Chapter 1
The Family of Map-Reduce

Sherif Sakr and Anna Liu

Abstract In the last two decades, the continuous increase of computational power has produced an overwhelming flow of data, which 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 that can process vast amounts of data on large clusters of commodity machines. MapReduce 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 following up work. This chapter provides a comprehensive survey for a family of approaches and mechanisms of large scale data analysis that have been implemented based on the original father idea of the MapReduce framework, and are currently gaining a lot of momentum in both research and industrial communities. Some case studies are discussed as well.

1.1 Introduction

In the last two decades, the continuous increase of computational power has produced an overwhelming flow of data which 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. Facebook collects 15 TB of data each day into a PetaByte-scale data warehouse. Jim Gray, a database software pioneer and a Microsoft researcher, called the shift a "*fourth paradigm*" [26]. The first three paradigms were *experimental*, *theoretical* and,

S. Sakr (✉) • A. Liu
NICTA and University of New South Wales, Sydney, NSW, Australia
e-mail: Sherif.Sakr@nicta.com.au; Anna.Liu@nicta.com.au

A. Gkoulalas-Divanis and A. Labbi (eds.), *Large-Scale Data Analytics*,
DOI 10.1007/978-1-4614-9242-9_1, © Springer Science+Business Media New York 2014

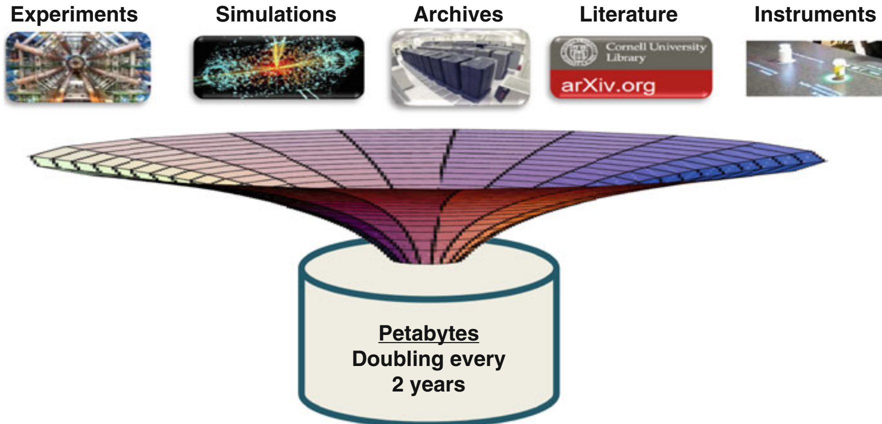

Fig. 1.1 Data explosion in scientific computing [26]

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 multi-core CPUs and mechanical hard disks is growing every year, which makes the challenges of data-intensive computing much harder to overcome [8]. 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 cheaper clusters of computers to manage and process all this data, instead of focusing on having the biggest and fastest single computer.

Figure 1.1 illustrates an example of the explosion in scientific data which creates major challenges for cutting-edge scientific projects. For example, modern high-energy physics experiments, such as *DZero*,[1] typically generate more than 1 TB of data per day. With datasets growing beyond a few hundreds of terabytes, scientists have no off-the-shelf solutions that they can readily use to manage and analyze these data [26]. Thus, significant human and material resources were allocated to support these data-intensive operations, which led to high storage and management costs.

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) [37]. Although parallel database systems serve some of these data analysis applications, they are expensive, difficult to administer and lack fault-tolerance for long-running queries [34]. MapReduce [16] is a framework which is introduced by Google for programming

[1] http://www-d0.fnal.gov/.

commodity computer clusters to perform large-scale data processing in a single pass. The framework is designed in a way that a MapReduce cluster can scale to thousands of nodes in a fault-tolerant manner. An important advantage of this framework is its reliance on a simple and powerful programming model. In addition, MapReduce isolates the application developer from all the complex details of running a distributed program, such as issues on data distribution, scheduling and fault tolerance.

Recently, there has been a great deal of hype about cloud computing [5]. 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 the network 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.

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 had been considered as 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)[2] is offered as a commodity that can be purchased and utilised. In addition, Amazon has also provided the Amazon Elastic MapReduce[3] as an online service to easily and cost-effectively process vast amounts of data without the need to worry about time-consuming set-up, 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. As a result, many research efforts have been triggered to tackle these limitations by introducing several advancements in the basic architecture in order to improve its performance. This chapter provides a comprehensive survey for a *family* of approaches and mechanisms of large scale data analysis that have been implemented based on the original *father* 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 chapter is organized as follows. Section 1.2 describes the basic architecture of the MapReduce framework. Section 1.3 discusses several techniques that have been proposed to improve the performance and

[2]http://aws.amazon.com/ec2/.

[3]http://aws.amazon.com/elasticmapreduce/.

capabilities of the MapReduce framework. Section 1.4 gives an overview of several systems that support high level SQL-like interface for the MapReduce framework, while Sect. 1.5 discusses the hybrid systems that support both MapReduce and SQL-like interfaces. Several case studies are discussed in Sect. 1.6, before we conclude the chapter in Sect. 1.7.

