The Family of MapReduce and Large-Scale Data Processing Systems

SHERIF SAKR and ANNA LIU, NICTA and University of New South Wales
AYMAN G. FAYOUMI, King Abdulaziz University

In the last two decades, the continuous increase of computational power has produced an overwhelming flow of data which has called for a paradigm shift in the computing architecture and large-scale data processing mechanisms. MapReduce is a simple and powerful programming model that enables easy development of scalable parallel applications to process vast amounts of data on large clusters of commodity machines. It isolates the application from the details of running a distributed program such as issues on data distribution, scheduling, and fault tolerance. However, the original implementation of the MapReduce framework had some limitations that have been tackled by many research efforts in several followup works after its introduction. This article provides a comprehensive survey for a *family* of approaches and mechanisms of large-scale data processing mechanisms that have been implemented based on the original idea of the MapReduce framework and are currently gaining a lot of momentum in both research and industrial communities. We also cover a set of introduced systems that have been implemented to provide declarative programming interfaces on top of the MapReduce framework. In addition, we review several large-scale data processing systems that resemble some of the ideas of the MapReduce framework for different purposes and application scenarios. Finally, we discuss some of the future research directions for implementing the next generation of MapReduce-like solutions.

Categories and Subject Descriptors: H.2.2 [**Database Management**]: Physical Design—*Access methods*; H.2.4 [**Systems**]: Distributed Databases—*Query processing*; H.2.5 [**Heterogeneous Databases**]: Data Translation

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: MapReduce, big data, large-scale data processing

ACM Reference Format:
Sakr, S., Liu, A., and Fayoumi, A. G. 2013. The family of mapreduce and large-scale data processing systems. ACM Comput. Surv. 46, 1, Article 11 (October 2013), 44 pages.
DOI: http://dx.doi.org/10.1145/2522968.2522979

1. INTRODUCTION

We live in the era of *big data* where we are witnessing a continuous increase on the computational power that produces an overwhelming flow of data which has called for a *paradigm shift* in the computing architecture and large-scale data processing mechanisms. Powerful telescopes in astronomy, particle accelerators in physics, and genome sequencers in biology are putting massive volumes of data into the hands of scientists. For example, the Large Synoptic Survey Telescope [LSS 2013] generates on the order

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.
Authors' addresses: S. Sakr (corresponding author) and A. Liu, Department of Computer Science and Engineering, NICTA and the University of New South Wales, Sydney, Australia; email: ssakr@cse.unsw.edu.au; A. G. Fayoumi, King Abdulaziz University, Jeddah, Saudia Arabia.

DOI: http://dx.doi.org/10.1145/2522968.2522979

of 30 TeraBytes of data every day. Many enterprises continuously collect large datasets that record customer interactions, product sales, results from advertising campaigns on the Web, and other types of information. For example, Facebook collects 15 TeraBytes of data each day into a PetaByte-scale data warehouse [Thusoo et al. 2009]. Jim Gray called the shift a "fourth paradigm" [Hey et al. 2009]. The first three paradigms were *experimental*, *theoretical* and, more recently, *computational science*. Gray argued that the only way to cope with this paradigm is to develop a new generation of computing tools to manage, visualize, and analyze the data flood. In general, current computer architectures are increasingly imbalanced where the latency gap between multicore CPUs and mechanical hard disks is growing every year, which makes the challenges of data-intensive computing much harder to overcome [Bell et al. 2006]. Hence, there is a crucial need for a systematic and generic approach to tackle these problems with an architecture that can also scale into the foreseeable future. In response, Gray argued that the new trend should instead focus on supporting less expensive clusters of computers to manage and process all this data instead of focusing on having the biggest and fastest single computer.

In general, the growing demand for large-scale data mining and data analysis applications has spurred the development of novel solutions from both the industry (e.g., Web data analysis, click-stream analysis, network-monitoring log analysis) and the sciences (e.g., analysis of data produced by massive-scale simulations, sensor deployments, high-throughput lab equipment). Although parallel database systems [DeWitt and Gray 1992] serve some of these data analysis applications (e.g., Teradata[1], SQL Server PDW[2], Vertica[3], Greenplum[4], ParAccel[5], Netezza[6]), they are expensive, difficult to administer, and lack fault tolerance for long-running queries [Pavlo et al. 2009]. MapReduce [Dean and Ghemawat 2004] is a framework which is introduced by Google for programming commodity computer clusters to perform large-scale data processing in a single pass. The framework is designed such that a MapReduce cluster can scale to thousands of nodes in a fault-tolerant manner. One of the main advantages of this framework is its reliance on a simple and powerful programming model. In addition, it isolates the application developer from all the complex details of running a distributed program such as: issues on data distribution, scheduling, and fault tolerance [Patterson 2008].

Recently, there has been a great deal of hype about cloud computing [Armbrust et al. 2009]. In principle, cloud computing is associated with a new paradigm for the provision of computing infrastructure. This paradigm shifts the location of this infrastructure to more centralized and larger-scale datacenters in order to reduce the costs associated with the management of hardware and software resources. In particular, cloud computing has promised a number of advantages for hosting the deployments of data-intensive applications such as:

—reduced time-to-market by removing or simplifying the time-consuming hardware provisioning, purchasing, and deployment processes;
—reduced monetary cost by following a *pay-as-you-go* business model;
—unlimited (virtually) throughput by adding servers if the workload increases.

[1]http://teradata.com/.
[2]http://www.microsoft.com/sqlserver/en/us/solutions-technologies/data-warehousing/pdw.aspx.
[3]http://www.vertica.com/.
[4]http://www.greenplum.com/.
[5]http://www.paraccel.com/.
[6]http://www-01.ibm.com/software/data/netezza/.

In principle, the success of many enterprises often relies on their ability to analyze expansive volumes of data. In general, cost-effective processing of large datasets is a nontrivial undertaking. Fortunately, MapReduce frameworks and cloud computing have made it easier than ever for everyone to step into the world of big data. This technology combination has enabled even small companies to collect and analyze terabytes of data in order to gain a competitive edge. For example, the Amazon Elastic Compute Cloud (EC2)[7] is offered as a commodity that can be purchased and utilised. In addition, Amazon has also provided the Amazon Elastic MapReduce[8] as an online service to easily and cost effectively process vast amounts of data without the need to worry about time-consuming setup, management, or tuning of computing clusters or the compute capacity upon which they sit. Hence, such services enable third parties to perform their analytical queries on massive datasets with minimum effort and cost by abstracting the complexity entailed in building and maintaining computer clusters.

The implementation of the basic MapReduce architecture had some limitations. Therefore, several research efforts have been triggered to tackle these limitations by introducing several advancements in the basic architecture in order to improve its performance. This article provides a comprehensive survey for a family of approaches and mechanisms of large-scale data analysis mechanisms that have been implemented based on the original idea of the MapReduce framework and are currently gaining a lot of momentum in both research and industrial communities. In particular, the remainder of this article is organized as follows. Section 2 describes the basic architecture of the MapReduce framework. Section 3 discusses several techniques that have been proposed to improve the performance and capabilities of the MapReduce framework from different perspectives. Section 4 covers several systems that support a high-level SQL-like interface for the MapReduce framework. Section 5 reviews several large-scale data processing systems that resemble some of the ideas of the MapReduce framework, without sharing its architecture or infrastructure, for different purposes and application scenarios. In Section 6, we conclude the article and discuss some of the future research directions for implementing the next generation of MapReduce/Hadoop-like solutions.

2. MAPREDUCE FRAMEWORK: BASIC ARCHITECTURE

The MapReduce framework is introduced as a simple and powerful programming model that enables easy development of scalable parallel applications to process vast amounts of data on large clusters of commodity machines [Dean and Ghemawat 2004, 2008]. In particular, the implementation described in the original paper is mainly designed to achieve high performance on large clusters of commodity PCs. One of the main advantages of this approach is that it isolates the application from the details of running a distributed program, such as issues on data distribution, scheduling, and fault tolerance. In this model, the computation takes a set of key/value pairs input and produces a set of key/value pairs as output. The user of the MapReduce framework expresses the computation using two functions: *Map* and *Reduce*. The Map function takes an input pair and produces a set of intermediate key/value pairs. The MapReduce framework groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function. The Reduce function receives an intermediate key I with its set of values and merges them together. Typically just zero or one output value is produced per Reduce invocation. The main advantage of this model is that it allows large computations to be easily parallelized and reexecuted to be used as the primary mechanism for fault tolerance. Figure 1 illustrates an example MapReduce

[7]http://aws.amazon.com/ec2/.
[8]http://aws.amazon.com/elasticmapreduce/.

