9th USENIX Symposium on Operating System Design and Implementation*. 1–16.

Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up all pairs similarity search. In *Proceedings of the 16th International World Wide Web Conference*. 131–140.

Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using semi-joins to solve relational queries. *J. ACM* 28, 1 (Jan. 1981), 25–40.

Kevin S. Beyer, Vuk Ercegovac, Rajasekar Krishnamurthy, Sriram Raghavan, Jun Rao, Frederick Reiss, Eugene J. Shekita, David E. Simmen, Sandeep Tata, Shivakumar Vaithyanathan, and Huaiyu Zhu. 2009. Towards a scalable enterprise content analytics platform. *Q. Bull. IEEE TC on Data Eng.* 32, 1 (2009), 28–35.

Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. 2010. A comparison of join algorithms for log processing in MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 975–986.

Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. 2010. HaLoop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 285–296.

Yu Cao, Chun Chen, Fei Guo, Dawei Jiang, Yuting Lin, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. 2011. ES2: A cloud data storage system for supporting both OLTP and OLAP. In *Proceedings of the 27th International Conference on Data Engineering*. 291–302. DOI:http://dx.doi.org/10.1109/ICDE.2011.5767881

Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*. 363–375.

Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, and Michael Wong. 2011. Tenzing: A SQL implementation on the MapReduce framework. *Proc. VLDB Endow.* 4, 12 (2011), 1318–1327.

Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering*. 5.

Surajit Chaudhuri and Gerhard Weikum. 2000. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *Proceedings of the 26th International Conference on Very Large Data Bases*. 1–10.

Chun Chen, Gang Chen, Dawei Jiang, Beng Chin Ooi, Hoang Tam Vo, Sai Wu, and Quanqing Xu. 2010. Providing scalable database aervices on the cloud. In *Proceedings of the 11th International Conference on Web Information Systems Engineering*. 1–19.

Gang Chen, Hoang Tam Vo, Sai Wu, Beng Chin Ooi, and M. Tamer Özsu. 2011. A Framework for Supporting DBMS-like Indexes in the Cloud. *PVLDB* 4, 11 (2011), 702–713.

Songting Chen. 2010. Cheetah: A high performance, custom data warehouse on top of MapReduce. *Proc. VLDB Endow.* 3, 2 (2010), 1459–1468.

Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*. 21–21.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation*. 137–150.

D. Dewitt and M Stonebraker. 2009. MapReduce: A Major Step Backwards. http://www.cs.washington.edu/homes/billhowe/mapreduce_a_major_step_backw ards.html. (2009).

David J. DeWitt, Erik Paulson, Eric Robinson, Jeffrey Naughton, Joshua Royalty, Srinath Shankar, and Andrew Krioukov. 2008. Clustera: An integrated computation and data management system. *Proc. VLDB Endow.* 1, 1 (2008), 28–41.

Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. 2010. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *Proc. VLDB Endow.* 3, 1 (2010), 518–529.

Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. 2010. Twister: A runtime for iterative MapReduce. In *Proceedings of the 19th IEEE International Symposium High Performance Distributed Computing*. 810–818.

Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. 2011. Column-oriented storage techniques for MapReduce. *Proc. VLDB Endow.* 4, 7 (April 2011), 419–429.

Eric Friedman, Peter Pawlowski, and John Cieslewicz. 2009. SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.* 2 (August 2009), 1402–1413. Issue 2.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating System Principles*. 29–43.

Amol Ghoting, Rajasekar Krishnamurthy, Edwin P. D. Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. 2011. SystemML: Declarative machine learning on MapReduce. In *Proceedings of the 27th International Conference on Data Engineering*. 231–242.

Lukasz Golab and M. Tamer Özsu. 2010. *Data Stream Systems*. Morgan & Claypool.

Goetz Graefe and Leonard D. Shapiro. 1991. Data compression and database performance. In *Proceedings of the 1991 ACM Symposium on Applied Computing*. 22–27.

Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. 2012. Load balancing in MapReduce based on scalable cardinality estimates. In *Proceedings of the 28th International Conference on Data Engineering*. 522–533.

Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. 2011. RCfile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *Proceedings of the 27th International Conference on Data Engineering*. 1199–1208.

Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 171–182.

Herodotos Herodotou and Shivnath Babu. 2011. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *Proc. VLDB Endow.* 4, 11 (2011), 1111–1122.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*. 59–72.

Eaman Jahani, Michael J. Cafarella, and Christopher Ré. 2011. Automatic optimization for MapReduce programs. *Proc. VLDB Endow.* 4, 6 (2011), 385–396.

Jeffrey Jestes, Ke Yi, and Feifei Li. 2011. Building wavelet histograms on large data in MapReduce. *Proc. VLDB Endow.* 5, 2 (Oct. 2011), 109–120.

Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. 2010. The performance of MapReduce: An in-depth study. *Proc. VLDB Endow.* 3, 1 (2010), 472–483.

David Jiang, Anthony K. H. Tung, and Gang Chen. 2011. MAP-JOIN-REDUCE: Toward scalable and efficient data analysis on large clusters. *IEEE Trans. Knowl. Data Eng.* 23, 9 (2011), 1299–1311.

Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. 2011. Trojan data layouts: Right shoes for a running elephant. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. Article 21, 14 pages.

Younghoon Kim and Kyuseok Shim. 2012. Parallel top-k similarity join algorithms using MapReduce. In *Proceedings of the 28th International Conference on Data Engineering*. 510–521.