1.2 The MapReduce Framework: Basic Architecture

The MapReduce framework is introduced as a simple and powerful programming model that enables easy development of scalable parallel applications which can process vast amounts of data on large clusters of commodity machines [16, 17]. In particular, the framework 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 input key/value pairs and produces a set of output key/value pairs.

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 re-executed to be used as the primary mechanism for fault tolerance.

Figure 1.2 illustrates an example MapReduce program expressed in pseudo-code 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 mark of occurrences, while the reduce function sums together all marks emitted for a particular word. In principle, the design of the MapReduce framework has considered the following main principles [46]:

```
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. 1.2 An example of a MapReduce program [16]

- *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 kept 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 in rackable systems that are connected with generic LAN switches. 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 off-line. 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 DBMS.
- *Highly Parallel yet Abstracted*: The most important contribution of the Map-Reduce 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, multi-threaded or network programming. Moreover, MapReduce's shared-nothing architecture [38] makes it much more scalable and ready for parallelization.

Hadoop[4] is an open source Java software that supports data-intensive distributed applications by realizing the implementation of the MapReduce framework. On the implementation level, the Map invocations are 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 1.3 illustrates an example of the overall flow of a MapReduce operation, which goes through the following sequence of actions:

1. The input files of the MapReduce program are split into M pieces and many copies of the program start up on a cluster of machines.

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

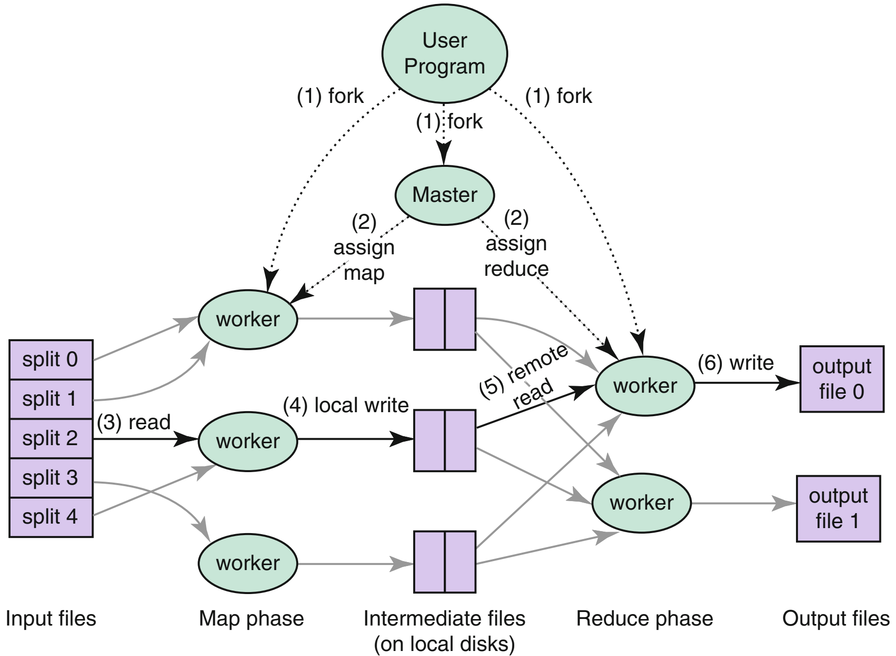

Fig. 1.3 An overview of the flow of execution in a MapReduce operation [16]

2. One of the copies 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 a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the corresponding input split, parses key/value pairs out of 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, 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.

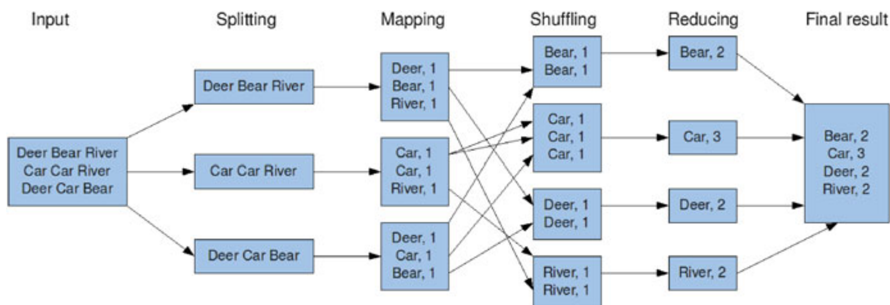

Fig. 1.4 Execution steps of the `WordCount` example using MapReduce

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 back to the user code.

Figure 1.4 illustrates a sample execution for the example program (`WordCount`), depicted in Fig. 1.2, using the steps of the MapReduce framework, which are illustrated in Fig. 1.3. During the execution process, the master pings every worker periodically. If no response is received from a worker in 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 on other workers. Completed map tasks are re-executed on a 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 re-executed since their output is stored in a global file system.

1.3 Improvements on 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 complicated functions that can be directly supported in SQL. However, the basic architecture suffers from certain limitations. Dean and Ghemawat [18] reported a set of possible improvements that need to be incorporated into the MapReduce framework. These include:

- MapReduce should take advantage of natural indices whenever possible.
- Most MapReduce output should 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.

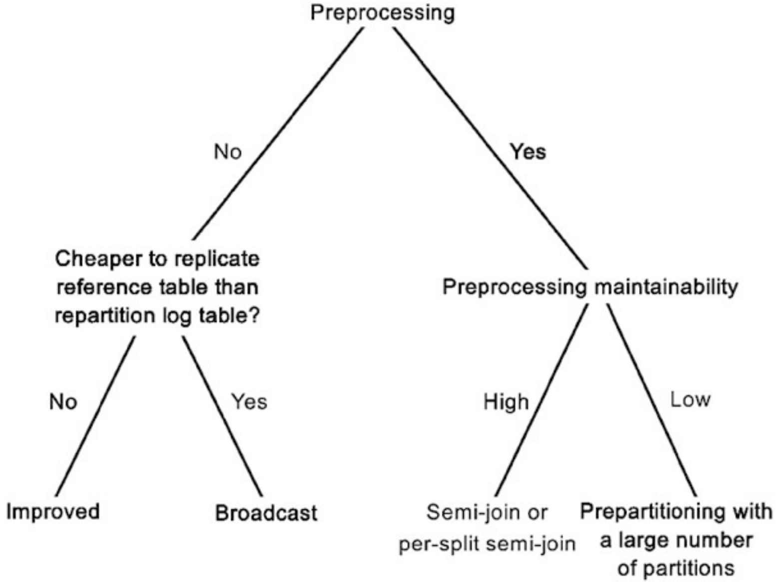

Fig. 1.5 Decision tree for choosing between various join strategies on the MapReduce framework [10]