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
        EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
        result += ParseInt(v);
Emit(AsString(result));
```

Fig. 1. An example MapReduce program. Adaptation with permission of Dean and Ghemawat [2004].

program expressed in pseudocode for counting the number of occurrences of each word in a collection of documents. In this example, the map function emits each word plus an associated count of occurrences while the reduce function sums together all counts emitted for a particular word. The design of the MapReduce framework has considered the following main principles [Yang et al. 2007].

—*Low-Cost Unreliable Commodity Hardware*. Instead of using expensive, high-performance, reliable Symmetric MultiProcessing (SMP) or Massively Parallel Processing (MPP) machines equipped with high-end network and storage subsystems, the MapReduce framework is designed to run on large clusters of commodity hardware. This hardware is managed and powered by open-source operating systems and utilities so that the cost is low.

—*Extremely Scalable RAIN Cluster*. Instead of using centralized RAID-based SAN or NAS storage systems, every MapReduce node has its own local off-the-shelf hard drives. These nodes are loosely coupled where they are placed in racks that can be connected with standard networking hardware connections. These nodes can be taken out of service with almost no impact to still-running MapReduce jobs. These clusters are called Redundant Array of Independent (and inexpensive) Nodes (RAIN).

—*Fault Tolerant yet Easy to Administer*. MapReduce jobs can run on clusters with thousands of nodes or even more. These nodes are not very reliable as at any point in time, a certain percentage of these commodity nodes or hard drives will be out of order. Hence, the MapReduce framework applies straightforward mechanisms to replicate data and launch backup tasks so as to keep still-running processes going. To handle crashed nodes, system administrators simply take crashed hardware offline. New nodes can be plugged in at any time without much administrative hassle. There is no complicated backup, restore, and recovery configurations like the ones that can be seen in many DBMSs.

—*Highly Parallel yet Abstracted*. The most important contribution of the MapReduce framework is its ability to automatically support the parallelization of task executions. Hence, it allows developers to focus mainly on the problem at hand rather than worrying about the low-level implementation details such as memory management, file allocation, parallel, multithreaded, or network programming. Moreover, MapReduce's shared-nothing architecture [Stonebraker 1986] makes it much more scalable and ready for parallelization.

Hadoop[9] is an open-source Java library [White 2012] that supports data-intensive distributed applications by realizing the implementation of the MapReduce framework[10]. It has been widely used by a large number of business companies for production purposes[11]. On the implementation level, the Map invocations of a MapReduce job are

[9]http://hadoop.apache.org/.
[10]In the rest of this article, we use the two names: MapReduce and Hadoop, interchangeably.
[11]http://wiki.apache.org/hadoop/PoweredBy.

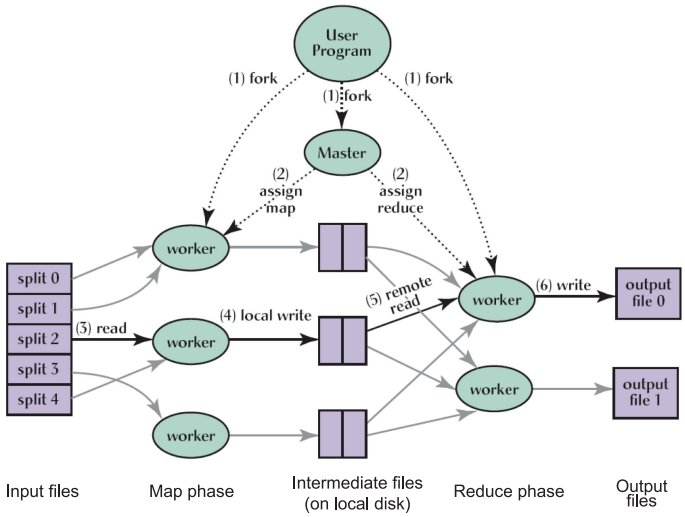


Fig. 2. An overview of the flow of execution of a MapReduce operation. Adaptation with permission of Dean and Ghemawat [2004].

distributed across multiple machines by automatically partitioning the input data into a set of M splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R). The number of partitions (R) and the partitioning function are specified by the user. Figure 2 illustrates an example of the overall flow of a MapReduce operation which goes through the following sequence of actions.

(1) The input data of the MapReduce program is split into M pieces and starts up many instances of the program on a cluster of machines.
(2) One of the instances of the program is elected to be the *master* copy while the rest are considered as *workers* that are assigned their work by the master copy. In particular, there are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one or more map tasks and/or reduce tasks.
(3) A worker who is assigned a map task processes the contents of the corresponding input split and generates key/value pairs from the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
(4) Periodically, the buffered pairs are written to local disk and partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
(5) When a reduce worker is notified by the master about these locations, it reads the buffered data from the local disks of the map workers, which is then sorted by the intermediate keys so that all occurrences of the same key are grouped together. The sorting operation is needed because typically many different keys map to the same reduce task.
(6) The reduce worker passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.

(7) When all map tasks and reduce tasks have been completed, the master program wakes up the user program. At this point, the MapReduce invocation in the user program returns the program control back to the user code.

During the execution process, the master pings every worker periodically. If no response is received from a worker within a certain amount of time, the master marks the worker as *failed*. Any map tasks marked *completed* or *in progress* by the worker are reset back to their initial idle state and therefore become eligible for scheduling by other workers. Completed map tasks are reexecuted on a task failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be reexecuted since their output is stored in a global file system.

3. EXTENSIONS AND ENHANCEMENTS OF THE MAPREDUCE FRAMEWORK

In practice, the basic implementation of MapReduce is very useful for handling data processing and data loading in a heterogenous system with many different storage systems. Moreover, it provides a flexible framework for the execution of more complicated functions than can be directly supported in SQL. However, this basic architecture suffers some limitations. Dean and Ghemawat [2010] reported about some possible improvements that can be incorporated into the MapReduce framework. Examples of these possible improvements include the following.

—MapReduce should take advantage of natural indices whenever possible.
—Most MapReduce output can be left unmerged since there is no benefit of merging them if the next consumer is just another MapReduce program.
—MapReduce users should avoid using inefficient textual formats.

In the following subsections we discuss some research efforts that have been conducted in order to deal with these challenges and the different improvements that have been made on the basic implementation of the MapReduce framework in order to achieve these goals.

3.1. Processing Join Operations

One main limitation of the MapReduce framework is that it does not support the joining of multiple datasets in one task. However, this can still be achieved with additional MapReduce steps. For example, users can map and reduce one dataset and read data from other datasets on-the-fly. Blanas et al. [2010] have reported about a study that evaluated the performance of different distributed join algorithms using the MapReduce framework. In particular, they have evaluated the following implementation strategies of distributed join algorithms.

—*Standard repartition join*. The two input relations are dynamically partitioned on the join key and the corresponding pairs of partitions are joined using the standard partitioned sort-merge join approach.
—*Improved repartition join*. One potential problem with the standard repartition join is that all the records for a given join key from both input relations have to be buffered. Therefore, when the key cardinality is small or when the data is highly skewed, all the records for a given join key may not fit in memory. The improved repartition join strategy fixes the buffering problem by introducing the following key changes.
  —In the map function, the output key is changed to a composite of the join key and the table tag. The table tags are generated in a way that ensures records from one input relation will be sorted ahead of those from the other input relation on a given join key.

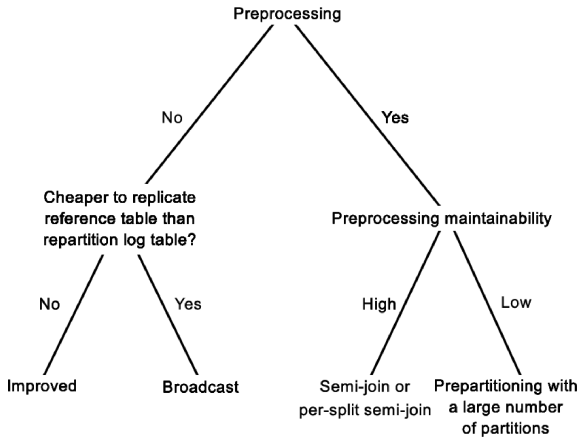


Fig. 3. Decision tree for choosing between various join strategies on the MapReduce framework. Adaptation with permission of Blanas et al. [2010].

  - —The partitioning function is customized so that the hashcode is computed from just the join key part of the composite key. This way records with the same join key are still assigned to the same reduce task.
  - —As records from the smaller input are guaranteed to be ahead of those from L for a given join key, only the records from the smaller input are buffered and the records of the larger input are streamed to generate the join output.
- —*Broadcast join*. Instead of moving both input relations across the network as in the repartition-based joins, the broadcast join approach moves only the smaller input relation so that it avoids the preprocessing sorting requirement of both input relations, and more importantly avoids the network overhead for moving the larger relation.
- —*Semi-join*. This join approach tries to avoid the problem of the broadcast join approach where it is possible to send many records of the smaller input relation across the network while they may not be actually referenced by any records in the other relation. It achieves this goal at the cost of an extra scan of the smaller input relation where it determines the set of unique join keys in the smaller relation, sends them to the other relation to specify the list of the actual referenced join keys, and then sends only these records across the network for executing the real execution of the join operation.
- —*Per-split semi-join*. This join approach tries to improve the semi-join approach with a further step to address the fact that not every record in the filtered version of the smaller relation will join with a particular split of the larger relation. Therefore, an extra process step is executed to determine the target split(s) of each filtered join key.

Figure 3 illustrates a decision tree that summarizes the trade-offs of the studied join strategies according to the results of that study. Based on statistics, such as the relative data size and the fraction of the join key referenced, this decision tree tries to determine what is the right join strategy for a given circumstance. If data is not preprocessed, the right join strategy depends on the size of the data transferred via the network. If the network cost of broadcasting an input relation R to every node is less expensive than transferring both R and projected L, then the broadcast join algorithm should be used. When preprocessing is allowed, semi-join, per-split semi-join, and directed join with sufficient partitions are the best choices. Semi-join and per-split semi-join offer further flexibility since their preprocessing steps are insensitive to how the log table is organized, and thus suitable for any number of reference tables. In addition,

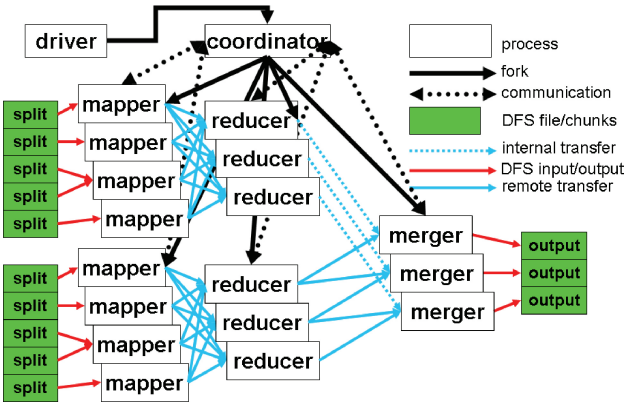


Fig. 4. An overview of the MapReduce-merge framework. Adaptation with permission of Yang et al. [2007].

the preprocessing steps of these two algorithms are less expensive since there is no shuffling of the log data.

To tackle the limitation of the extra processing requirements for performing join operations in the MapReduce framework, the *MapReduce-merge* model [Yang et al. 2007] has been introduced to enable the processing of multiple datasets. Figure 4 illustrates the framework of this model where the map phase transforms an input key/value pair $(k1, v1)$ into a list of intermediate key/value pairs $[(k2, v2)]$. The reduce function aggregates the list of values $[v2]$ associated with $k2$ and produces a list of values $[v3]$ which is also associated with k2. Note that inputs and outputs of both functions belong to the same lineage (α). Another pair of map and reduce functions produce the intermediate output $(k3, [v4])$ from another lineage (β). Based on keys $k2$ and $k3$, the merge function combines the two reduced outputs from different lineages into a list of key/value outputs $[(k4, v5)]$. This final output becomes a new lineage (γ). If $\alpha = \beta$ then this merge function does a self-merge which is similar to self-join in relational algebra. The main differences between the processing model of this framework and the original MapReduce is the production of a key/value list from the reduce function instead of just that of values. This change is introduced because the merge function requires input datasets to be organized (partitioned, then either sorted or hashed) by keys and these keys have to be passed into the function to be merged. In the original framework, the reduced output is final. Hence, users pack whatever is needed in $[v3]$ while passing $k2$ for the next stage is not required. Figure 5 illustrates a sample execution of the MapReduce-merge framework. In this example, there are two datasets *Employee* and *Department* where Employee's key attribute is `emp-id` and the Department's key is `dept-id`. The execution of this example query aims to join these two datasets and compute employee bonuses. On the left-hand side of Figure 5, a mapper reads Employee entries and computes a bonus for each entry. A reducer then sums up these bonuses for every employee and sorts them by `dept-id`, then `emp-id`. On the right-hand side, a mapper reads Department entries and computes bonus adjustments. A reducer then sorts these department entries. At the end, a merger matches the output records from the two reducers on `dept-id` and applies a department-based bonus adjustment on employee bonuses. Yang and Parker [2009] have also proposed an approach for improving the MapReduce-merge framework by adding a new primitive called *Traverse*. This primitive can process index file entries recursively, select data partitions based on query conditions, and feed only selected partitions to other primitives.

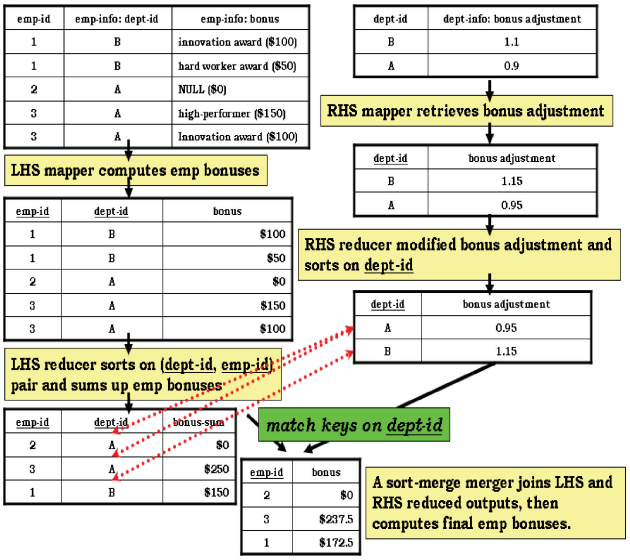


Fig. 5. A sample execution of the MapReduce-merge framework. Adaptation with permission of Yang et al. [2007].

The *Map-Join-Reduce* [Jiang et al. 2011] represents another approach that has been introduced with a filtering-join-aggregation programming model as an extension of the standard MapReduce's filtering-aggregation programming model. In particular, in addition to the standard mapper and reducer operation of the standard MapReduce framework, they introduce a third operation, join (called joiner), to the framework. Hence, to join multiple datasets for aggregation, users specify a set of *join*() functions and the join order between them. Then, the runtime system automatically joins the multiple input datasets according to the join order and invokes *join*() functions to process the joined records. They have also introduced a one-to-many shuffling strategy which shuffles each intermediate key/value pair to many joiners at one time. Using a tailored partition strategy, they can utilize the one-to-many shuffling scheme to join multiple datasets in one phase instead of a sequence of MapReduce jobs. The runtime system for executing a Map-Join-Reduce job launches two kinds of processes: *MapTask*, and *ReduceTask*. Mappers run inside the MapTask process while joiners and reducers are invoked inside the ReduceTask process. Therefore, Map-Join-Reduce's process model allows for the pipelining of intermediate results between joiners and reducers since joiners and reducers are run inside the same ReduceTask process.

Afrati and Ullman [2010, 2011] have presented another approach to improve the join phase in the MapReduce framework. The approach aims to optimize the communication cost by focusing on selecting the most appropriate attributes that are used to partition and replicate the data among the reduce process. Therefore, it begins by identifying the *map-key*, the set of attributes that identify the Reduce process to which a Map process must send a particular tuple. Each attribute of the map-key gets a "share" which is the number of buckets into which its values are hashed, to form a component of the identifier of a Reduce process. Relations have their tuples replicated in limited fashion of which the degree of replication depends on the shares for those map-key attributes that are missing from their schema. The approach considers two important special join cases: *chain* joins (represents a sequence of 2-way join operations where the output of one operation in this sequence is used as an input to another operation in a pipelined fashion) and *star* joins (represents joining of a large fact table with several smaller

dimension tables). In each case, the proposed algorithm is able to determine the map-key and determine the shares that yield the least replication. The proposed approach is not always superior to the conventional way of using MapReduce to implement joins. However, there are some cases where the proposed approach results in clear wins such as: (1) analytic queries in which a very large fact table is joined with smaller dimension tables; (2) queries involving paths through graphs with high out-degree, such as the Web or a social network.

3.2. Supporting Iterative Processing

The basic MapReduce framework does not directly support these iterative data analysis applications. Instead, programmers must implement iterative programs by manually issuing multiple MapReduce jobs and orchestrating their execution using a driver program. In practice, there are two key problems with manually orchestrating an iterative program in MapReduce.

—Even though much of the data may be unchanged from iteration to iteration, the data must be reloaded and reprocessed at each iteration, wasting I/O, network bandwidth, and CPU resources.
—The termination condition may involve the detection of when a fixpoint has been reached. This condition may itself require an extra MapReduce job on each iteration, again incurring overhead in terms of scheduling extra tasks, reading extra data from disk, and moving data across the network.

The *HaLoop* system [Bu et al. 2010] is designed to support iterative processing on the MapReduce framework by extending the basic MapReduce framework with two main functionalities.

(1) One functionality is for caching the invariant data in the first iteration and then reusing them in later iterations.
(2) One functionality is for caching the reducer outputs, which makes checking for a fixpoint more efficient, without an extra MapReduce job.

In order to accommodate the requirements of iterative data analysis applications, HaLoop has incorporated the following changes to the basic Hadoop MapReduce framework.

—It exposes a new application programming interface to users that simplifies the expression of iterative MapReduce programs.
—HaLoop's master node contains a new loop control module that repeatedly starts new map-reduce steps that compose the loop body until a user-specified stopping condition is met.
—It uses a new task scheduler that leverages data locality.
—It caches and indices application data on slave nodes. In principle, the task tracker not only manages task execution but also manages caches and indices on the slave node and redirects each task's cache and index accesses to the local file system.

In principle, HaLoop relies on the same file system and has the same task queue structure as Hadoop but the task scheduler and task tracker modules are modified, and the loop control, caching, and indexing modules are newly introduced to the architecture. The task tracker not only manages task execution but also manages caches and indices on the slave node, and redirects each task's cache and index accesses to the local file system.

In the MapReduce framework, each map or reduce task contains its portion of the input data and the task runs by performing the map/reduce function on its input data records where the life cycle of the task ends when finishing the processing of all

the input data records has been completed. The *iMapReduce* framework [Zhang et al. 2012] supports the feature of iterative processing by keeping alive each map and reduce task during the whole iterative process. In particular, when all of the input data of a persistent task are parsed and processed, the task becomes dormant, waiting for the new updated input data. For a map task, it waits for the results from the reduce tasks and is activated to work on the new input records when the required data from the reduce tasks arrive. For the reduce tasks, they wait for the map tasks' output and are activated synchronously as in MapReduce. Jobs can terminate their iterative process in one of two ways.

(1) *Defining fixed number of iterations*. The iterative algorithm stops after it iterates n times.
(2) *Bounding the distance between two consecutive iterations*. The iterative algorithm stops when the distance is less than a threshold.

The iMapReduce runtime system does the termination check after each iteration. To terminate the iterations by a fixed number of iterations, the persistent map/reduce task records its iteration number and terminates itself when the number exceeds a threshold. To bound the distance between the output from two consecutive iterations, the reduce tasks can save the output from two consecutive iterations and compute the distance. If the termination condition is satisfied, the master will notify all the map and reduce tasks to terminate their execution.

Other projects have been implemented for supporting iterative processing on the MapReduce framework. For example, *Twister*[12] is a MapReduce runtime with an extended programming model that supports iterative MapReduce computations efficiently [Ekanayake et al. 2010]. It uses a publish/subscribe messaging infrastructure for communication and data transfers, and supports long-running map/reduce tasks. In particular, it provides programming extensions to MapReduce with broadcast and scatter-type data transfers. Microsoft has also developed a project that provides an iterative MapReduce runtime for Windows Azure called *Daytona*[13].

3.3. Data and Process Sharing

With the emergence of cloud computing, the use of an analytical query processing infrastructure (e.g., Amazon EC2) can be directly mapped to *monetary* value. Taking into account that different MapReduce jobs can perform similar work, there could be many opportunities for sharing the execution of their work. Thus, this sharing can reduce the overall amount of work, which consequently leads to the reduction of the monetary charges incurred while utilizing the resources of the processing infrastructure. The *MRShare* system [Nykiel et al. 2010] has been presented as a sharing framework which is tailored to transform a batch of queries into a new batch that will be executed more efficiently by merging jobs into groups and evaluating each group as a single query. Based on a defined cost model, they described an optimization problem that aims to derive the optimal grouping of queries in order to avoid performing redundant work and thus resulting in significant savings on both processing time and money. In particular, the approach considers exploiting the following sharing opportunities.

—*Sharing Scans*. To share scans between two mapping pipelines M_i and M_j, the input data must be the same. In addition, the key/value pairs should be of the same type. Given that, it becomes possible to merge the two pipelines into a single pipeline and scan the input data only once. However, it should be noted that such combined

[12]http://www.iterativemapreduce.org/.
[13]http://research.microsoft.com/en-us/projects/daytona/.

mapping will produce two streams of output tuples (one for each mapping pipeline M_i and M_j) . In order to distinguish the streams at the reducer stage, each tuple is tagged with a `tag()` part. This tagging part is used to indicate the origin mapping pipeline during the reduce phase.

—*Sharing Map Output*. If the map output key and value types are the same for two mapping pipelines M_i and M_j then the map output streams for M_i and M_j can be shared, in particular, if Map_i and Map_j are applied to each input tuple. Then, the map output tuples coming only from Map_i are tagged with `tag(i)` only. If a map output tuple was produced from an input tuple by both Map_i and Map_j, it is then tagged by `tag(i)+tag(j)`. Therefore, any overlapping parts of the map output will be shared. In principle, producing a smaller map output leads to savings on sorting and copying intermediate data over the network.

—*Sharing Map Functions*. Sometimes the map functions are identical and thus they can be executed once. At the end of the map stage two streams are produced, each tagged with its job tag. If the map output is shared, then clearly only one stream needs to be generated. Even if only some filters are common in both jobs, it is possible to share parts of map functions.

In practice, sharing scans and sharing map output yields I/O savings while sharing map functions (or parts of them) additionally yields CPU savings.

While the *MRShare* system focuses on sharing the processing between queries that are executed concurrently, the *ReStore* system [Elghandour and Aboulnaga 2012a, 2012b] has been introduced so that it can enable the queries that are submitted at different times to share the intermediate results of previously executed jobs and reusing them for future submitted jobs to the system. In particular, each MapReduce job produces output that is stored in the distributed file system used by the MapReduce system (e.g., HDFS). These intermediate results are kept (for a defined period) and managed so that they can be used as input by subsequent jobs. ReStore can make use of whole jobs or subjobs reuse opportunities. To achieve this goal, the ReStore consists of two main components.

—*Repository of MapReduce job outputs*. It stores the outputs of previously executed MapReduce jobs and the physical plans of these jobs.

—*Plan matcher and rewriter*. Its aim is to find physical plans in the repository that can be used to rewrite the input jobs using the available matching intermediate results.

In principle, the approach of the *ReStore* system can be viewed as analogous to the steps of building and using materialized views for relational databases [Halevy 2001].

3.4. Support of Data Indices and Column Storage

One of the main limitations of the original implementation of the MapReduce framework is that it is designed in a way that the jobs can only scan the input data in a sequential-oriented fashion. Hence, the query processing performance of the MapReduce framework is unable to match the performance of a well-configured parallel DBMS [Pavlo et al. 2009]. In order to tackle this challenge, Dittrich et al. [2010] have presented the *Hadoop++* system which aims to boost the query performance of the Hadoop system without changing any of the system internals. They achieved this goal by injecting their changes through User-Defined Functions (UDFs) which only affect the Hadoop system from inside without any external effect. In particular, they introduce the following main changes.

—*Trojan Index*. The original Hadoop implementation does not provide index access due to the lack of a priori knowledge of the schema and the MapReduce jobs being executed. Hence, the Hadoop++ system is based on the assumption that if we know the

schema and the anticipated MapReduce jobs, then we can create appropriate indices for the Hadoop tasks. In particular, trojan index is an approach to integrate indexing capability into Hadoop in a noninvasive way. These indices are created during the data loading time and thus have no penalty at query time. Each trojan index provides an optional index access path which can be used for selective MapReduce jobs. The scan access path can still be used for other MapReduce jobs. These indices are created by injecting appropriate UDFs inside the Hadoop implementation. Specifically, the main features of trojan indices can be summarized as follows.

—*No External Library or Engine*. Trojan indices integrate indexing capability natively into the Hadoop framework without imposing a distributed SQL-query engine on top of it.
—*Noninvasive*. They do not change the existing Hadoop framework. The index structure is implemented by providing the right UDFs.
—*Optional Access Path*. They provide an optional index access path which can be used for selective MapReduce jobs. However, the scan access path can still be used for other MapReduce jobs.
—*Seamless Splitting*. Data indexing adds an index overhead for each data split. Therefore, the logical split includes the data as well as the index as it automatically splits the indexed data at logical split boundaries.
—*Partial Index*. Trojan index need not be built on the entire split. However, it can be built on any contiguous subset of the split as well.
—*Multiple Indexes*. Several trojan indexes can be built on the same split. However, only one of them can be the primary index. During query processing, an appropriate index can be chosen for data access based on the logical query plan and the cost model.

—*Trojan Join*. Similar to the idea of the trojan index, the Hadoop++ system assumes that if we know the schema and the expected workload, then we can copartition the input data during the loading time. In particular, given any two input relations, they apply the same partitioning function on the join attributes of both the relations at data loading time and place the cogroup pairs, having the same join key from the two relations, on the same split and hence on the same node. As a result, join operations can be then processed locally within each node at query time. Implementing the trojan joins does not require any changes to be made to the existing implementation of the Hadoop framework. The only changes are made on the internal management of the data splitting process. In addition, trojan indices can be freely combined with trojan joins.

The design and implementation of a column-oriented and binary backend storage format for Hadoop has been presented in Floratou et al. [2011]. In general, a straightforward way to implement a column-oriented storage format for Hadoop is to store each column of the input dataset in a separate file. However, this raises two main challenges.

—It requires generating roughly equal-sized splits so that a job can be effectively parallelized over the cluster.
—It needs to ensure that the corresponding values from different columns in the dataset are colocated on the same node running the map task.

The first challenge can be tackled by horizontally partitioning the dataset and storing each partition in a separate subdirectory. The second challenge is harder to tackle because of the default 3-way block-level replication strategy of HDFS that provides fault tolerance on commodity servers but does not provide any colocation guarantees. Floratou et al. [2011] tackle this challenge by implementing a modified HDFS block placement policy which guarantees that the files corresponding to the different columns