Paraschos Koutris and Dan Suciu. 2011. Parallel evaluation of conjunctive queries. In *Proceedings of the 30th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, New York, NY, USA, 223–234.

Vibhore Kumar, Henrique Andrade, Bugra Gedik, and Kun-Lung Wu. 2010. DEDUCE: At the intersection of MapReduce and stream processing. In *Proceedings of the 13th International Conference on Extending Database Technology*. 657–662.

YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. SkewTune: mitigating skew in MapReduce applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 25–36.

Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. 2011. A platform for scalable one-pass analytics using MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 985–996.

Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. 2011. Llama: leveraging columnar storage for scalable join processing in the MapReduce framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 961–972.

Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning in the cloud. *CoRR* abs/1204.6078 (2012).

Peng Lu, Sai Wu, Lidan Shou, and Kian-Lee Tan. 2013. An Efficient and Compact Indexing Scheme for Large-scale Data Store. In *Proceedings of the 29th International Conference on Data Engineering*. 326–337.

Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. 2012. Efficient processing of k nearest neighbor joins using MapReduce. *Proc. VLDB Endow.* 5, 10 (June 2012), 1016–1027.

Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 135–146.

Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2011. Dremel: Interactive analysis of web-scale datasets. *Commun. ACM* 54, 6 (June 2011), 114–123.

Ahmed Metwally and Christos Faloutsos. 2012. V-SMART-join: A scalable MapReduce framework for all-pair similarity joins of multisets and vectors. *Proc. VLDB Endow.* 5, 8 (April 2012), 704–715.

Kristi Morton, Magdalena Balazinska, and Dan Grossman. 2010a. ParaTimer: A progress indicator for MapReduce DAGs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 507–518.

Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. 2010b. Estimating the progress of MapReduce pipelines. In *Proceedings of the 26th International Conference on Data Engineering*. 681–684.

Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. 2010. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*. 170–177.

Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: Sharing across multiple queries in MapReduce. *Proc. VLDB Endow.* 3, 1 (2010), 494–505.

Alper Okcan and Mirek Riedewald. 2011. Processing theta-joins using MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 949–960.

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1099–1110.

Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 29–41.

M. Tamer Özsu and Patrick Valduriez. 2011. *Principles of Distributed Database Systems* (3 ed.). Springer.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999–66. Stanford InfoLab.

Niketan Pansare, Vinayak R. Borkar, Chris Jermaine, and Tyson Condie. 2011. Online aggregation for large MapReduce jobs. *Proc. VLDB Endow.* 4, 11 (2011), 1135–1145.

Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 165–178.

Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. 2005. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.* 13, 4 (Oct. 2005), 277–298.

Smriti R. Ramakrishnan, Garret Swart, and Aleksey Urmanov. 2012. Balancing reducer skew in MapReduce workloads using progressive sampling. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*. 16:1–16:14.

Vijayshankar Raman and Garret Swart. 2006. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. 858–869.

Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. 2012. Sailfish: a framework for large scale data processing. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*. 4:1–4:14.

Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. 2012. Themis: an I/O-efficient MapReduce. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*. 13:1–13:14.

Donovan A. Schneider and David J. Dewitt. 1989. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 110–121.

R. Stephens. 1997. A survey of stream processing. *Acta Informatica* 34, 7 (1997), 491–541.

Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. 2010. MapReduce and parallel DBMSs: Friends or foes? *Commun. ACM* 53, 1 (2010), 64–71.

Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-store: A column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 553–564.

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*. 1150–1160.

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive—A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.* 2, 2 (2009), 1626–1629.

Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. DOI:http://dx.doi.org/10.1145/79173.79181

Rares Vernica, Michael J. Carey, and Chen Li. 2010. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 495–506.

Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. 2012. LogBase: a scalable log-structured database system in the cloud. *Proc. VLDB Endow.* 5, 10 (2012), 1004–1015.

Guanying Wang, Ali Raza Butt, Prashant Pandey, and Karan Gupta. 2009. A simulation approach to evaluating design decisions in MapReduce setups. In *Proceedings of the 17th IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Syst.* 1–11.

Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. 2010. Indexing multi-dimensional data in a cloud system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 591–602.

Xiaodan Wang, Christopher Olston, Anish Das Sarma, and Randal Burns. 2011. CoScan: cooperative scan sharing in the cloud. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. 11:1–11:12.

Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. 2010. Efficient B-tree based indexing for cloud data processing. *Proc. VLDB Endow.* 3, 1 (2010), 1207–1218.

Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. 2011. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. Article 12, 13 pages.

Sai Wu, Beng Chin Ooi, and Kian-Lee Tan. 2010. Continuous sampling for online aggregation over multiple queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, 651–662. DOI: http://dx.doi.org/10.1145/1807167.1807238

Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Efficient similarity joins for near duplicate detection. In *Proceedings of the 17th International World Wide Web Conference*. 131–140.

Hung-Chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and Douglas Stott Parker Jr. 2007. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1029–1040.

Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2009. *Job Scheduling for Multi-User MapReduce Clusters*. Technical Report. Berkeley.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*.

Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation*. 29–42.

Chi Zhang, Feifei Li, and Jeffrey Jestes. 2012b. Efficient parallel kNN joins for large data in MapReduce. In *Proceedings of the 12th International Conference on Extending Database Technology*. 38–49.

Xiaofei Zhang, Lei Chen, and Min Wang. 2012a. Efficient multi-way theta-join processing using MapReduce. *Proc. VLDB Endow.* 5, 11 (2012), 1184–1195.

Received 15 September 2012; revised 21 February 2013; accepted 26 June 2013