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

1.3.1 Map-Reduce-Merge

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. [10] report on a study that evaluated the performance of different distributed join algorithms (e.g., Repartition Join, Broadcast Join) using the MapReduce framework. Figure 1.5 illustrates a decision tree that summaries the tradeoffs of the considered 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

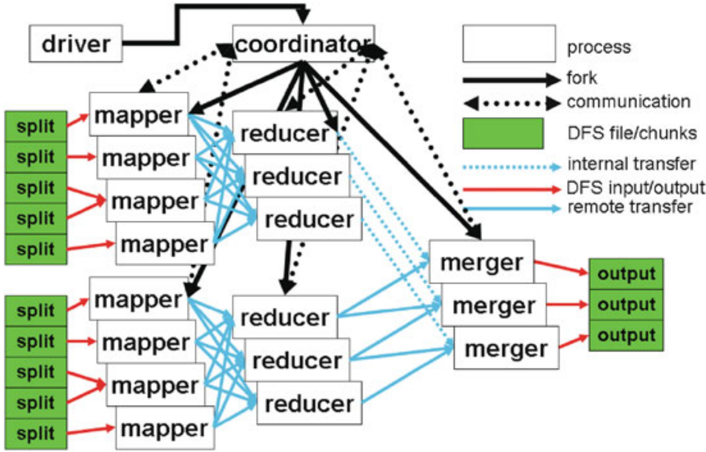

Fig. 1.6 An overview of the Map-Reduce-Merge framework [46]

join algorithm should be used. When preprocessing is allowed, semi-join, per-split semi-join and directed join with enough 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, the preprocessing steps of these two algorithms are cheaper since there is no shuffling of the log data.

To tackle the limitation of the join phase in the MapReduce framework, Yang et al. [46] have proposed the Map-Reduce-Merge model that enables the processing of multiple datasets. Figure 1.6 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 needs input datasets 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 needed in $[v3]$ while passing $k2$ for the next stage is not required.

Figure 1.7 illustrates a sample execution of the Map-Reduce-Merge framework. In this example, there are two datasets: *Employee* and *Department*, where

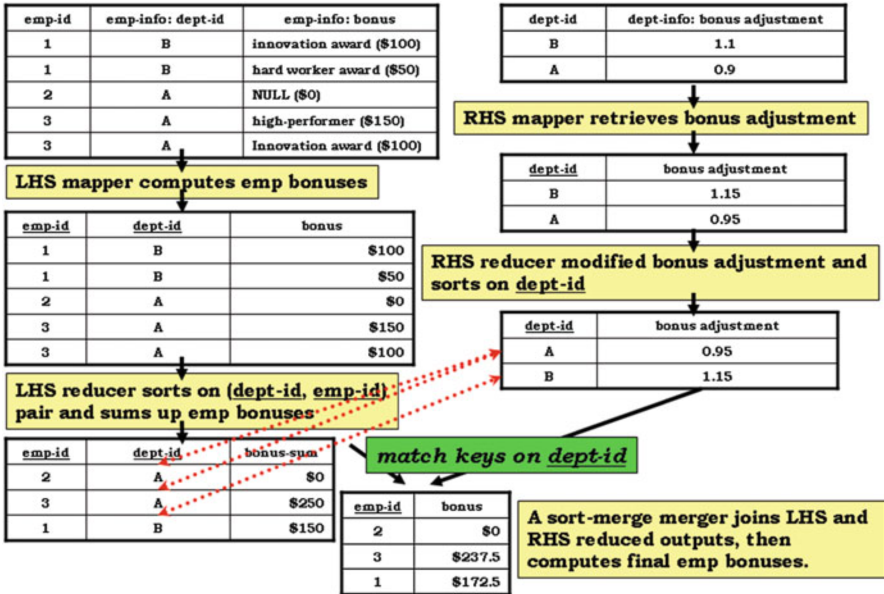

Fig. 1.7 A sample execution of the Map-Reduce-Merge framework [46]

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 Fig. 1.7, 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 [45] have also proposed an approach for improving the Map-Reduce-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.

Afrati and Ullman [3] have presented another approach to improve the join phase in the MapReduce framework. This approach 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, where 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 map-reduce to implement joins. However, there are some cases where the proposed approach results in clear wins, such as:

- Analytic queries in which a very large fact table is joined with smaller dimension tables.
- Queries involving paths through graphs with high out-degree, such as the Web or a social network.

1.3.2 MapReduce Online

The basic architecture of the MapReduce framework requires the entire output of each map and reduce task to be *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. Alvaro et al. [4] proposed a modified architecture 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:

- 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 in applications such as event monitoring and stream processing.
- Pipelining delivers data to downstream operators more promptly, which can increase opportunities for parallelism, improve utilization as well as reduce response time.

1.3.3 MRShare

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. 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. Nykiel et al. [32] have proposed *MRShare* 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 associated cost. In particular, the proposed 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 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 yield I/O savings, while sharing map functions (or parts of them) additionally yield CPU savings.

1.3.4 HaLoop

Many data analysis techniques (e.g., the PageRank algorithm, recursive relational queries, social network analysis) require iterative computations. These techniques have a common requirement which is that data are processed iteratively until the computation satisfies a convergence or stopping condition. 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:

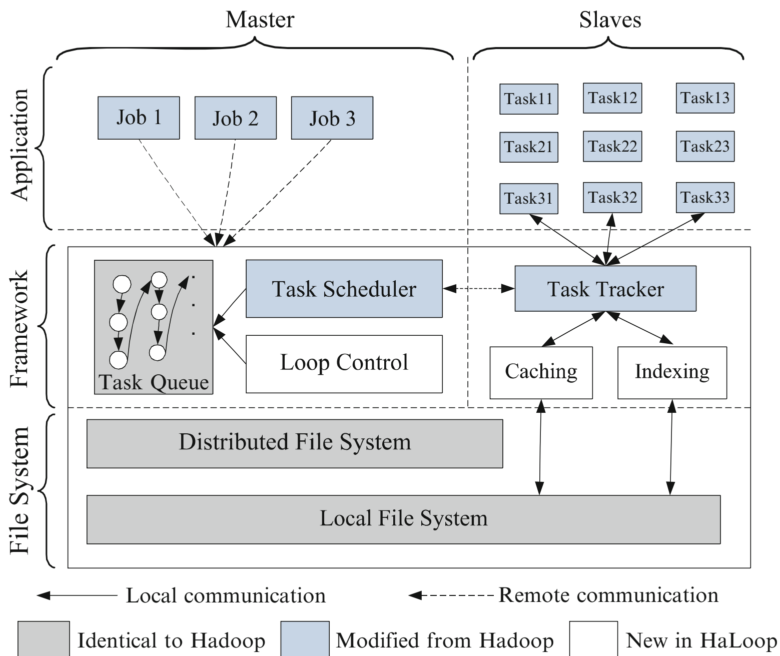

Fig. 1.8 An overview of the HaLoop architecture [11]

- Even though much of the data may be unchanged from iteration to iteration, the data must be re-loaded and re-processed 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.