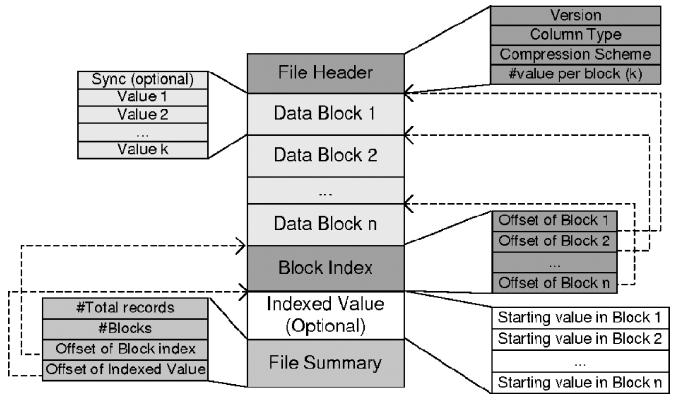


Fig. 6. An example structure of *CFile*. Adaptation with permission of Lin et al. [2011].

of a split are always colocated across replicas. Hence, when reading a dataset, the column input format can actually assign one or more split directories to a single split and the column files of a split directory are scanned sequentially where the records are reassembled using values from corresponding positions in the files. A lazy record construction technique is used to mitigate the deserialization overhead in Hadoop, as well as to eliminate unnecessary disk I/O. The basic idea behind lazy record construction is to deserialize only those columns of a record that are actually accessed in a map function. Each column of the input dataset can be compressed using one of the following compression schemes.

(1) *Compressed Blocks*. This scheme uses a standard compression algorithm to compress a block of contiguous column values. Multiple compressed blocks may fit into a single HDFS block. A header indicates the number of records in a compressed block and the block's size. This allows the block to be skipped if no values are accessed in it. However, when a value in the block is accessed, the entire block needs to be decompressed.
(2) *Dictionary Compressed Skip List*. This scheme is tailored for map-typed columns. It takes advantage of the fact that the keys used in maps are often strings that are drawn from a limited universe. Such strings are well suited for dictionary compression. A dictionary is built of keys for each block of map values and stores the compressed keys in a map using a skip list format. The main advantage of this scheme is that a value can be accessed without having to decompress an entire block of values.

One advantage of this approach is that adding a column to a dataset is not an expensive operation. This can be done by simply placing an additional file for the new column in each of the split directories. On the other hand, a potential disadvantage of this approach is that the available parallelism may be limited for smaller datasets. Maximum parallelism is achieved for a MapReduce job when the number of splits is at least equal to the number of map tasks.

The *Llama* system [Lin et al. 2011] has introduced another approach of providing column storage support for the MapReduce framework. In this approach, each imported table is transformed into column groups where each group contains a set of files representing one or more columns. Llama introduced a column-wise format for Hadoop, called *CFile*, where each file can contain multiple data blocks and each block of the file contains a fixed number of records (Figure 6). However, the size of each logical block may vary since records can be variable-sized. Each file includes a block index, which is stored after all data blocks, stores the offset of each block, and is used to locate a

specific block. In order to achieve storage efficiency, Llama uses block-level compression by using any of the well-known compression schemes. In order to improve the query processing and the performance of join operations, Llama columns are formed into correlation groups to provide the basis for the vertical partitioning of tables. In particular, it creates multiple vertical groups where each group is defined by a collection of columns; one of them is specified as the sorting column. Initially, when a new table is imported into the system, a basic vertical group is created which contains all the columns of the table and sorted by the table's primary key by default. In addition, based on statistics of query patterns, some auxiliary groups are dynamically created or discarded to improve the query performance. The *Clydesdale* system [Kaldewey et al. 2012; Balmin et al. 2012], a system which has been implemented for targeting workloads where the data fits a star schema, uses *CFile* for storing its fact tables. It also relies on tailored join plans and a block iteration mechanism [Zukowski et al. 2005] for optimizing the execution of its target workloads.

RCFile [He et al. 2011] (Record Columnar File) is another data placement structure that provides column-wise storage for the Hadoop file system (HDFS). In RCFile, each table is first stored as horizontally partitioned into multiple row groups where each row group is then vertically partitioned so that each column is stored independently. In particular, each table can have multiple HDFS blocks where each block organizes records with the basic unit of a row group. Depending on the row group size and the HDFS block size, an HDFS block can have only one or multiple row groups. In particular, a row group contains the following three sections.

(1) The *sync marker* is placed in the beginning of the row group and mainly used to separate two continuous row groups in an HDFS block.
(2) A metadata header stores the information items on how many records are in this row group, how many bytes are in each column, and how many bytes are in each field in a column.
(3) The table data section is actually a column store where all the fields in the same column are stored continuously together.

RCFile utilizes a column-wise data compression within each row group and provides a lazy decompression technique to avoid unnecessary column decompression during query execution. In particular, the metadata header section is compressed using the *RLE* (Run Length Encoding) algorithm. The table data section is not compressed as a whole unit. However, each column is independently compressed with the *Gzip* compression algorithm. When processing a row group, RCFile does not need to fully read the whole content of the row group into memory. However, it only reads the metadata header and the needed columns in the row group for a given query and thus it can skip unnecessary columns and gain the I/O advantages of a column store. The metadata header is always decompressed and held in memory until RCFile processes the next row group. However, RCFile does not decompress all the loaded columns and uses a lazy decompression technique where a column will not be decompressed in memory until RCFile has determined that the data in the column will be really useful for query execution.

The notion of *trojan data layout* has been coined in Jindal et al. [2011] which exploits the existing data block replication in HDFS to create different trojan layouts on a per-replica basis. This means that, rather than keeping all data block replicas in the same layout, it uses *different* trojan layouts for each replica which is optimized for a different subclass of queries. As a result, every incoming query can be scheduled to the most suitable data block replica. In particular, trojan layouts change the internal organization of a data block and not among data blocks. They colocate attributes together according to query workloads by applying a column grouping algorithm which uses an interestingness measure that denotes how well a set of attributes speeds up most or all queries in a workload. The column groups are then packed in order to maximize

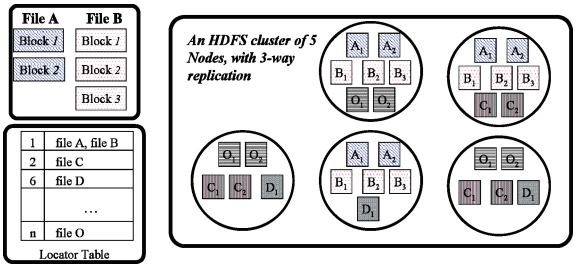


Fig. 7. Example file colocation in CoHadoop. Adaptation with permission of Eltabakh et al. [2011].

the total interestingness of data blocks. At query time, an incoming MapReduce job is transparently adapted to query the data block replica that minimizes the data access time. The map tasks are then routed of the MapReduce job to the data nodes storing such data block replicas.

3.5. Effective Data Placement

In the basic implementation of the Hadoop project, the objective of the data placement policy is to achieve good load balance by distributing the data evenly across the data servers, independently of the intended use of the data. This simple data placement policy works well with most Hadoop applications that access just a *single* file. However, there are some other applications that process data from *multiple* files which can get a significant boost in performance with customized strategies. In these applications, the absence of data colocation increases the data shuffling costs, increases the network overhead, and reduces the effectiveness of data partitioning. *CoHadoop* [Eltabakh et al. 2011] is a lightweight extension to Hadoop which is designed to enable colocating related files at the file system level while at the same time retaining the good load-balancing and fault-tolerance properties. It introduces a new file property to identify related data files and modify the data placement policy of Hadoop to colocate copies of those related files in the same server. These changes are designed in a way to retain the benefits of Hadoop, including load balancing and fault tolerance. In principle, CoHadoop provides a generic mechanism that allows applications to control data placement at the file system level. In particular, a new file-level property called a *locator* is introduced and the Hadoop's data placement policy is modified so that it makes use of this locator property. Each locator is represented by a unique value (ID) where each file in HDFS is assigned to at most one locator and many files can be assigned to the same locator. Files with the same locator are placed on the same set of data nodes, whereas files with no locator are placed via Hadoop's default strategy. It should be noted that this colocation process involves all data blocks, including replicas. Figure 7 shows an example of colocating two files, A and B, via a common locator. All of A's two HDFS blocks and B's three blocks are stored on the same set of data nodes. To manage the locator information and keep track of collocated files, CoHadoop introduces a new data structure, *the locator table*, which stores a mapping of locators to the list of files that share this locator. In practice, the CoHadoop extension enables a wide variety of applications to exploit data colocation by simply specifying related files such as: colocating log files with reference files for joins, colocating partitions for grouping and aggregation, colocating index files with their data files, and colocating columns of a table.

3.6. Pipelining and Streaming Operations

The original implementation of the MapReduce framework has been designed in a way that the entire output of each map and reduce task is *materialized* into a local file before

it can be consumed by the next stage. This materialization step allows for the implementation of a simple and elegant checkpoint/restart fault-tolerance mechanism. The *MapReduce online* approach [Condie et al. 2010a, 2010b] has been proposed as a modified architecture of the MapReduce framework in which intermediate data is *pipelined* between operators while preserving the programming interfaces and fault-tolerance models of previous MapReduce frameworks. This pipelining approach provides important advantages to the MapReduce framework, such as described next.

—The reducers can begin their processing of the data as soon as it is produced by mappers. Therefore, they can generate and refine an approximation of their final answer during the course of execution. In addition, they can provide initial estimates of the results several orders of magnitude faster than the final results.
—It widens the domain of problems to which MapReduce can be applied. For example, it facilitates the ability to design MapReduce jobs that run continuously, accepting new data as it arrives and analyzing it immediately (continuous queries). This allows MapReduce to be used for applications such as event monitoring and stream processing.
—Pipelining delivers data to downstream operators more promptly, which can increase opportunities for parallelism, improve utilization, and reduce response time.

In this approach, each reduce task contacts every map task upon initiation of the job and opens a TCP socket which will be used to pipeline the output of the map function. As each map output record is produced, the mapper determines which partition (reduce task) the record should be sent to, and immediately sends it via the appropriate socket. A reduce task accepts the pipelined data it receives from each map task and stores it in an in-memory buffer. Once the reduce task learns that every map task has completed, it performs a final merge of all the sorted runs. In addition, the reduce tasks of one job can optionally pipeline their output directly to the map tasks of the next job, sidestepping the need for expensive fault-tolerant storage in HDFS for what amounts to a temporary file. However, the computation of the reduce function from the previous job and the map function of the next job cannot be overlapped as the final result of the reduce step cannot be produced until all map tasks have completed, which prevents effective pipelining. Therefore, the reducer treats the output of a pipelined map task as *tentative* until the JobTracker informs the reducer that the map task has committed successfully. The reducer can merge together spill files generated by the same uncommitted mapper, but will not combine those spill files with the output of other map tasks until it has been notified that the map task has committed. Thus, if a map task fails, each reduce task can ignore any tentative spill files produced by the failed map attempt. The JobTracker will take care of scheduling a new map task attempt, as in standard Hadoop. In principle, the main limitation of the *MapReduce online* approach is that it is based on HDFS. Therefore, it is not suitable for streaming applications, in which data streams have to be processed without any disk involvement. A similar approach has been presented in Logothetis and Yocum [2008] which defines an *incremental* MapReduce job as one that processes data in large batches of tuples and runs continuously according to a specific window range and slide of increment. In particular, it produces a MapReduce result that includes all data within a window (of time or data size) every slide and considers landmark MapReduce jobs where the trailing edge of the window is fixed and the system incorporates new data into the existing result. Map functions are trivially continuous, and process data on a tuple-by-tuple basis. However, before the reduce function may process the mapped data, the data must be partitioned across the reduce operators and sorted. When the map operator first receives a new key-value pair, it calls the map function and inserts the result into the latest increment in the map results. The operator then assigns output key-value

pairs to reduce tasks, grouping them according to the partition function. Continuous reduce operators participate in the sort as well, grouping values by their keys before calling the reduce function.

The *Incoop* system [Bhatotia et al. 2011] has been introduced as a MapReduce implementation that has been adapted for incremental computations. It detects the changes on the input datasets and enables the automatic update of the outputs of the MapReduce jobs by employing a fine-grained result reuse mechanism. In particular, it allows MapReduce programs which are not designed for incremental processing to be executed transparently in an incremental manner. To achieve this goal, the design of Incoop introduces new techniques that are incorporated into the Hadoop MapReduce framework. For example, instead of relying on HDFS to store the input to MapReduce jobs, Incoop devises a file system called *Inc-HDFS* (Incremental HDFS) that provides mechanisms to identify similarities in the input data of consecutive job runs. In particular, Inc-HDFS splits the input into chunks whose boundaries depend on the file contents so that small changes to input do not change all chunk boundaries. Therefore, this partitioning mechanism can maximize the opportunities for reusing results from previous computations, while preserving compatibility with HDFS by offering the same interface and semantics. In addition, Incoop controls the granularity of tasks so that large tasks can be divided into smaller subtasks that can be reused even when the large tasks cannot. Therefore, it introduces a new *contraction phase* that leverages *combiner* functions to reduce the network traffic by anticipating a small part of the processing done by the reducer tasks and control their granularity. Furthermore, Incoop improves the effectiveness of memoization by implementing an affinity-based scheduler that applies a work stealing algorithm to minimize the amount of data movement across machines. This modified scheduler strikes a balance between exploiting the locality of previously computed results and executing tasks on any available machine to prevent straggling effects. On the runtime, instances of incremental Map tasks take advantage of previously stored results by querying the memoization server. If they find that the result has already been computed, they fetch the result from the location of their memoized output and conclude. Similarly, the results of a Reduce task are remembered by storing them persistently and locally where a mapping from a collision-resistant hash of the input to the location of the output is inserted in the memoization server.

The *DEDUCE* system [Kumar et al. 2010] has been presented as a middleware that attempts to combine real-time stream processing with the capabilities of a large-scale data analysis framework like MapReduce. In particular, it extends the *IBM's System S* stream processing engine and augments its capabilities with those of the MapReduce framework. In this approach, the input dataset to the MapReduce operator can be either prespecified at compilation time or could be provided at runtime as a punctuated list of files or directories. Once the input data is available, the MapReduce operator spawns a MapReduce job and produces a list of punctuated list of files or directories, which point to the output data. Therefore, a MapReduce operator can potentially spawn multiple MapReduce jobs over the application lifespan but such jobs are spawned only when the preceding job (if any) has completed its execution. Hence, multiple jobs can be cascaded together to create a data flow of MapReduce operators where the output from the MapReduce operators can be read to provide updates to the stream processing operators.

3.7. System Optimizations

In general, running a single program in a MapReduce framework may require tuning a number of parameters by users or system administrators. The settings of these parameters control various aspects of job behavior during execution such as memory allocation and usage, concurrency, I/O optimization, and network bandwidth usage. The submitter

of a Hadoop job has the option to set these parameters either using a program-level interface or through XML configuration files. For any parameter whose value is not specified explicitly during job submission, default values, either shipped along with the system or specified by the system administrator, are used [Babu 2010]. Users can run into performance problems because they do not know how to set these parameters correctly, or because they do not even know that these parameters exist. Herodotou and Babu [2011] have focused on the optimization opportunities presented by the large space of configuration parameters for these programs. They introduced a *profiler* component to collect detailed statistical information from unmodified MapReduce programs and a *what-if* engine for fine-grained cost estimation. In particular, the profiler component is responsible for two main aspects.

(1) It captures information at the fine granularity of phases within the map and reduces tasks of a MapReduce job execution. This information is crucial to the accuracy of decisions made by the what-if engine and the cost-based optimizer components.
(2) It uses dynamic instrumentation to collect runtime monitoring information from unmodified MapReduce programs. The dynamic nature means that monitoring can be turned on or off on demand.

The what-if engine's accuracy comes from how it uses a mix of simulation and model-based estimation at the phase level of MapReduce job execution [Herodotou 2011; Herodotou et al. 2011a, 2011b]. For a given MapReduce program, the role of the cost-based optimizer component is to enumerate and search efficiently through the high-dimensional space of configuration parameter settings, making appropriate calls to the what-if engine, in order to find a good configuration setting. It clusters parameters into lower-dimensional subspaces such that the globally optimal parameter setting in the high-dimensional space can be generated by composing the optimal settings found for the subspaces. *Stubby* [Lim et al. 2012] has been presented as a cost-based optimizer for MapReduce workflows that searches through the subspace of the full plan space that can be enumerated correctly and costed based on the information available in any given setting. Stubby enumerates the plan space based on plan-to-plan transformations and an efficient search algorithm.

The *Manimal* system [Jahani et al. 2011; Cafarella and Ré 2010] is designed as a static analysis-style mechanism for detecting opportunities for applying relational-style optimizations in MapReduce programs. Like most programming language optimizers, it is a best-effort system where it does not guarantee that it will find every possible optimization and it only indicates an optimization when it is entirely safe to do so. In particular, the analyzer component of the system is responsible for examining the MapReduce program and sends the resulting optimization descriptor to the optimizer component. In addition, the analyzer also emits an index generation program that can yield a B+Tree of the input file. The optimizer uses the optimization descriptor, plus a catalog of precomputed indexes, to choose an optimized execution plan, called an execution descriptor. This descriptor, plus a potentially modified copy of the user's original program, is then sent for execution on the Hadoop cluster. These steps are performed transparently from the user where the submitted program does not need to be modified by the programmer in any way. In particular, the main task of the analyzer is to produce a set of optimization descriptors which enable the system to carry out a phase roughly akin to logical rewriting of query plans in a relational database. The descriptors characterize a set of potential modifications that remain logically identical to the original plan. The catalog is a simple mapping from a filename to zero or more (X, O) pairs, where X is an index file and O is an optimization descriptor. The optimizer examines the catalog to see if there is any entry for the input file. If not, then it simply indicates that Manimal should run the unchanged user program without any

optimization. If there is at least one entry for the input file, and a catalog-associated optimization descriptor is compatible with analyzer output, then the optimizer can choose an execution plan that takes advantage of the associated index file.

A key feature of MapReduce is that it automatically handles failures, hiding the complexity of fault tolerance from the programmer. In particular, if a node crashes, MapReduce automatically restarts the execution of its tasks. In addition, if a node is available but is performing poorly, MapReduce runs a speculative copy of its task (backup task) on another machine to finish the computation faster. Without this mechanism of speculative execution, a job would be as slow as the misbehaving task. This situation can arise for many reasons, including faulty hardware and system misconfiguration. On the other hand, launching too many speculative tasks may take away resources from useful tasks. Therefore, the accuracy in estimating the progress and time remaining of long-running jobs is an important challenge for a runtime environment like the MapReduce framework. In particular, this information can play an important role in improving resource allocation, enhancing the task scheduling, enabling query debugging, or tuning the cluster configuration. The *ParaTimer* system [Morton et al. 2010a, 2010b] has been proposed to tackle this challenge. In particular, ParaTimer provides techniques for handling several challenges including failures and data skew. To handle unexpected changes in query execution times such as those due to failures, ParaTimer provides users with a set of time-remaining estimates that correspond to the predicted query execution times in different scenarios (i.e., a single worst-case failure, or data skew at an operator). Each of these indicators can be annotated with the scenario to which it corresponds, giving users a detailed picture of possible expected behaviors. To achieve this goal, ParaTimer estimates time remaining by breaking queries into pipelines where the time remaining for each pipeline is estimated by considering the work to be done and the speed at which that work will be performed, taking (time-varying) parallelism into account. To get processing speeds, ParaTimer relies on earlier debug runs of the same query on input data samples generated by the user. In addition, ParaTimer identifies the critical path in a query plan where it then estimates progress along that path, effectively ignoring other paths. Zaharia et al. [2008] have presented an approach to estimate the progress of MapReduce tasks within environments of clusters with heterogenous hardware configuration. In these environments, choosing the node on which to run a speculative task is as important as choosing the task. They proposed an algorithm for speculative execution called *LATE* (Longest Approximate Time to End) which is based on three principles: prioritizing tasks to speculate, selecting fast nodes on which to run, and capping speculative tasks to prevent thrashing. In particular, the algorithm speculatively executes the task that it suspects will finish farthest into the future, because this task provides the greatest opportunity for a speculative copy to overtake the original and reduce the job's response time. To really get the best chance of beating the original task with the speculative task, the algorithm only launches speculative tasks on fast nodes (and not the first available node). The *RAFT* (*R*ecovery *A*lgorithms for *F*ast *T*racking) system [Quiané-Ruiz et al. 2011a, 2011b] has been introduced as a part of the *Hadoop++* system [Dittrich et al. 2010], for tracking and recovering MapReduce jobs under task or node failures. In particular, RAFT uses two main checkpointing mechanisms: *local checkpointing* and *query metadata checkpointing*. On the one hand, the main idea of local checkpointing is to utilize intermediate results, which are by default persisted by Hadoop, as checkpoints of ongoing task progress computation. In general, map tasks spill buffered intermediate results to local disk whenever the output buffer is on the verge to overflow. RAFT exploits this spilling phase to piggy-back checkpointing metadata on the latest spill of each map task. For each checkpoint, RAFT stores a triplet of metadata that includes the *taskID* which represents a unique task identifier, *spillID* which represents the local path to the

spilled data, and *offset* which specifies the last byte of input data that was processed in that spill. To recover from a task failure, the RAFT scheduler reallocates the failed task to the same node that was running the task. Then, the node resumes the task from the last checkpoint and reuses the spills previously produced for the same task. This simulates a situation where previous spills appear as if they were just produced by the task. In case that there is no local checkpoint available, the node recomputes the task from the beginning. On the other hand, the idea behind query metadata checkpointing is to push intermediate results to reducers as soon as map tasks are completed and to keep track of those incoming key-value pairs that produce local partitions and hence that are not shipped to another node for processing. Therefore, in case of a node failure, the RAFT scheduler can recompute local partitions.

4. SYSTEMS OF DECLARATIVE INTERFACES FOR THE MAPREDUCE FRAMEWORK

For programmers, a key appealing feature in the MapReduce framework is that there are only two main high-level declarative primitives (*map* and *reduce*) that can be written in any programming language of choice and without worrying about the details of their parallel execution. On the other hand, the MapReduce programming model has its own limitations such as given next.

—Its one-input data format (key/value pairs) and two-stage data flow is extremely rigid. As we have previously discussed, to perform tasks that have a different data flow (e.g., joins or n stages) would require the need to devise inelegant workarounds.
—Custom code has to be written for even the most common operations (e.g., projection and filtering) which leads to the fact that the code is usually difficult to reuse and maintain unless the users build and maintain their own libraries with the common functions they use for processing their data.

Moreover, many programmers could be unfamiliar with the MapReduce framework and they would prefer to use SQL (in which they are more proficient) as a high-level declarative language to express their task while leaving all of the execution optimization details to the backend engine. In addition, it is beyond doubt that high-level language abstractions enable the underlying system to perform automatic optimization. In the following subsection we discuss research efforts that have been proposed to tackle these problems and add SQL-like interfaces on top of the MapReduce framework.

4.1. Sawzall

Sawzall [Pike et al. 2005] is a scripting language used at Google on top of MapReduce. A Sawzall program defines the operations to be performed on a single record of the data. There is nothing in the language to enable examining multiple input records simultaneously, or even to have the contents of one input record influence the processing of another. The only output primitive in the language is the *emit* statement, which sends data to an external aggregator (e.g., Sum, Average, Maximum, Minimum) that gathers the results from each record after which the results are then correlated and processed. The authors argue that aggregation is done outside the language for a couple of reasons: (1) A more traditional language can use the language to correlate results but some of the aggregation algorithms are sophisticated and are best implemented in a native language and packaged in some form. (2) Drawing an explicit line between filtering and aggregation enables a high degree of parallelism and hides the parallelism from the language itself.

Figure 8 depicts an example Sawzall program where the first three lines declare the aggregators *count*, *total*, and *sum of squares*. The keyword *table* introduces an aggregator type which are called tables in Sawzall even though they may be singletons. These particular tables are *sum* tables which add up the values emitted to them, *ints* or

```
count: table sum of int;
total: table sum of float;
sumOfSquares: table sum of float;
x: float = input;
emit count $<$- 1;
emit total $<$- x;
emit sumOfSquares $<$- x * x;
```

Fig. 8. An example sawzall program.

floats as appropriate. The Sawzall language is implemented as a conventional compiler, written in C++, whose target language is an interpreted instruction set, or bytecode. The compiler and the bytecode interpreter are part of the same binary, so the user presents source code to Sawzall and the system executes it directly. It is structured as a library with an external interface that accepts source code which is then compiled and executed, along with bindings to connect to externally provided aggregators. The datasets of Sawzall programs are often stored in Google File System (GFS) [Ghemawat et al. 2003]. The business of scheduling a job to run on a cluster of machines is handled by a software called *Workqueue* which creates a large-scale time sharing system out of an array of computers and their disks. It schedules jobs, allocates resources, reports status, and collects the results.

Google has also developed *FlumeJava* [Chambers et al. 2010], a Java library for developing and running data-parallel pipelines on top of MapReduce. FlumeJava is centered around a few classes that represent parallel collections. Parallel collections support a modest number of parallel operations which are composed to implement data-parallel computations where an entire pipeline, or even multiple pipelines, can be translated into a single Java program using the FlumeJava abstractions. To achieve good performance, FlumeJava internally implements parallel operations using *deferred* evaluation. The invocation of a parallel operation does not actually run the operation, but instead simply records the operation and its arguments in an internal execution plan graph structure. Once the execution plan for the whole computation has been constructed, FlumeJava optimizes the execution plan and then runs the optimized execution plan. When running the execution plan, FlumeJava chooses which strategy to use to implement each operation (e.g., local sequential loop versus remote parallel MapReduce) based in part on the size of the data being processed, places remote computations near the data on which they operate, and performs independent operations in parallel.

4.2. Pig Latin

Olston et al. [2008] have presented a language called *Pig Latin* that takes a *middle* position between expressing tasks using the high-level declarative querying model in the spirit of SQL and the low-level/procedural programming model using MapReduce. Pig Latin is implemented in the scope of the *Apache Pig* project[14] and is used by programmers at Yahoo! for developing data analysis tasks. Writing a Pig Latin program is similar to specifying a query execution plan (e.g., a data-flow graph). To experienced programmers, this method is more appealing than encoding their task as an SQL query and then coercing the system to choose the desired plan through optimizer hints. In general, automatic query optimization has its limits, especially with uncataloged data, prevalent user-defined functions, and parallel execution, which are all features of the data analysis tasks targeted by the MapReduce framework. Figure 9 shows an example SQL query and its equivalent Pig Latin program. Given a *URL* table with the structure

[14]http://incubator.apache.org/pig.

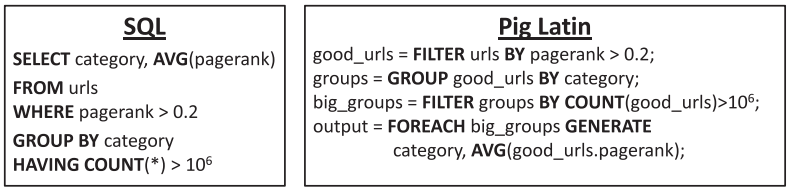


Fig. 9. An example SQL query and its equivalent pig latin program. Adaptation with permission of Gates et al. [2009].

(*url*, *category*, *pagerank*), the task of the SQL query is to find each large category and its average pagerank of high-pagerank urls (>0.2). A Pig Latin program is described as a sequence of steps where each step represents a single data transformation. This characteristic is appealing to many programmers. At the same time, the transformation steps are described using high-level primitives (e.g., filtering, grouping, aggregation) much like in SQL.

Pig Latin has several other features that are important for casual ad hoc data analysis tasks. These features include support for a flexible, fully nested data model, extensive support for user-defined functions, and the ability to operate over plain input files without any schema information [Gates 2011]. In particular, Pig Latin has a simple data model consisting of the following four types.

(1) *Atom*. An atom contains a simple atomic value such as a string or a number, such as "alice".
(2) *Tuple*. A tuple is a sequence of fields, each of which can be any of the data types, such as ("alice", "lakers").
(3) *Bag*. A bag is a collection of tuples with possible duplicates. The schema of the constituent tuples is flexible where not all tuples in a bag need to have the same number and type of fields
for example $\left\{\begin{array}{l}(\text{"alice"}, \text{"lakers"}) \\ (\text{"alice"}, (\text{"iPod"}, \text{"apple"}))\end{array}\right\}$
(4) *Map*. A map is a collection of data items, where each item has an associated key through which it can be looked up. As with bags, the schema of the constituent data items is flexible However, the keys are required to be data atoms, such as $\left\{\begin{array}{l}\text{"k1"} \rightarrow (\text{"alice"}, \text{"lakers"}) \\ \text{"k2"} \rightarrow \text{"20"}\end{array}\right\}$.

To accommodate specialized data processing tasks, Pig Latin has extensive support for User-Defined Functions (UDFs). The input and output of UDFs in Pig Latin follow its fully nested data model. Pig Latin is architected such that the parsing of the Pig Latin program and the logical plan construction is independent of the execution platform. Only the compilation of the logical plan into a physical plan depends on the specific execution platform chosen. Currently, Pig Latin programs are compiled into sequences of MapReduce jobs which are executed using the Hadoop MapReduce environment. In particular, a Pig Latin program goes through a series of transformation steps [Olston et al. 2008] before being executed as depicted in Figure 10. The parsing steps verify that the program is syntactically correct and that all referenced variables are defined. The output of the parser is a canonical logical plan with a one-to-one correspondence between Pig Latin statements and logical operators which are arranged in a Directed Acyclic Graph (DAG). The logical plan generated by the parser is passed through a logical optimizer. In this stage, logical optimizations such as projection pushdown are carried out. The optimized logical plan is then compiled into a series of MapReduce jobs which are then passed through another optimization phase. The

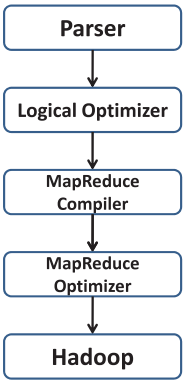


Fig. 10. Pig compilation and execution steps. Adaptation with permission of Olston et al. [2008].

DAG of optimized MapReduce jobs is then topologically sorted and jobs are submitted to Hadoop for execution.

4.3. Hive

The *Hive* project[15] is an open-source data warehousing solution which has been built by the Facebook Data Infrastructure Team on top of the Hadoop environment [Thusoo et al. 2009]. The main goal of this project is to bring the familiar relational database concepts (e.g., tables, columns, partitions) and a subset of SQL to the unstructured world of Hadoop while still maintaining the extensibility and flexibility that Hadoop provides. Thus, it supports all the major primitive types (e.g., integers, floats, strings) as well as complex types (e.g., maps, lists, structs). Hive supports queries expressed in an SQL-like declarative language, *HiveQL*[16], and therefore can be easily understood by anyone who is familiar with SQL. These queries are compiled into MapReduce jobs that are executed using Hadoop. In addition, HiveQL enables users to plug in custom MapReduce scripts into queries [Thusoo et al. 2010b]. For example, the canonical MapReduce word count example on a table of documents (Figure 1) can be expressed in HiveQL as depicted in Figure 11 where the *MAP* clause indicates how the input columns (*doctext*) can be transformed using a user program ("python wc_mapper.py") into output columns (*word* and *cnt*). The *REDUCE* clause specifies the user program to invoke ("python wc_reduce.py") on the output columns of the subquery.

HiveQL supports Data Definition Language (DDL) statements which can be used to create, drop, and alter tables in a database [Thusoo et al. 2010a]. It allows users to load data from external sources and insert query results into Hive tables via the load and insert Data Manipulation Language (DML) statements, respectively. However, HiveQL currently does not support the update and deletion of rows in existing tables (in particular, INSERT INTO, UPDATE, and DELETE statements) which allows the use of very simple mechanisms to deal with concurrent read and write operations without implementing complex locking protocols. The metastore component is the Hive's system catalog which stores metadata about the underlying table. This metadata is specified during table creation and reused every time the table is referenced in HiveQL. The metastore distinguishes Hive as a traditional warehousing solution when compared with similar data processing systems that are built on top of MapReduce-like architectures like Pig Latin [Olston et al. 2008].

[15]http://hadoop.apache.org/hive/.
[16]http://wiki.apache.org/hadoop/Hive/LanguageManual.