Bu et al. [11] have presented the *HaLoop* system which is designed to efficiently handle the above types of applications. HaLoop extends the basic MapReduce framework with two main functionalities:

1. A MapReduce cluster can cache the invariant data in the first iteration and then reuse them in later iterations.
2. A MapReduce cluster can cache reducer outputs, which makes checking for a fixpoint more efficient, without an extra MapReduce job.

Figure 1.8 illustrates the architecture of HaLoop as a modified version of the basic MapReduce framework. 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 for iterative applications that leverages data locality in these applications.
- 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.

1.3.5 Hadoop++

An important limitation of the Basic MapReduce framework is that it is designed in a way that jobs can only scan the input data in a *sequential*-oriented fashion. Hence, the query processing performance of the MapReduce framework does not match the one of a well-configured parallel DBMS [34]. In order to tackle this challenge, Dittrich et al. [19] have presented the *Hadoop*++ system, which aims to boost the query performance of the Hadoop project (the open source implementation of the MapReduce framework) without changing any of the system internals. They achieve 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 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 non-invasive 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.
  - *Non-Invasive*: 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 co-partition 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 co-group 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.

1.3.6 CoHadoop

In the basic implementation of the Hadoop project, the objective of the data placement policy is to achieve load balancing 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 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 co-location increases the data shuffling costs, increases the network overhead and reduces the effectiveness of data partitioning. For example, log processing is a very common usage scenario for the Hadoop framework. In this scenario, data are accumulated in batches from event logs, such as clickstreams, phone call records, application logs or a sequences of transactions. Each batch of data is ingested into Hadoop and stored in one or more HDFS files at regular intervals. Two of the most common operations in log analysis of these applications are (1) joining the log data with some reference data and (2) sessionization, i.e., computing user sessions. The performance of such operations can be significantly improved if they utilize the benefits of data co-location.

CoHadoop [20] is a lightweight extension to Hadoop which is designed to enable co-locating related files at the file system level, while at the same time retaining the good load balancing and fault tolerance properties. CoHadoop introduces a new file property to identify related data files and modify the data placement policy

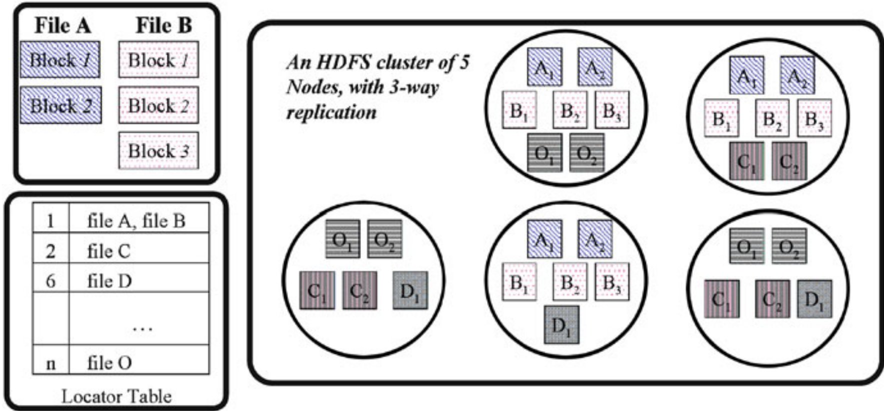

Fig. 1.9 Example file co-location in CoHadoop [20]

of Hadoop to co-locate all copies of those related files in the same server. These changes are designed in a way that retains 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 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 datanodes, whereas files with no locator are placed via Hadoop's default strategy. It should be noted that this co-location process involves all data blocks, including replicas. Figure 1.9 shows an example of co-locating 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 datanodes. To manage the locator information and keep track of co-located 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 co-location by simply specifying related files, such as co-locating log files with reference files for joins, co-locating partitions for grouping and aggregation, co-locating index files with their data files and co-locating columns of a table.