```
FROM (
    MAP doctext USING 'python wc_mapper.py' AS (word, cnt)
    FROM docs
    CLUSTER BY word
) a
REDUCE word, cnt USING 'python wc_reduce.py';
```

Fig. 11. An example HiveQl query. Adaptation with permission of Thusoo et al. [2009].

4.4. Tenzing

The *Tenzing* system [Chattopadhyay et al. 2011] has been presented by Google as an SQL query execution engine which is built on top of MapReduce and provides a comprehensive SQL92 implementation with some SQL99 extensions (e.g., ROLLUP() and CUBE() OLAP extensions). Tenzing also supports querying data in different formats such as: row stores (e.g., MySQL database), column stores, *Bigtable* (Google's built-in key-value store) [Chang et al. 2008], *GFS* (Google File System) [Ghemawat et al. 2003], text, and protocol buffers. In particular, the Tenzing system has four major components.

—*The distributed worker pool*. This represents the execution system which takes a query execution plan and executes the MapReduce jobs. The pool consists of master and worker nodes plus an overall gatekeeper called the master watcher. The workers manipulate the data for all the tables defined in the metadata layer.
—*The query server*. This serves as the gateway between client and pool. The query server parses the query, applies different optimization mechanisms, and sends the plan to the master for execution. In principle, the Tenzing optimizer applies some basic rule- and cost-based optimizations to create an optimal execution plan.
—*Client interfaces*. Tenzing has several client interfaces including a command-line client (CLI) and a Web UI. The CLI is a more powerful interface that supports complex scripting while the Web UI supports easier-to-use features such as query and table browsers tools. There is also an API to directly execute queries on the pool and a standalone binary which does not need any server-side components but rather can launch its own MapReduce jobs.
—*The metadata server*. This provides an API to store and fetch metadata such as table names and schemas and pointers to the underlying data.

A typical Tenzing query is submitted to the query server (through the Web UI, CLI, or API) which is responsible for parsing the query into an intermediate parse tree and fetching the required metadata from the metadata server. The query optimizer goes through the intermediate format, applies various optimizations, and generates a query execution plan that consists of one or more MapReduce jobs. For each MapReduce, the query server finds an available master using the master watcher and submits the query to it. At this stage, the execution is physically partitioned into multiple units of work where idle workers poll the masters for available work. The query server monitors the generated intermediate results, gathers them as they arrive, and streams the output back to the client. In order to increase throughput, decrease latency, and execute SQL operators more efficiently, Tenzing has enhanced the MapReduce implementation with the following main changes.

—*Streaming and In-memory Chaining*. The implementation of Tenzing does not serialize the intermediate results of MapReduce jobs to GFS. Instead, it streams the intermediate results between the Map and Reduce tasks using the network and uses GFS only for backup purposes. In addition, it uses a memory chaining mechanism