1.4 SQL-Like MapReduce Implementations

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

- Its one-input and two-stage data flow is extremely rigid. As we previously discussed, to perform tasks having a different data flow (e.g. joins or *n* stages), inelegant workarounds have to be devised.
- 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.
- The opaque nature of the map and reduce functions impedes the ability of the system to perform optimizations.

Moreover, many programmers could be unfamiliar with the MapReduce framework and they would prefer to use SQL (because they are more proficient in) 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 what follows, we discuss research efforts to tackle these problems and add the SQL flavor on top of the MapReduce framework.

1.4.1 Pig Latin

Gates et al. [23] have presented a programming language, called *Pig Latin*, which takes a *middle* position between expressing tasks using a high-level declarative querying model in the spirit of SQL, and low-level/procedural programming using MapReduce. Pig Latin is implemented in the scope of the *Apache Pig* project[5] 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 1.10 shows an example SQL query and its equivalent Pig Latin program. Given a URL table with the structure $(url, category, pagerank)$, 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,

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

SQL	**Pig Latin**

SELECT category, **AVG**(pagerank)
FROM urls
WHERE pagerank > 0.2
GROUP BY category
HAVING COUNT(*) > 10^6
	good_urls = **FILTER** urls **BY** pagerank > 0.2;
	groups = **GROUP** good_urls **BY** category;
	big_groups = **FILTER** groups **BY COUNT**(good_urls)>10^6;
	output = **FOREACH** big_groups **GENERATE**
	category, **AVG**(good_urls.pagerank);

Fig. 1.10 An example SQL query and its equivalent Pig Latin program [23]

extensive support for user-defined functions and the ability to operate over plain input files without any schema information. In particular, Pig Latin has a simple data model consisting of the following four types:

- *Atom*: An atom contains a simple atomic value, such as a string or a number, e.g., "alice".
- *Tuple*: A tuple is a sequence of fields, each of which can be any of the data types, e.g., ("alice", "lakers").
- *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
  e.g., $\left\{ \begin{array}{l} (\text{"alice", "lakers"}) \\ (\text{"alice", ("iPod", "apple")}) \end{array} \right\}$
- *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, e.g., $\left\{ \begin{array}{l} \text{"k1"} \rightarrow (\text{"alice", "lakers"}) \\ \text{"k2"} \rightarrow \text{"20"} \end{array} \right\}$

To accommodate specialized data processing tasks, Pig Latin has extensive support for user-defined functions. 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 [33] before being executed, as depicted in Fig. 1.11. The parsing steps verifies 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

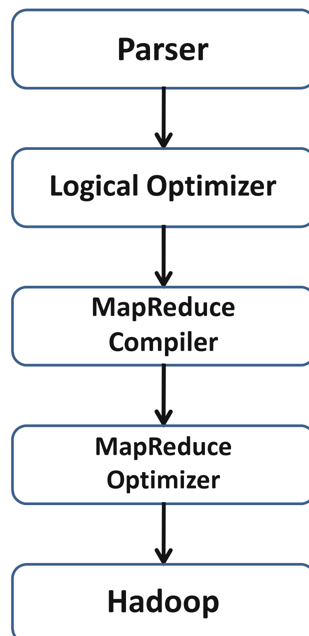

Fig. 1.11 Pig compilation and execution steps [33]

compiled into a series of MapReduce jobs, which are then passed through another optimization phase. The DAG of optimized MapReduce jobs is then topologically sorted and jobs are submitted to Hadoop for execution.

1.4.2 Sawzall

Sawzall [35] 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 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, and (2) drawing an explicit line between filtering and aggregation enables a high degree of parallelism and hides the parallelism from the language itself.

Figure 1.12 depicts an example Sawzall program where the first three lines declare the aggregators *count*, *total* and *sum of squares*. The keyword *table*