```
SELECT ...
FROM functionname(
   ON table-or-query
   [PARTITION BY expr, ...]
   [ORDER BY expr, ...]
   [clausename(arg, ...) ...]
   )
```

Fig. 12. Basic syntax of SQL/MR query function. Adaptation with permission of Friedman et al. [2009].

where the reducer and the mapper of the same intermediate results are colocated in the same process.

—*Sort Avoidance*. Certain operators such as hash join and hash aggregation require shuffling but not sorting. The MapReduce API was enhanced to automatically turn off sorting for these operations, when possible, so that the mapper feeds data to the reducer which automatically bypasses the intermediate sorting step. Tenzing also implements a block-based shuffle mechanism that combines many small rows into compressed blocks which are treated as one row in order to avoid reducer-side sorting and avoid some of the overhead associated with row serialization and deserialization in the underlying MapReduce framework code.

4.5. SQL/MapReduce

In general, a User-Defined Function (UDF) is a powerful database feature that allows users to customize database functionality. Friedman et al. [2009] introduced the SQL/MapReduce (SQL/MR) UDF framework which is designed to facilitate parallel computation of procedural functions across hundreds of servers working together as a single relational database. The framework is implemented as part of the *Aster Data Systems*[17] nCluster shared-nothing relational database. The framework leverage ideas from the MapReduce programming paradigm to provide users with a straightforward API through which they can implement a UDF in the language of their choice. Moreover, it allows maximum flexibility as the output schema of the UDF is specified by the function itself at query plan time. This means that a SQL/MR function is polymorphic as it can process arbitrary input because its behavior as well as output schema are dynamically determined by information available at query plan time. This also increases reusability as the same SQL/MR function can be used on inputs with many different schemas or with different user-specified parameters. In particular, SQL/MR allows the user to write custom-defined functions in any programming language and insert them into queries that leverage traditional SQL functionality. An SQL/MR function is defined in a manner that is similar to MapReduce's map and reduce functions.

The syntax for using an SQL/MR function is depicted in Figure 12 where the SQL/MR function invocation appears in the SQL *FROM* clause and consists of the function name followed by a set of clauses that are enclosed in parentheses. The *ON* clause specifies the input to the invocation of the SQL/MR function. It is important to note that the input schema to the SQL/MR function is specified implicitly at query plan time in the form of the output schema for the query used in the ON clause.

In practice, an SQL/MR function can be either a mapper (*Row* function) or a reducer (*Partition* function). The definitions of row and partition functions ensure that they can be executed in parallel in a scalable manner. In the *Row Function*, each row from the input table or query will be operated on by exactly one instance of the SQL/MR function. Semantically, each row is processed independently, allowing the execution engine to control parallelism. For each input row, the row function may emit zero or more rows.

[17]http://www.asterdata.com/.

In the *Partition Function*, each group of rows as defined by the *PARTITION BY* clause will be operated on by exactly one instance of the SQL/MR function. If the *ORDER BY* clause is provided, the rows within each partition are provided to the function instance in the specified sort order. Semantically, each partition is processed independently, allowing parallelization by the execution engine at the level of a partition. For each input partition, the SQL/MR partition function may output zero or more rows.

4.6. HadoopDB

There has been a long debate on the comparison between MapReduce framework and parallel database systems[18] [Stonebraker et al. 2010]. Pavlo et al. [2009] have conducted a large-scale comparison between the Hadoop implementation of MapReduce framework and parallel SQL database management systems in terms of performance and development complexity. The results of this comparison have shown that parallel database systems display a significant performance advantage over MapReduce in executing a variety of data-intensive analysis tasks. On the other hand, the Hadoop implementation is very much easier and more straightforward to set up and use in comparison to that of the parallel database systems. MapReduce has also been shown to have superior performance in minimizing the amount of work that is lost when a hardware failure occurs. In addition, MapReduce (with its open-source implementations) represents a very in-expensive solution in comparison to the very financially expensive parallel DBMS solutions (the price of an installation of a parallel DBMS cluster usually consists of 7 figures of U.S. dollars) [Stonebraker et al. 2010].

The *HadoopDB* project[19] is a hybrid system that tries to combine the scalability advantages of MapReduce with the performance and efficiency advantages of parallel databases [Abouzeid et al. 2009]. The basic idea behind HadoopDB is to connect multiple single-node database systems (PostgreSQL) using Hadoop as the task coordinator and network communication layer. Queries are expressed in SQL but their execution is parallelized across nodes using the MapReduce framework, however, as much of the single-node query work as possible is pushed inside of the corresponding node databases. Thus, HadoopDB tries to achieve fault tolerance and the ability to operate in heterogeneous environments by inheriting the scheduling and job tracking implementation from Hadoop. In parallel, it tries to achieve the performance of parallel databases by doing most of the query processing inside the database engine. Figure 13 illustrates the architecture of HadoopDB which consists of two layers: (1) a data storage layer or the Hadoop Distributed File System[20] (HDFS); (2) a data processing layer or the MapReduce framework. In this architecture, HDFS is a block-structured file system managed by a central *NameNode*. Individual files are broken into blocks of a fixed size and distributed across multiple *DataNodes* in the cluster. The NameNode maintains metadata about the size and location of blocks and their replicas. The MapReduce framework follows a simple master-slave architecture. The master is a single *JobTracker* and the slaves or worker nodes are *TaskTrackers*. The JobTracker handles the runtime scheduling of MapReduce jobs and maintains information on each TaskTracker's load and available resources. The *Database Connector* is the interface between independent database systems residing on nodes in the cluster and TaskTrackers. The Connector connects to the database, executes the SQL query, and returns results as key-value pairs. The *Catalog* component maintains metadata about the databases, their location, replica locations, and data partitioning properties. The *Data Loader* component is responsible for globally repartitioning data on a given

[18]http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.
[19]http://db.cs.yale.edu/hadoopdb/hadoopdb.html.
[20]http://hadoop.apache.org/hdfs/.

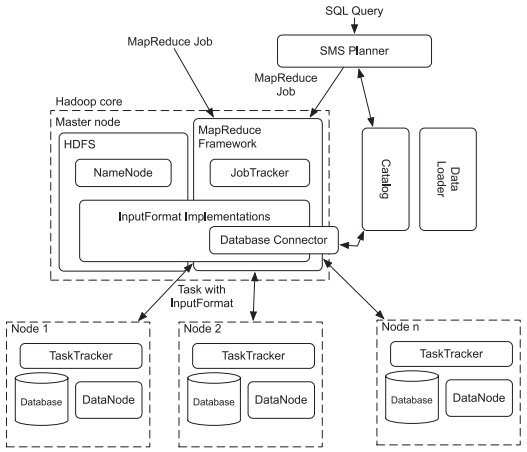


Fig. 13. The architecture of HadoopDB. Adaptation with permission of Abouzeid et al. [2009].

partition key upon loading and breaking apart single-node data into multiple smaller partitions or chunks. The *SMS planner* extends the HiveQL translator [Thusoo et al. 2009] (Section 4.3) and transforms SQL into MapReduce jobs that connect to tables stored as files in HDFS. Abouzeid et al. [2010] have demonstrated HadoopDB in action running the following two different application types: (1) a semantic Web application that provides biological data analysis of protein sequences; (2) a classical business data warehouse.

4.7. Jaql

Jaql[21] is a query language which is designed for Javascript Object Notation (JSON)[22], a data format that has become popular because of its simplicity and modeling flexibility. JSON is a simple, yet flexible way to represent data that ranges from flat, relational data to semistructured, XML data. Jaql is primarily used to analyze large-scale semistructured data. It is a functional, declarative query language which rewrites high-level queries when appropriate into a low-level query consisting of Map-Reduce jobs that are evaluated using the Apache Hadoop project. Core features include user extensibility and parallelism. Jaql consists of a scripting language and compiler, as well as a runtime component [Beyer et al. 2011]. It is able to process data with no schema or only with a partial schema. However, Jaql can also exploit rigid schema information when it is available, for both type checking and improved performance. Jaql uses a very simple data model, wherein a *JDM value* is either an atom, an array, or a record. Most common atomic types are supported, including strings, numbers, nulls, and dates. Arrays and records are compound types that can be arbitrarily nested. In more detail, an array is an ordered collection of values and can be used to model data structures such as vectors, lists, sets, or bags. A record is an unordered collection of name-value pairs and can model structs, dictionaries, and maps. Despite its simplicity, JDM is very flexible. It allows Jaql to operate with a variety of different data representations for both input and output, including delimited text files, JSON files, binary files, Hadoop's SequenceFiles, relational databases, key-value stores, or XML documents. Functions are first-class values in Jaql. They can be assigned to a variable and are high order in that they can be passed as parameters or used as a return value. Functions are the

[21]http://code.google.com/p/jaql/.
[22]http://www.json.org/.

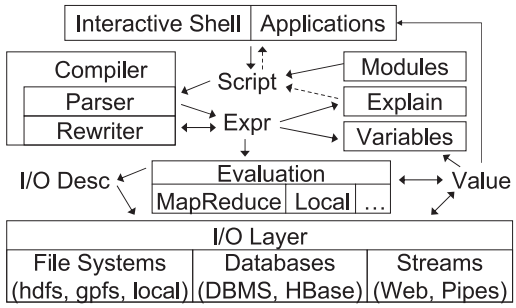


Fig. 14. Jaql system architecture. Adaptation with permission of Beyer et al. [2011].

key ingredient for reusability as any Jaql expression can be encapsulated in a function, and a function can be parameterized in powerful ways.

At a high level, the Jaql architecture depicted in Figure 14 is similar to most database systems. Scripts are passed into the system from the interpreter or an application, compiled by the parser and rewrite engine, and either explained or evaluated over data from the I/O layer. The storage layer is similar to a federated database. It provides an API to access data of different systems including local or distributed file systems (e.g., Hadoop's HDFS), database systems (e.g., DB2, Netezza, HBase), or from streamed sources like the Web. Unlike federated databases, however, most of the accessed data is stored within the same cluster and the I/O API describes data partitioning, which enables parallelism with data affinity during evaluation. Jaql derives much of this flexibility from Hadoop's I/O API. It reads and writes many common file formats (e.g., delimited files, JSON text, Hadoop Sequence files). Custom adapters are easily written to map a dataset to or from Jaql's data model. The input can even simply be values constructed in the script itself. The Jaql interpreter evaluates the script locally on the computer that compiled the script, but spawns interpreters on remote nodes using MapReduce. The Jaql compiler automatically detects parallelization opportunities in a Jaql script and translates it to a set of MapReduce jobs.

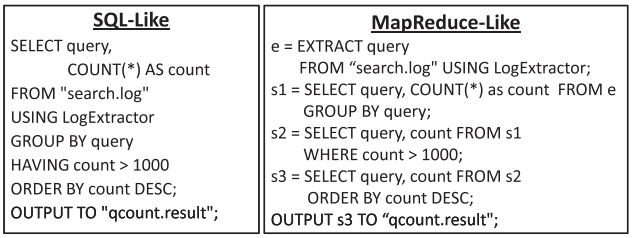
5. RELATED LARGE-SCALE DATA PROCESSING SYSTEMS

In this section, we give an overview of several large-scale data processing systems that resemble some of the ideas of the MapReduce framework for different purposes and application scenarios. It must be noted, however, the design architectures and the implementations of these systems do not follow the architecture of the MapReduce framework and thus, they do not utilize nor are they related to the infrastructure of the framework's open-source implementations such as Hadoop.

5.1. SCOPE

SCOPE (Structured Computations Optimized for Parallel Execution) is a scripting language which is targeted for large-scale data analysis and is used daily for a variety of data analysis and data mining applications inside Microsoft [Chaiken et al. 2008]. SCOPE is a declarative language. It allows users to focus on the data transformations required to solve the problem at hand and hides the complexity of the underlying platform and implementation details. The SCOPE compiler and optimizer are responsible for generating an efficient execution plan and the runtime for executing the plan with minimal overhead.

Like SQL, data is modeled as sets of rows composed of typed columns. SCOPE is highly extensible. Users can easily define their own functions and implement their own versions of operators: extractors (parsing and constructing rows from a file), processors



SQL-Like

```
SELECT query,
       COUNT(*) AS count
FROM "search.log"
USING LogExtractor
GROUP BY query
HAVING count > 1000
ORDER BY count DESC;
OUTPUT TO "qcount.result";
```

MapReduce-Like

```
e = EXTRACT query
    FROM "search.log" USING LogExtractor;
s1 = SELECT query, COUNT(*) as count  FROM e
     GROUP BY query;
s2 = SELECT query, count FROM s1
     WHERE count > 1000;
s3 = SELECT query, count FROM s2
      ORDER BY count DESC;
OUTPUT s3 TO "qcount.result";
```

Fig. 15. Two equivalent SCOPE scripts in SQL-like style and MapReduce-like style. Adaptation with permission of Chaiken et al. [2008].

(row-wise processing), reducers (group-wise processing), and combiners (combining rows from two inputs). This flexibility greatly extends the scope of the language and allows users to solve problems that cannot be easily expressed in traditional SQL. SCOPE provides a functionality which is similar to that of SQL views. This feature enhances modularity and code reusability. It is also used to restrict access to sensitive data. SCOPE supports writing a program using traditional SQL expressions or as a series of simple data transformations. Figure 15 illustrates two equivalent scripts in the two different styles (SQL-like and MapReduce-like) to find from the search log the popular queries that have been requested at least 1000 times. In the MapReduce-like style, the *EXTRACT* command extracts all query strings from the log file. The first *SELECT* command counts the number of occurrences of each query string. The second *SELECT* command retains only rows with a count greater than 1000. The third *SELECT* command sorts the rows on count. Finally, the *OUTPUT* command writes the result to the file "*qcount.result*".

Microsoft has developed a distributed computing platform, called *Cosmos*, for storing and analyzing massive datasets. Cosmos is designed to run on large clusters consisting of thousands of commodity servers. Figure 16 shows the main components of the Cosmos platform which is described as follows.

—*Cosmos Storage*. A distributed storage subsystem is designed to reliably and efficiently store extremely large sequential files.
—*Cosmos Execution Environment*. An environment is made for deploying, executing, and debugging distributed applications.
—*SCOPE*. A high-level scripting language is provided for writing data analysis jobs. The SCOPE compiler and optimizer translate scripts to efficient parallel execution plans.

The Cosmos storage system is an append-only file system that reliably stores petabytes of data. The system is optimized for large sequential I/O. All writes are append only and concurrent writers are serialized by the system. Data is distributed and replicated for fault tolerance and compressed to save storage and increase I/O throughput. In Cosmos, an application is modeled as a data-flow graph: a Directed Acyclic Graph (DAG) with vertices representing processes and edges representing data flows. The runtime component of the execution engine is called the Job Manager which represents the central and coordinating process for all processing vertices within an application.

The SCOPE scripting language resembles SQL but with C# expressions. Thus, it reduces the learning curve for users and eases the porting of existing SQL scripts into SCOPE. Moreover, SCOPE expressions can use C# libraries where custom C# classes can compute functions of scalar values, or manipulate whole rowsets. A SCOPE script consists of a sequence of commands which are data transformation operators that take one or more rowsets as input, perform some operation on the data, and output a rowset.

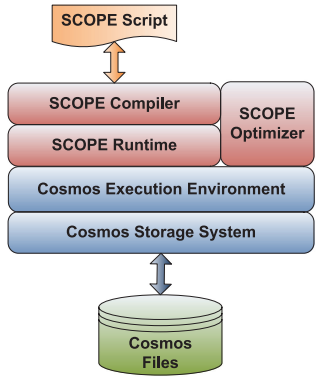


Fig. 16. SCOPE/Cosmos execution platform. Adaptation with permission of Chaiken et al. [2008].

Every rowset has a well-defined schema to which all its rows must adhere. The SCOPE compiler parses the script, checks the syntax, and resolves names. The result of the compilation is an internal parse tree which is then translated to a physical execution plan. A physical execution plan is a specification of a Cosmos job which describes a data-flow DAG where each vertex is a program and each edge represents a data channel. The translation into an execution plan is performed by traversing the parse tree in a bottom-up manner. For each operator, SCOPE has an associated default implementation rule. Many of the traditional optimization rules from database systems are clearly also applicable in this new context, for example, removing unnecessary columns, pushing down selection predicates, and preaggregating when possible. However, the highly distributed execution environment offers new opportunities and challenges, making it necessary to explicitly consider the effects of large-scale parallelism during optimization. For example, choosing the right partition scheme and deciding when to partition are crucial for finding an optimal plan. It is also important to correctly reason about partitioning, grouping, and sorting properties and their interaction, to avoid unnecessary computations [Zhou et al. 2010].

5.2. Dryad/DryadLinq

Dryad is a general-purpose distributed execution engine introduced by Microsoft for coarse-grain data-parallel applications [Isard et al. 2007]. A Dryad application combines computational *vertices* with communication *channels* to form a data-flow graph. Dryad runs the application by executing the vertices of this graph on a set of available computers, communicating as appropriate through files, TCP pipes, and shared-memory FIFOs. The Dryad system allows the developer fine control over the communication graph as well as the subroutines that live at its vertices. A Dryad application developer can specify an arbitrary directed acyclic graph to describe the application's communication patterns and express the data transport mechanisms (files, TCP pipes, and shared-memory FIFOs) between the computation vertices. This direct specification of the graph gives the developer greater flexibility to easily compose basic common operations, leading to a distributed analogue of *piping* together traditional Unix utilities such as grep, sort, and head.

Dryad is notable for allowing graph vertices (and computations in general) to use an arbitrary number of inputs and outputs while MapReduce restricts all computations to take a single input set and generate a single output set. The overall structure of a Dryad job is determined by its communication flow. A job is a directed acyclic graph where each vertex is a program and edges represent data channels. It is a

logical computation graph that is automatically mapped onto physical resources by the runtime. At runtime each channel is used to transport a finite sequence of structured items. A Dryad job is coordinated by a process called the *job manager* that runs either within the cluster or on a user's workstation with network access to the cluster. The job manager contains the application-specific code to construct the job's communication graph along with library code to schedule the work across the available resources. All data is sent directly between vertices and thus the job manager is only responsible for control decisions and is not a bottleneck for any data transfers. Therefore, much of the simplicity of the Dryad scheduler and fault-tolerance model come from the assumption that vertices are deterministic.

Dryad has its own high-level language called *DryadLINQ* [Yu et al. 2008]. It generalizes execution environments such as SQL and MapReduce in two ways: (1) adopting an expressive data model of strongly typed .NET objects; (2) supporting general-purpose imperative and declarative operations on datasets within a traditional high-level programming language. DryadLINQ[23] exploits LINQ (Language INtegrated Query[24], a set of .NET constructs for programming with datasets) to provide a powerful hybrid of declarative and imperative programming. The system is designed to provide flexible and efficient distributed computation in any LINQ-enabled programming language including C#, VB, and F#[25]. Objects in DryadLINQ datasets can be of any .NET type, making it easy to compute with data such as image patches, vectors, and matrices. In practice, a DryadLINQ program is a sequential program composed of LINQ expressions that perform arbitrary side-effect-free transformations on datasets and can be written and debugged using standard .NET development tools. The DryadLINQ system automatically translates the data-parallel portions of the program into a distributed execution plan which is then passed to the Dryad execution platform. A commercial implementation of Dryad and DryadLINQ was released in 2011 under the name *LINQ to HPC*[26].

5.3. Spark

The *Spark* system [Zaharia et al. 2012, 2010b] has been proposed to support those applications which need to reuse a working set of data across multiple parallel operations (e.g., iterative machine learning algorithms and interactive data analytic) while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called *Resilient Distributed Datasets* (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Therefore, users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations. RDDs do not need to be materialized at all times. RDDs achieve fault tolerance through a notion of *lineage*. In particular, each RDD object contains a pointer to its parent and information about how the parent was transformed. Hence, if a partition of an RDD is lost, the RDD has sufficient information about how it was derived from other RDDs to be able to rebuild just that partition.

Spark is implemented in the Scala programming language[27] [Odersky et al. 2011]. It is built on top of *Mesos* [Hindman et al. 2009], a cluster operating system that lets multiple parallel frameworks share a cluster in a fine-grained manner and provides an API for applications to launch tasks on a cluster. It provides isolation and efficient

[23]http://research.microsoft.com/en-us/projects/dryadlinq/.
[24]http://msdn.microsoft.com/en-us/netframework/aa904594.aspx.
[25]http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/.
[26]http://msdn.microsoft.com/en-us/library/hh378101.aspx.
[27]http://www.scala-lang.org/.

resource sharing across frameworks running on the same cluster while giving each framework freedom to implement its own programming model and fully control the execution of its jobs. Mesos uses two main abstractions: *tasks* and *slots*. A task represents a unit of work. A slot represents a computing resource in which a framework may run a task, such as a core and some associated memory on a multicore machine. It employs the two-level scheduling mechanism. At the first level, Mesos allocates slots between frameworks using fair sharing. At the second level, each framework is responsible for dividing its work into tasks, selecting which tasks to run in each slot. This lets frameworks perform application-specific optimizations. For example, Spark's scheduler tries to send each task to one of its preferred locations using a technique called *delay scheduling* [Zaharia et al. 2010a].

To use Spark, developers need to write a driver program that implements the high-level control flow of their application and launches various operations in parallel. Spark provides two main abstractions for parallel programming: resilient distributed datasets and parallel operations on these datasets (invoked by passing a function to apply on a dataset). In particular, each RDD is represented by a Scala object which can be constructed in different ways.

—It can be constructed from a file in a shared file system (e.g., HDFS).
—It can be constructed by parallelizing a Scala collection (e.g., an array) in the driver program which means dividing it into a number of slices that will be sent to multiple nodes.
—It can be constructed by transforming an existing RDD. A dataset with elements of type A can be transformed into a dataset with elements of type B using an operation called $flatMap$.
—It can be constructed by changing the persistence of an existing RDD. A user can alter the persistence of an RDD through two actions:
  —The cache action leaves the dataset lazy but hints that it should be kept in memory after the first time it is computed because it will be reused.
  —The save action evaluates the dataset and writes it to a distributed file system such as HDFS. The saved version is used in future operations on it.

Different parallel operations can be performed on RDDs.

—The *reduce* operation combines dataset elements using an associative function to produce a result at the driver program.
—The *collect* operation sends all elements of the dataset to the driver program.
—The *foreach* passes each element through a user-provided function.

Spark does not currently support a grouped reduce operation as in MapReduce. The results of reduce operations are only collected at the driver process.

5.4. Nephle/Pact

The *Nephele/PACT* system [Battré et al. 2010; Alexandrov et al. 2010] has been presented as a parallel data processor centered around a programming model of so-called *Parallelization Contracts* (PACTs) and the scalable parallel execution engine *Nephele*. The PACT programming model is a generalization of map/reduce as it is based on a key/value data model and the concept of *Parallelization Contracts* (PACTs). A PACT consists of exactly one second-order function which is called *Input Contract* and an optional *Output Contract*. An Input Contract takes a first-order function with task-specific user code and one or more datasets as input parameters. The Input Contract invokes its associated first-order function with independent subsets of its input data in a data-parallel fashion. In this context, the two functions of *map* and *reduce* are

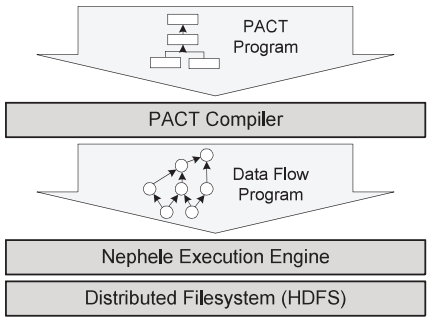


Fig. 17. The Nephele/PACT system architecture. Adaptation with permission of Alexandrov et al. [2010].

just examples of the Input Contracts. Other example of Input Contracts include the following.

—The *Cross* contract operates on multiple inputs and builds a distributed Cartesian product over its input sets.
—The *CoGroup* contract partitions each of its multiple inputs along the key. Independent subsets are built by combining equal keys of all inputs.
—The *Match* contract operates on multiple inputs. It matches key/value pairs from all input datasets with the same key (equivalent to the inner join operation).

An Output Contract is an optional component of a PACT and gives guarantees about the data that is generated by the assigned user function. The set of Output Contracts includes the following.

—The *Same-Key* contract is where each key/value pair that is generated by the function has the same key as the key/value pair(s) from which it was generated. This means the function will preserve any partitioning and order property on the keys.
—The *Super-Key* is where each key/value pair that is generated by the function has a superkey of the key/value pair(s) from which it was generated. This means the function will preserve the partitioning and the partial order on the keys.
—The *Unique-Key* is where each key/value pair that is produced has a unique key. The key must be unique across all parallel instances. Any produced data is therefore partitioned and grouped by the key.
—The *Partitioned-by-Key* is where key/value pairs are partitioned by key. This contract has similar implications as the Super-Key contract, specifically that a partitioning by the keys is given, but there is no order inside the partitions.

Figure 17 illustrates the system architecture of Nephele/PACT where a PACT program is submitted to the PACT Compiler which translates the program into a dataflow execution plan which is then handed to the Nephele system for parallel execution. Hadoop distributed file system (HDFS) is used for storing both the input and the output data.

Due to the declarative character of the PACT programming model, the PACT compiler can apply different optimization mechanisms and select from several execution plans with varying costs for a single PACT program. For example, the *Match* contract can be satisfied using either a repartition strategy which partitions all inputs by keys or a broadcast strategy that fully replicates one input to every partition of the other input. Choosing the right strategy can dramatically reduce network traffic and execution time. Therefore, the PACT compiler applies standard SQL optimization techniques [Selinger et al. 1979] where it exploits information provided by the Output Contracts and applies

different cost-based optimization techniques. In particular, the optimizer generates a set of candidate execution plans in a bottom-up fashion (starting from the data sources) where the more expensive plans are pruned using a set of *interesting properties* for the operators. These properties are also used to spare plans from pruning that comes with an additional property that may amortize their cost overhead later.

5.5. Boom Analytics

The *BOOM Analytics* (Berkeley Orders of Magnitude) [Alvaro et al. 2010] is an API-compliant reimplementation of the HDFS distributed file system (*BOOM-FS*) and the Hadoop MapReduce engine (*BOOM-MR*). The implementation of BOOM Analytics uses the *Overlog* logic language [Loo et al. 2005] which has been originally presented as an event-driven language and evolved a semantics more carefully grounded in *Datalog*, the standard deductive query language from database theory [Ullman 1990]. In general, the Datalog language is defined over relational tables as a purely logical query language that makes no changes to the stored tables. Overlog extends Datalog in three main features [Condie et al. 2008].

(1) It adds notation to specify the location of data.
(2) It provides some SQL-style extensions such as primary keys and aggregation.
(3) It defines a model for processing and generating changes to tables.

When Overlog tuples arrive at a node, either through rule evaluation or external events, they are handled in an atomic local Datalog *timestep*. Within a timestep, each node sees only locally stored tuples. Communication between Datalog and the rest of the system (Java code, networks, and clocks) is modeled using events corresponding to insertions or deletions of tuples in Datalog tables. BOOM Analytics uses a Java-based Overlog runtime called *JOL* which compiles Overlog programs into pipelined data-flow graphs of operators. In particular, JOL provides metaprogramming support where each Overlog program is compiled into a representation that is captured in rows of tables. In BOOM Analytics, *everything* is data. This includes traditional persistent information like file system metadata, runtime state like TaskTracker status, summary statistics like those used by the JobTracker's scheduling policy, communication messages, system events, and execution state of the system.

The BOOM-FS component represents the file system metadata as a collection of relations (*file*, *fqpath*, *fchunk*, *datanode*, *hbchunk*) where file system operations are implemented by writing queries over these tables. The *file* relation contains a row for each file or directory stored in BOOM-FS. The set of chunks in a file is identified by the corresponding rows in the *fchunk* relation. The *datanode* and *hbchunk* relations contain the set of live DataNodes and the chunks stored by each DataNode, respectively. The NameNode updates these relations as new heartbeats arrive. If the NameNode does not receive a heartbeat from a DataNode within a configurable amount of time, it assumes that the DataNode has failed and removes the corresponding rows from these tables. Since a file system is naturally hierarchical, the file system queries that need to traverse it are recursive. Therefore, the parent-child relationship of files is used to compute the transitive closure of each file and store its fully qualified path in the *fqpath* relation. Because path information is accessed frequently, the *fqpath* relation is configured to be cached after it is computed. Overlog will automatically update *fqpath* when a file is changed, using standard relational view maintenance logic [Ullman 1990]. BOOM-FS also defines several views to compute derived file system metadata such as the total size of each file and the contents of each directory. The materialization of each view can be changed via simple Overlog table definition statements without altering the semantics of the program. In general, HDFS uses three different communication protocols: the *metadata protocol* which is used by clients and NameNodes to exchange file metadata,

the *heartbeat protocol* which is used by the DataNodes to notify the NameNode about chunk locations and DataNode liveness, and the *data protocol* which is used by the clients and DataNodes to exchange chunks. BOOM-FS reimplemented these three protocols using a set of Overlog rules. BOOM-FS also achieves the high availability failover mechanism by using Overlog to implement the *hot standby* NameNodes feature using Lamport's Paxos algorithm [Lamport 1998].

BOOM-MR reimplements the MapReduce framework by replacing Hadoop's core scheduling logic with Overlog. The JobTracker tracks the ongoing status of the system and transient state in the form of messages sent and received by the JobTracker by capturing this information in four Overlog tables: *job*, *task*, *taskAttempt*, and *taskTracker*. The *job* relation contains a single row for each job submitted to the JobTracker. The *task* relation identifies each task within a job. The attributes of this relation identify the task type (map or reduce), the input partition (a chunk for map tasks, a bucket for reduce tasks), and the current running status. The *taskAttempt* relation maintains the state of each task attempt (a task may be attempted more than once due to speculation or if the initial execution attempt failed). The *taskTracker* relation identifies each TaskTracker in the cluster with a unique name. Overlog rules are used to update the JobTracker's tables by converting inbound messages into tuples of the four Overlog tables. Scheduling decisions are encoded in the *taskAttempt* table which assigns tasks to *TaskTrackers*. A scheduling policy is simply a set of rules that join against the *taskTracker* relation to find *TaskTrackers* with unassigned slots and schedules tasks by inserting tuples into *taskAttempt*. This architecture allows new scheduling policies to be defined easily.

5.6. Hyracks/ASTERIX

Hyracks is presented as a partitioned-parallel data-flow execution platform that runs on shared-nothing clusters of computers [Borkar et al. 2011]. Large collections of data items are stored as local partitions distributed across the nodes of the cluster. A Hyracks job is submitted by a client and processes one or more collections of data to produce one or more output collections (partitions). Hyracks provides a programming model and an accompanying infrastructure to efficiently divide computations on large data collections (spanning multiple machines) into computations that work on each partition of the data separately. Every Hyracks cluster is managed by a *Cluster Controller* process. The Cluster Controller accepts job execution requests from clients, plans their evaluation strategies, and then schedules the jobs' tasks to run on selected machines in the cluster. In addition, it is responsible for monitoring the state of the cluster to keep track of the resource loads at the various worker machines. The Cluster Controller is also responsible for replanning and reexecuting some or all of the tasks of a job in the event of a failure. On the task execution side, each worker machine that participates in a Hyracks cluster runs a *Node Controller* process. The Node Controller accepts task execution requests from the Cluster Controller and also reports on its health via a heartbeat mechanism.

In principle, Hyracks has been designed with the goal of being a runtime platform where users can create their jobs and also to serve as an efficient target for the compilers of higher-level programming languages such as Pig, Hive, or Jaql. The *ASTERIX* project [Behm et al. 2011; Borkar et al. 2012a] uses this feature with the aim of building a scalable information management system that supports the storage, querying, and analysis of large collections of semistructured nested data objects. The ASTERIX data storage and query processing are based on its own semistructured model called the *ASTERIX Data Model* (ADM). Each individual ADM data instance is typed and self-describing. All data instances live in *datasets* (the ASTERIX analogy to tables) and

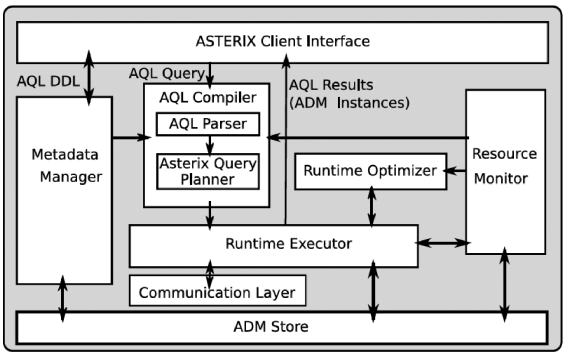


Fig. 18. The ASTERIX system architecture. Adaptation with permission of Behm et al. [2011].

datasets can be indexed, partitioned, and possibly replicated to achieve the scalability and availability goals. External datasets which reside in files that are not under ASTERIX control are also supported. An instance of the ASTERIX data model can either be a primitive type (e.g., integer, string, time) or a derived type, which may include the following.

—*Enum*. This is an enumeration type, whose domain is defined by listing the sequence of possible values.
—*Record*. This is a set of fields where each field is described by its name and type. A record can be either an open record where it contains fields that are not part of the type definition or a closed record which cannot.
—*Ordered list*. This is a sequence of values for which the order is determined by creation or insertion.
—*Unordered list*. This is an unordered sequence of values which is similar to bags in SQL.
—*Union*. This is describes a choice between a finite set of types.