```
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. 1.12 An example of a Sawzall program [35]

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 *floats* as appropriate. The Sawzall language is implemented as a conventional compiler, written in C++, whose target language is an interpreted instruction set, or byte-code. The compiler and the byte-code 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) [24]. The business of scheduling a job to execute on a cluster of machines is handled by 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.

1.4.3 SQL/MapReduce

In general, a user-defined function is a powerful database feature that allows users to customize database functionality. Friedman et al. [22] 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*[6] nCluster shared-nothing relational database.

The framework leverages 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

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

Fig. 1.13 Basic syntax of SQL/MR query function [22]

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

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 otherwise leverage traditional SQL functionality. A SQL/MR function is defined in a manner that is similar to MapReduce's map and reduce functions.

The syntax for using a SQL/MR function is depicted in Fig. 1.13, 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, a 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. 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.

1.4.4 SCOPE

SCOPE (Structured Computations Optimized for Parallel Execution) is a scripting language which is targeted for large-scale data analysis and is used for a variety of data analysis and data mining applications inside Microsoft [13]. 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.

SQL-Like	MapReduce-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";	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. 1.14 Two equivalent SCOPE scripts in SQL-like style and in MapReduce-like style [13]

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 (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 1.14 illustrates two equivalent scripts in two different styles that are used to find from a search log queries that have been requested at least 1,000 times. In the MapReduce-like style, the *EXTRACT* command extracts all query string 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 1,000. The third *SELECT* command sorts the rows on count. Finally, the *OUTPUT* command writes the result to a file.

Microsoft has developed a distributed computing platform, called *Cosmos*, for storing and analyzing massive data sets. Cosmos is designed to run on large clusters consisting of thousands of commodity servers. Figure 1.15 shows the main components of the Cosmos platform, described as follows:

- *Cosmos Storage*: A distributed storage subsystem designed to reliably and efficiently store extremely large sequential files.
- *Cosmos Execution Environment*: An environment for deploying, executing and debugging distributed applications.
- *SCOPE*: A high-level scripting language for writing data analysis jobs. The SCOPE compiler and optimizer translate these 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

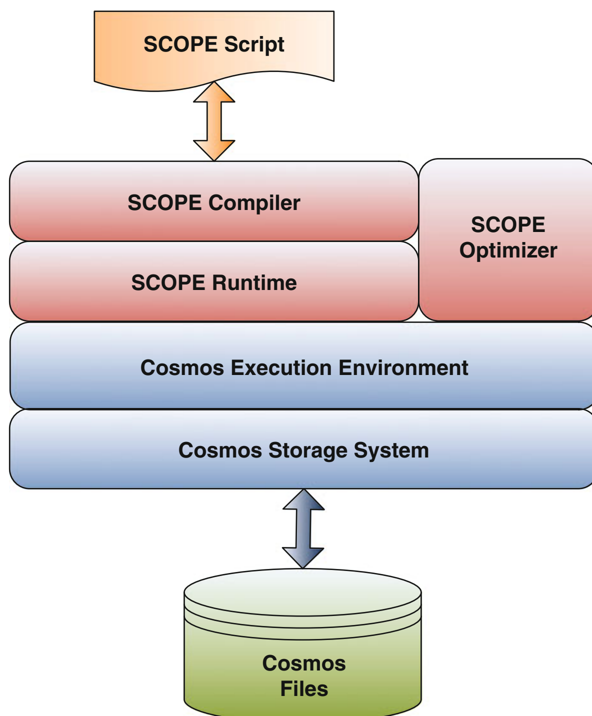

Fig. 1.15 The SCOPE/Cosmos execution platform [13]

and replicated for fault tolerance and compressed to save storage and increase I/O throughput. In Cosmos, an application is modeled as a dataflow graph: a directed acyclic graph 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. 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 set of default implementation rules. 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 pre-aggregating 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 partitioning 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 [49].

Using a similar approach to that of SCOPE, Murray and Hand [31] have presented *Skywriting* as a purely-functional script language with its execution engine for performing distributed and parallel computations. A Skywriting script can create new tasks asynchronously, evaluate data dependencies and perform unbounded (while-loop) iteration. This enables Skywriting to describe a more general class of distributed computations.

1.4.5 DryadLINQ

Dryad is a general-purpose distributed execution engine introduced by Microsoft for coarse-grain data-parallel applications [27]. A Dryad application combines computational *vertices* with communication *channels* to form a dataflow 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 offers to 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 run time, 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 [47]. It generalizes execution environments, such as SQL and MapReduce, in two ways: (1) adopting an expressive data model of strongly typed .NET objects and (2) supporting general-purpose imperative and declarative operations on datasets within a traditional high-level programming language. DryadLINQ[7] exploits LINQ (Language INtegrated Query,[8] 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#.[9] 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. Figure 1.16 illustrates the flow of execution when a program is executed by DryadLINQ [47].

1. When a .NET user application runs, it creates a DryadLINQ expression object.
2. The application triggers a data-parallel execution, where the expression object is handed to DryadLINQ.
3. DryadLINQ compiles the LINQ expression into a distributed Dryad execution plan. In particular, it performs the following tasks:
   (a) Decomposes the expression into subexpressions, where each expression can be assigned to run in a separate Dryad vertex.
   (b) Generates the code and static data for the remote Dryad vertices.
   (c) Generates the serialization code for the required data types.
4. DryadLINQ invokes a custom Dryad job manager.
5. The job manager creates the job graph and schedules the vertices as resources become available.
6. Each Dryad vertex executes a vertex-specific program as created in Step 3(b).
7. When the Dryad job completes successfully, it writes the data to the output table(s).

[7] http://research.microsoft.com/en-us/projects/dryadlinq/.

[8] http://msdn.microsoft.com/en-us/netframework/aa904594.aspx.

[9] http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/.

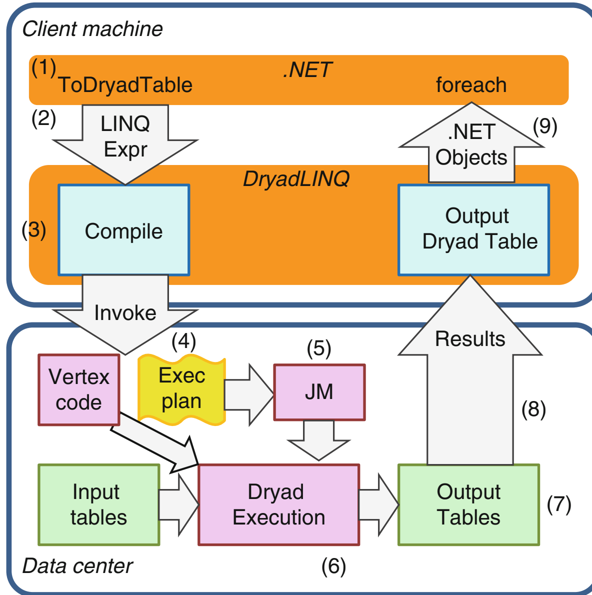

Fig. 1.16 LINQ-expression execution in DryadLINQ [47]

8. The job manager process terminates and returns control back to DryadLINQ, which creates objects encapsulating the outputs of the execution. These objects may be used as inputs to subsequent expressions in the user program.
9. Control returns to the user application. The iterator interface over a DryadTable allows the user to read its contents as .NET objects.
10. The application may generate subsequent DryadLINQ expressions that can be executed by a repetition of Steps 2–9.

1.4.6 Jaql

Jaql[10] is a query language which is designed for Javascript Object Notation (JSON),[11] 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 semi-structured, XML data. Jaql is primarily used to analyze large-scale semi-structured data. It is a functional, declarative query language which rewrites high-level queries (when appropriate) into a low-level

[10] http://code.google.com/p/jaql/.

[11] http://www.json.org/.

```
import myrecord;

count Fields = fn(records) (
  records
  -> transform myrecord: :names($)
  -> expand
  -> group by fName = $ as occurrences
     into { name: fName, num: count (occurrences) }
  );

  read(hdfs("docs.dat"))
  -> countFields()
  -> write(hdfs("fields.dat"));
```

Fig. 1.17 A sample Jaql script [9]

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 [9]. It is able to process data with no schema or only 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; a *JDM value* is either an atom, an array or a record. Most common atomic types are supported by Jaql, 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 key ingredient for reusability as any Jaql expression can be encapsulated in a function, and a function can be parameterized in powerful ways.

Figure 1.17 depicts an example of a Jaql script that consists of a sequence of operators. The read operator loads raw data, in this case from Hadoop's Distributed File System (HDFS), and converts it into Jaql values. These values are subsequently processed by the `countFields` subflow, which extracts field names and computes their frequencies. Finally, the write operator stores the result back into HDFS. In general, the core expressions of the Jaql scripting language include:

1. *Transform*: The transform expression applies a function (or projection) to every element of an array to produce a new array. It has the form `e1->transform e2`, where `e1` is an expression that describes the input array and `e2` is applied to each element of `e1`.