A dataset is a target for AQL queries and updates and is also the attachment point for indexes. A collection of datasets related to an application are grouped into a namespace called a *dataverse* which is analogous to a database in the relational world. In particular, data is accessed and manipulated through the use of the *ASTERIX Query Language* (AQL) which is designed to cleanly match and handle the data structuring constructs of ADM. It borrows from *XQuery* and *Jaql* their programmer-friendly declarative syntax that describes bulk operations such as iteration, filtering, and sorting. Therefore, AQL is comparable to those languages in terms of expressive power. The major difference with respect to XQuery is AQL's focus on data-centric use cases at the expense of built-in support for mixed content for document-centric use cases. In ASTERIX, there is no notion of document order or node identity for data instances. Differences between AQL and Jaql stem from the usage of the languages. While ASTERIX data is stored in and managed by the ASTERIX system, Jaql runs against data that are stored externally in Hadoop files or in the local file system. Figure 18 presents an overview of the ASTERIX system architecture. AQL requests are compiled into jobs for an ASTERIX execution layer, Hyracks. ASTERIX concerns itself with the data details of AQL and ADM, turning AQL requests into Hyracks jobs while Hyracks determines and oversees the utilization of parallelism based on information and constraints associated with the resulting jobs' operators as well as on the runtime state of the cluster.

6. CONCLUSIONS

The database community has been always focusing on dealing with the challenges of *big data* management, although the meaning of "big" has been evolving continuously to represent different scales over the time [Borkar et al. 2012b]. According to IBM, we are currently creating 2.5 quintillion bytes of data, everyday. This data comes from many different sources and in different formats including digital pictures, videos, posts to social media sites, intelligent sensors, purchase transaction records, and cell phone GPS signals. This is a new scale of *big data* which is attracting a huge interest from both the industrial and research communities with the aim of creating the best means to process and analyze this data in order to make the best use of it. In the last decade, the MapReduce framework has emerged as a popular mechanism to harness the power of large clusters of computers. It allows programmers to think in a *data-centric* fashion where they can focus on applying transformations to sets of data records while the details of distributed execution and fault tolerance are transparently managed by the MapReduce framework.

In this article, we presented a survey of the MapReduce family of approaches for developing scalable data processing systems and solutions. In general we noticed that the MapReduce framework, and its open-source implementation of Hadoop, are now considered to be sufficiently mature such that they are widely used for developing many solutions by academia and industry in different application domains. We believe that it is unlikely that MapReduce will substitute database systems even for data warehousing applications. We expect that they will always coexist and complement each other in different scenarios. We are also convinced that there is still room for further optimization and advancement in different directions on the spectrum of the MapReduce framework that is required to bring forward the vision of providing large-scale data analysis as a commodity for novice end-users. For example, energy efficiency in the MapReduce is an important problem which has not attracted enough attention from the research community, yet. The traditional challenge of debugging large-scale computations on a distributed system has not been considered as a research priority by the MapReduce research community. Related with the issue of the power of expressiveness of the programming model, we feel that this is an area that requires more investigation. We also noticed that the oversimplicity of the MapReduce programming model has raised some key challenges on dealing with complex data models (e.g., nested models, XML, and hierarchical model, RDF, and graphs) efficiently. This limitation has called for the need of next-generation big data architectures and systems that can provide the required scale and performance attributes for these domains. For example, Google has created the *Dremel* system [Melnik et al. 2010], commercialized under the name of *BigQuery*[28], to support interactive analysis of nested data. Google has also presented the *Pregel* system [Malewicz et al. 2010], open-sourced by *Apache Giraph* and *Apache Hama* projects, that uses a BSP-based programming model for efficient and scalable processing of massive graphs on distributed clusters of commodity machines. Recently, *Twitter* has announced the release of the *Storm*[29] system as a distributed and fault-tolerant platform for implementing continuous and real-time processing applications of streamed data. We believe that more of these domain-specific systems will be introduced in the future to form the new generation of big data systems. Defining the right and most convenient programming abstractions and declarative interfaces of these domain-specific big data systems is another important research direction that will need to be deeply investigated.

[28]https://developers.google.com/bigquery/.
[29]https://github.com/nathanmarz/storm/.

ELECTRONIC APPENDIX

The electronic appendix to this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

The authors would like to thank the VLDB Endowment for granting the permission to use the adopted figures of its referenced publications.

REFERENCES

ABADI, D. J., MARCUS, A., MADDEN, S., AND HOLLENBACH, K. 2009. SW-store: A vertically partitioned dbms for semantic web data management. *VLDB J. 18*, 2, 385–406.

ABOUZEID, A., BAJDA-PAWLIKOWSKI, K., ABADI, D., RASIN, A., AND SILBERSCHATZ, A. 2009. HadoopDB: An architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow. 2*, 1, 922–933.

ABOUZEID, A., BAJDA-PAWLIKOWSKI, K., HUANG, J., ABADI, D., AND SILBERSCHATZ, A. 2010. HadoopDB in action: Building real world applications. In *Proceedings of the 36th ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*.

AFRATI, F. AND ULLMAN, J. 2010. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT'10)*. 99–110.

AFRATI, F. N., SARMA, A. D., MENESTRINA, D., PARAMESWARAN, A. G., AND ULLMAN, J. D. 2012. Fuzzy joins using mapreduce. In *Proceedings of the 28th IEEE International Conference on Data Engineering (ICDE'12)*. 498–509.

AFRATI, F. N. AND ULLMAN, J. D. 2011. Optimizing multiway joins in a map-reduce environment. *IEEE Trans. Knowl. Data Engin. 23*, 9, 1282–1298.

ALEXANDROV, A., BATTRE, D., EWEN, S., HEIMEL, M., HUESKE, F., KAO, O., MARKL, V., NIJKAMP, E., AND WARNEKE, D. 2010. Massively parallel data analysis with pacts on nephele. *Proc. VLDB Endow. 3*, 2, 1625–1628.

ALVARO, P., CONDIE, T., CONWAY, N., ELMELEEGY, K., HELLERSTEIN, J. M., AND SEARS, R. 2010. Boom analytics: Exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European Conference on Computer Systems* (*EuroSys'10*). 223–236.

ARMBRUST, M., FOX, A., REAN, G., JOSEPH, A., KATZ, R., KONWINSKI, A., GUNHO, L., DAVID, P., RABKIN, A., STOICA, I., AND ZAHARIA, M. 2009. Above the clouds: A berkeley view of cloud computing. http://www.cs.columbia.edu/~roxana/teaching/COMS-E6998-7-Fall-2011/papers/armbrust-tr09.pdf.

BABU, S. 2010. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. 137–142.

BALMIN, A., KALDEWEY, T., AND TATA, S. 2012. Clydesdale: Structured data processing on hadoop. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*. 705–708.

BATTRE, D., EWEN, S., HUESKE, F., KAO, O., MARKL, V., AND WARNEKE, D. 2010. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'10)*. 119–130.

BEHM, A., BORKAR, V. R., CAREY, M. J., GROVER, R., LI, C., ONOSE, N., VERNICA, R., DEUTSCH, A., PAPAKONSTANTINOU, Y., AND TSOTRAS, V. J. 2011. ASTERIX: Towards a scalable, semistructured data platform for evolving-world models. *Distrib. Parallel Databases 29*, 3, 185–216.

BELL, G., GRAY, J., AND SZALAY, A. S. 2006. Petascale computational systems. *IEEE Comput. 39*, 1, 110–112.

BEYER, K. S., ERCEGOVAC, V., GEMULLA, R., BALMIN, A., ELTABAKH, M. Y., KANNE, C.-C., OZCAN, F., AND SHEKITA, E. J. 2011. Jaql: A scripting language for large scale semistructured data analysis. *Proc. VLDB Endow. 4*, 12, 1272–1283.

BHATOTIA, P., WIEDER, A., RODRIGUES, R., ACAR, U. A., AND PASQUINI, R. 2011. Incoop: MapReduce for Incremental computations. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'11)*.

BLANAS, S., PATEL, J., ERCEGOVAC, V., RAO, J., SHEKITA, E., AND TIAN, Y. 2010. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. 975–986.

BOAG, S., CHAMBERLIN, D., FERNANDEZ, M. F., FLORESCU, D., ROBIE, J., AND SIMEON, J. 2010. XQuery 1.0: An xml query language. http://www.w3.org/TR/xquery.

BORKAR, V., ALSUBAIEE, S., ALTOWIM, Y., ALTWAIJRY, H., BEHM, A., BU, Y., CAREY, M., GROVER, R., HEILBRON, Z., KIM, Y.-S., LI, C., PIRZADEH, P., ONOSE, N., VERNICA, R., AND WEN, J. 2012a. ASTERIX: An open source system for big data management and analysis. *Proc. VLDB Endow. 5,* 2.

Borkar, V. R., Carey, M. J., Grover, R., Onose, N., and Vernica, R. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE'11).* 1151–1162.

Borkar, V. R., Carey, M. J., and Li, C. 2012b. Inside "big data management": Ogres, onions, or parfaits? In *Proceedings of the 15th International Conference on Extending Database Technology (EDBT'12).* 3–14.

Bray, T., Paoli, J., Sperberg-Mcqueen, C. M., Maler, E., and Yergeau, F. 2008. Extensible markup language (xml) 1.0, 5th ed. http://www.w3.org/TR/REC-xml/.

Bu, Y., Howe, B., Balazinska, M., and Ernst, M. 2010. HaLoop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow. 3*, 1, 285–296.

Cafarella, M. J. and Re, C. 2010. Manimal: Relational optimization for data-intensive programs. In *Proceedings of the 13th International Workshop on the Web and Databases (WebDB'10).*

Cary, A., Sun, Z., Hristidis, V., and Rishe, N. 2009. Experiences on processing spatial data with mapreduce. In *Proceedings of the 21st International Conference on Scientific and Statistical Database Management (SSDBM'09).* 302–319.

Chaiken, R., Jenkins, B., Larson, P., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. 2008. SCOPE: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1*, 2, 1265–1276.

Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. 2010. FlumeJava: Easy, efficient data-parallel pipelines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10).* 363–375.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. 2008. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2.

Chattopadhyay, B., Lin, L., Liu, W., Mittal, S., Aragonda, P., Lychagina, V., Kwon, Y., and Wong, M. 2011. Tenzing a sql implementation on the mapreduce framework. *Proc. VLDB Endow. 4,* 12, 1318–1327.

Chen, R., Weng, X., He, B., and Yang, M. 2010. Large graph processing in the cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10).* 1123–1126.

Condie, T., Chu, D., Hellerstein, J. M., and Maniatis, P. 2008. Evita raced: Metacompilation for declarative networks. *Proc. VLDB Endow. 1*, 1, 1153–1165.

Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. 2010a. MapReduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10).* 313–328.

Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Gerth, J., Talbot, J., Elmeleegy, K., and Sears, R. 2010b. Online aggregation and continuous query support in mapreduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10).* 1115–1118.

Cordeiro, R. L. F., Traina, C., Jr., Traina, A. J. M., Lopez, J., Kang, U., and Faloutsos, C. 2011. Clustering very large multi-dimensional datasets with mapreduce. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'11).* 690–698.

Das, S., Sismanis, Y., Beyer, K., Gemulla, R., Haas, P., and McPherson, J. 2010. Ricardo: Integrating r and hadoop. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10).* 987–998.

Dean, J. and Ghemawat, S. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI'04).* 137–150.

Dean, J. and Ghemawat, S. 2008. MapReduce: Simplified data processing on large clusters. *Comm. ACM 51*, 1, 107–113.

Dean, J. and Ghemawat, S. 2010. MapReduce: A flexible data processing tool. *Comm. ACM 53*, 1, 72–77.

Dewitt, D. J. and Gray, J. 1992. Parallel database systems: The future of high performance database systems. *Comm. ACM 35*, 6, 85–98.

Dittrich, J., Quiane -Ruiz, J., Jindal, A., Kargin, Y., Setty, V., and Schad, J. 2010. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow. 3,* 1, 518–529.

Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. 2010. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10).* 810–818.

Elghandour, I. and Aboulnaga, A. 2012a. ReStore: Reusing results of mapreduce jobs. *Proc. VLDB Endow. 5,* 6, 586–597.

Elghandour, I. and Aboulnaga, A. 2012b. ReStore: Reusing results of mapreduce jobs in pig. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10).* 701–704.

Eltabakh, M. Y., Tian, Y., Ozcan, F., Gemulla, R., Krettek, A., and McPherson, J. 2011. CoHadoop: Flexible data placement and its exploitation in hadoop. *Proc. VLDB Endow. 4*, 9, 575–585.

ENE, A., IM, S., AND MOSELEY, B. 2011. Fast clustering using mapreduce. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'11).* 681–689.

FEGARAS, L., LI, C., GUPTA, U., AND PHILIP, J. 2011. XML query optimization in map-reduce. In *Proceedings of the International Workshop on the Web and Databases (WebDB).*

FLORATOU, A., PATEL, J. M., SHEKITA, E. J., AND TATA, S. 2011. Column-oriented storage techniques for mapreduce. *Proc. VLDB Endow. 4*, 7, 419–429.

FRIEDMAN, E., PAWLOWSKI, P., AND CIESLEWICZ, J. 2009. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow. 2,* 2, 1402–1413.

GATES, A. 2011. *Programming Pig*. O'Reilly Media.

GATES, A., NATKOVICH, O., CHOPRA, S., KAMATH, P., NARAYANAM, S., OLSTON, C., REED, B., SRINIVASAN, S., AND SRIVASTAVA, U. 2009. Building a highlevel dataflow system on top of mapreduce: The pig experience. *Proc. VLDB Endow. 2*, 2, 1414–1425.

GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. 2003. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03).* 29–43.

GHOTING, A., KAMBADUR, P., PEDNAULT, E. P. D., AND KANNAN, R. 2011a. NIMBLE: A toolkit for the implementation of parallel data mining and machine learning algorithms on mapreduce. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'11).* 334–342.

GHOTING, A., KRISHNAMURTHY, R., PEDNAULT, E. P. D., REINWALD, B., SINDHWANI, V., TATIKONDA, S., TIAN, Y., AND VAITHYANATHAN, S. 2011b. SystemML: Declarative machine learning on mapreduce. In *Proceedings of the IEEE 27th International Conference on Data Engineering (ICDE'11).* 231–242.

HALEVY, A. Y. 2001. Answering queries using views: A survey. *VLDB J. 10*, 4, 270–294.

HE, Y., LEE, R., HUAI, Y., SHAO, Z., JAIN, N., ZHANG, X., AND XU, Z. 2011. RCFile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Proceedings of the IEEE 27th International Conference on Data Engineering (ICDE'11).* 1199–1208.

HERODOTOU, H. 2011. Hadoop performance models. CoRR abs/1106.0940. http://arxiv.org/abs/1106.0940.

HERODOTOU, H. AND BABU, S. 2011. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proc. VLDB Endow. 4*, 11, 1111–1122.

HERODOTOU, H., DONG, F., AND BABU, S. 2011a. Mapreduce programming and cost-based optimization? crossing this chasm with starfish. *Proc. VLDB Endow. 4*, 12, 1446–1449.

HERODOTOU, H., LIM, H., LUO, G., BORISOV, N., DONG, L., CETIN, F. B., AND BABU, S. 2011b. Starfish: A self-tuning system for big data analytics. In *Proceedings of the 5th Conference on Innovative Data Systems Research (CIDR'11).* 261–272.

HEY, T., TANSLEY, S., AND TOLLE, K., EDS. 2009. The fourth paradigm: Data-intensive scientific discovery. Microsoft Research. http://research.microsoft.com/en-us/collaboration/fourthparadigm/4th_paradigm_book_complete_lr.pdf.

HINDMAN, B., KONWINSKI, A., ZAHARIA, M., AND STOICA, I. 2009. A common substrate for cluster computing. In *HotCloud Workshop held in conjunction with the USENIX Annual Technical Conference.* https://www.usenix.org/legacy/event/hotcloud09/tech/full_papers/hindman.pdf.

HUANG, J., ABADI, D. J., AND REN, K. 2011. Scalable sparql querying of large rdf graphs. *Proc. VLDB Endow. 4*, 11, 1123–1134.

HUSAIN, M. F., MCGLOTHLIN, J. P., MASUD, M. M., KHAN, L. R., AND THURAISINGHAM, B. M. 2011. Heuristics-based query processing for large rdf graphs using cloud computing. *IEEE Trans. Knowl. Data Engin. 23*, 9, 1312–1327.

ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07).* 59–72.

JAHANI, E., CAFARELLA, M. J., AND RE, C. 2011. Automatic optimization for mapreduce programs. *Proc. VLDB Endow. 4,* 6, 385–396.

JIANG, D., OOI, B. C., SHI, L., AND WU, S. 2010. The performance of mapreduce: An in-depth study. *Proc. VLDB Endow. 3*, 1, 472–483.

JIANG, D., TUNG, A. K. H., AND CHEN, G. 2011. MAP-JOIN-REDUCE: Toward scalable and efficient data analysis on large clusters. *IEEE Trans. Knowl. Data Engin. 23*, 9, 1299–1311.

JINDAL, A., QUIANE-RUIZ, J.-A., AND DITTRICH, J. 2011. Trojan data layouts: Right shoes for a running elephant. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC'11).*

KALDEWEY, T., SHEKITA, E. J., AND TATA, S. 2012. Clydesdale: Structured data processing on mapreduce. In *Proceedings of the 15th International Conference on Extending Database Technology (EDBT'12).* 15–25.

KANG, U., MEEDER, B., AND FALOUTSOS, C. 2011a. Spectral analysis for billion-scale graphs: Discoveries and implementation. In *Proceedings of the 15th Pacific-Asia conference on Advances in Knowledge Discovery and Data Mining (PAKDD'11)*. 13–25.

KANG, U., TONG, H., SUN, J., LIN, C.-Y., AND FALOUTSOS, C. 2011b. GBASE: A scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'11)*. 1091–1099.

KANG, U., TSOURAKAKIS, C. E., AND FALOUTSOS, C. 2009. PEGASUS: A peta-scale graph mining system. In *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM'09)*. 229–238.

KANG, U., TSOURAKAKIS, C. E., AND FALOUTSOS, C. 2011c. PEGASUS: Mining peta-scale graphs. *Knowl. Inf. Syst. 27*, 2, 303–325.

KHATCHADOURIAN, S., CONSENS, M. P., AND SIMEON, J. 2011. Having a chuql at xml on the cloud. In *Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW'11)*.

KIM, H., RAVINDRA, P., AND ANYANWU, K. 2011. From sparql to mapreduce: The journey using a nested triplegroup algebra. *Proc. VLDB Endow. 4*, 12, 1426–1429.

KOLB, L., THOR, A., AND RAHM, E. 2012a. Dedoop: Efficient deduplication with hadoop. *Proc. VLDB Endow. 5,* 12.

KOLB, L., THOR, A., AND RAHM, E. 2012b. Load balancing for mapreduce-based entity resolution. In *Proceedings of the 28th International Conference on Data Engineering (ICDE'12)*. 618–629.

KUMAR, V., ANDRADE, H., GEDIK, B., AND WU, K.-L. 2010. DEDUCE: At the intersection of mapreduce and stream processing. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT'10)*. 657–662.

LAMPORT, L. 1998. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2, 133–169.

LARGE SYNOPTIC SURVEY. 2013. http://www.lsst.org/.

LATTANZI, S., MOSELEY, B., SURI, S., AND VASSILVITSKII, S. 2011. Filtering: A method for solving graph problems in mapreduce. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. 85–94.

LIM, H., HERODOTOU, H., AND BABU, S. 2012. Stubby: A transformation-based optimizer for mapreduce workflows. *Proc. VLDB Endow. 5*, 12.

LIN, J. J. 2009. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'09)*. 155–162.

LIN, Y., AGRAWAL, D., CHEN, C., OOI, B. C., AND WU, S. 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 (SIGMOD'11)*. 961–972.

LOGOTHETIS, D. AND YOCUM, K. 2008. Ad-hoc data processing in the cloud. *Proc. VLDB Endow. 1,* 2, 1472–1475.

LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. 2005. Implementing declarative overlays. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. 75–90.

LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. 2010. GraphLab: A new framework for parallel machine learning. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI'10)*. 340–349.

LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. 2012. Distributed graphlab: A framework for machine learning in the cloud. *Proc. VLDB Endow. 5*, 8, 716–727.

MALEWICZ, G., AUSTERN, M., BIK, A., DEHNERT, J., HORN, I., LEISER, N., AND CZAJKOWSKI, G. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. 135–146.

MANOLA, F. AND MILLER, E. 2004. RDF Primer, W3C Recommendation. http://www.w3.org/TR/REC-rdf-syntax/.

MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. 2010. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow. 3*, 1, 330–339.

METWALLY, A. AND FALOUTSOS, C. 2012. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proc. VLDB Endow. 5*, 8, 704–715.

MORALES, G. F., GIONIS, A., AND SOZIO, M. 2011. Social content matching in mapreduce. *Proc. VLDB Endow. 4*, 7, 460–469.

MORTON, K., BALAZINSKA, M., AND GROSSMAN, D. 2010a. ParaTimer: A progress indicator for mapreduce dags. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. 507–518.

MORTON, K., FRIESEN, A., BALAZINSKA, M., AND GROSSMAN, D. 2010b. Estimating the progress of mapreduce pipelines. In *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE'10)*. 681–684.

MYUNG, J., YEON, J., AND GOO LEE, S. 2010. SPARQL basic graph pattern processing with iterative mapreduce. In *Proceedings of the Workshop on Massive Data Analytics on the Cloud (MDAC'10)*.

NEUMANN, T. AND WEIKUM, G. 2008. RDF-3x: A risc-style engine for rdf. *Proc. VLDB Endow. 1*, 1.

NYKIEL, T., POTAMIAS, M., MISHRA, C., KOLLIOS, G., AND KOUDAS, N. 2010. MRShare: Sharing across multiple queries in mapreduce. *Proc. VLDB Endow. 3*, 1, 494–505.

ODERSKY, M., SPOON, L., AND VENNERS, B. 2011. *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima Inc.

OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. 2008. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. 1099–1110.

PANDA, B., HERBACH, J., BASU, S., AND BAYARDO, R. J. 2009. PLANET: Massively parallel learning of tree ensembles with mapreduce. *Proc. VLDB Endow, 2,* 2, 1426–1437.

PAPADIMITRIOU, S. AND SUN, J. 2008. DisCo: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM'08)*. 512–521.

PATTERSON, D. A. 2008. Technical perspective: The data center is the computer. *Comm. ACM 51*, 1, 105.

PAVLO, A., PAULSON, E., RASIN, A., ABADI, D., DEWITT, D., MADDEN, S., AND STONEBRAKER, M. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*. 165–178.

PIKE, R., DORWARD, S., GRIESEMER, R., AND QUINLAN, S. 2005. Interpreting the data: Parallel analysis with sawzall. *Sci. Program. 13*, 4, 277–298.

PRUDHOMMEAUX, E. AND SEABORNE, A. 2008. SPARQL query language for rdf, w3c recommendation. http://www.w3.org/TR/rdf-sparql-query/.

QUIANE-RUIZ, J.-A., PINKEL, C., SCHAD, J., AND DITTRICH, J. 2011a. RAFT at work: Speeding-up mapreduce applications under task and node failures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*. 1225–1228.

QUIANE-RUIZ, J.-A., PINKEL, C., SCHAD, J., AND DITTRICH, J. 2011b. RAFTing mapreduce: Fast recovery on the raft. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE'11)*. 589–600.

RAVINDRA, P., KIM, H., AND ANYANWU, K. 2011. An intermediate algebra for optimizing rdf graph pattern matching on mapreduce. In *Proceedings of the 8th Extended Semantic Web Conference on the Semanic Web: Research and Applications (ESWC'11)*. 46–61.

SCHATZLE, A., PRZYJACIEL-ZABLOCKI, M., HORNUNG, T., AND LAUSEN, G. 2011. PigSPARQL: Mapping sparql to pig latin. In *Proceedings of the International Workshop on Semantic Web Information Management (SWIM'11)*. 65–84.

SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. 1979. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'79)*. 23–34.

STONEBRAKER, M. 1986. The case for shared nothing. *IEEE Datab. Engin. Bull. 9*, 1, 4–9.

STONEBRAKER, M., ABADI, D., DEWITT, D., MADDEN, S., PAULSON, E., PAVLO, A., AND RASIN, A. 2010. MapReduce and parallel dbmss: Friends or foes? *Comm. ACM 53*, 1, 64–71.

STUTZ, P., BERNSTEIN, A., AND COHEN, W. W. 2010. Signal/collect: Graph algorithms for the (semantic) web. In *Proceedings of the International Semantic Web Conference*. 764–780.

THUSOO, A., SARMA, J., JAIN, N., SHAO, Z., CHAKKA, P., ANTHONY, S., LIU, H., WYCKOFF, P., AND MURTHY, R. 2009. Hive - A warehousing solution over a map-reduce framework. *Proc. VLDB Endow. 2,* 2, 1626–1629.

THUSOO, A., SARMA, J., JAIN, N., SHAO, Z., CHAKKA, P., ZHANG, N., ANTHONY, S., LIU, H., AND MURTHY, R. 2010a. Hive - A petabyte scale data warehouse using hadoop. In *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE'10)*. 996–1005.

THUSOO, A., SHAO, Z., ANTHONY, S., BORTHAKUR, D., JAIN, N., SARMA, J. S., MURTHY, R., AND LIU, H. 2010b. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. 1013–1020.

ULLMAN, J. D. 1990. *Principles of Database and Knowledge Base Systems: Volume II: The New Technologies*. W. H. Freeman and Co., New York.

VALIANT, L. G. 1990. A bridging model for parallel computation. *Comm. ACM 33*, 8, 103–111.

VERNICA, R., CAREY, M., AND LI, C. 2010. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. 495–506.

WANG, C., WANG, J., LIN, X., WANG, W., WANG, H., LI, H., TIAN, W., XU, J., AND LI, R. 2010. MapDupReducer: Detecting near duplicates over massive datasets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. 1119–1122.

WANG, G., XIE, W., DEMERS, A., AND GEHRKE, J. 2013. Asynchronous large-scale graph processing made easy. In *Proceedings of the 7th Conference on Innovative Data Systems Research (CIDR'13)*.

WHITE, T. 2012. *Hadoop: The Definitive Guide*. O'Reilly Media.

XIAO, C., WANG, W., LIN, X., YU, J. X., AND WANG, G. 2011. Efficient similarity joins for near-duplicate detection. *ACM Trans. Datab. Syst. 36*, 3, 15.

YANG, H., DASDAN, A., HSIAO, R., AND PARKER, D. 2007. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. 1029–1040.

YANG, H. AND PARKER, D. 2009. Traverse: Simplified indexing on large map-reduce-merge clusters. In *Proceedings of the 14th International Conference on Database Systems for Advanced Applications (DASFAA'09)*. 308–322.

YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P., AND CURREY, J. 2008. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 1–14.

ZAHARIA, M., BORTHAKUR, D., SARMA, J. S., ELMELEEGY, K., SHENKER, S., AND STOICA, I. 2010a. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems* (*EuroSys'10)*. 265–278.

ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*.

ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. 2010b. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. 10.

ZAHARIA, M., KONWINSKI, A., JOSEPH, A., KATZ, R., AND STOICA, I. 2008. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 29–42.

ZHANG, Y., GAO, Q., GAO, L., AND WANG, C. 2012. IMapReduce: A Distributed Computing Framework for Iterative Computation. *J. Grid Comput. 10*, 1, 47–68.

ZHOU, J., LARSON, P., AND CHAIKEN, R. 2010. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE'10)*. 1060–1071.

ZUKOWSKI, M., BONCZ, P. A., NES, N., AND HEMAN, S. 2005. MonetDB/X100 - A dbms in the cpu cache. *IEEE Data Engin. Bull. 28*, 2, 17–22.

Received July 2012; revised November 2012; accepted January 2013