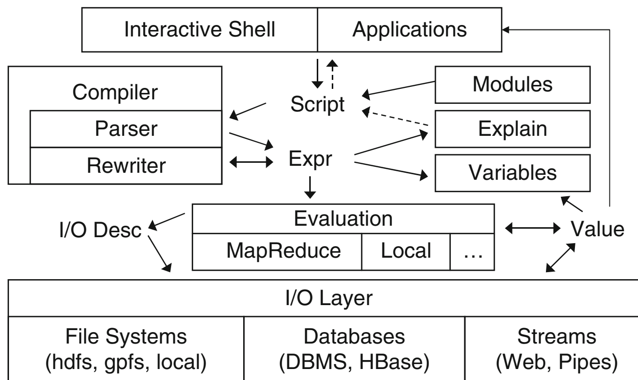

Fig. 1.18 The Jaql system architecture [9]

2. *Expand*: The expand expression is most often used to unnest an input array. It differs from transform in two primary ways: (1) `e2` must produce a value `v` that is an array type, and (2) each of the elements of `v` is returned to the output array, thereby removing one level of nesting.
3. *Group by*: Similar to SQL's GROUP BY, Jaql's group-by expression partitions its input on a grouping expression and applies an aggregation expression to each group.
4. *Filter*: The filter expression, `e− >filter p`, retains input values from `e` for which predicate `p` evaluates to true.
5. *Join*: The join expression supports equijoin of 2 or more inputs. All of the options for inner and outer joins are also supported.
6. *Union*: The union expression is a Jaql function that merges multiple input arrays into a single output array. It has the form: `union(`e_1`, ...)` where each e_i is an array.
7. *Control-Flow*: The two most commonly used control-flow expressions in Jaql are `if-then-else` and `block` expressions. The `if-then-else` expression is similar to conditional expressions found in most scripting and programming languages. A `block` establishes a local scope where zero or more local variables can be declared and the last statement provides the return value of the block.

At a high-level, the Jaql system architecture, depicted in Fig. 1.18, 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 data set 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.

1.5 Hybrid Systems

Originally, the applications of the MapReduce framework have been mainly focusing on analyzing very large non-structured datasets, e.g., web indexing, text analytics, and graph data mining. Recently, however, as MapReduce is steadily developing into the de facto data analysis standard, it repeatedly becomes employed for querying structured data [7]. For a long time, relational database and its standard query language (i.e., SQL) has dominated the deployments of data warehousing systems and data analysis on structured data. Therefore, there has been an increasing interest in combining MapReduce and traditional database systems in an effort to maintain the benefits of both worlds. In the following section, we present some systems that have been designed to achieve this goal of integrating the two environments.

1.5.1 Hive

The *Hive* project[12] is an open-source data warehousing solution which has been built by the Facebook Data Infrastructure Team on top of the Hadoop environment [40]. 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 enjoyed. 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, called *HiveQL*,[13] 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. For example, the canonical MapReduce word count example on a table of documents (Fig. 1.2) can be expressed in HiveQL as depicted in

[12] http://hadoop.apache.org/hive/.

[13] 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. 1.19 An example HiveQl query [40]

Fig. 1.19, 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 [41]. 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, such as Pig Latin [33].

1.5.2 HadoopDB

Parallel database systems have been commercially available for nearly two decades and there are now about a dozen of different implementations in the marketplace (e.g., Teradata,[14] Aster Data,[15] Netezza,[16] Vertica,[17] ParAccel,[18] Greenplum[19]). The main aim of these systems is to improve performance through the parallelization of various operations, such as loading data, building indices and

[14] http://www.teradata.com/.
[15] http://www.asterdata.com/.
[16] http://www.netezza.com/.
[17] http://www.vertica.com/.
[18] http://www.paraccel.com/.
[19] http://www.greenplum.com/.

evaluating queries. These systems are usually designed to run on top of a shared-nothing architecture [38], where data may be stored in a distributed fashion and input/output speeds are improved by using multiple CPUs and disks in parallel. On the other hand, there are some key reasons that make MapReduce a more preferable approach over a parallel RDBMS in some scenarios [10], such as:

- Formatting and loading a huge amount of data into a parallel RDBMS in a timely manner is a challenging and time-consuming task.
- The input data records may not always follow the same schema. Developers often want the flexibility to add and drop attributes, and the interpretation of an input data record may also change over time.
- Large scale data processing can be very time consuming and therefore it is important to keep the analysis job going even in the event of failures. While most parallel RDBMSs have fault tolerance support, a query usually has to be restarted from scratch, even if just one node in the cluster fails. In contrast, MapReduce deals more gracefully with failures and can redo only the part of the computation that was lost because of a failure.

There has been a long debate on the comparison between the MapReduce framework and parallel database systems[20] [39]. Pavlo et al. [34] have conducted a large scale comparison between the Hadoop implementation of the 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 displayed a significant performance advantage over MapReduce in executing a variety of data intensive analysis tasks. On the other hand, the Hadoop implementation was significantly easier and more straightforward to set up and use in comparison to that of the parallel database systems. MapReduce have also 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 cheap solution in comparison to the expensive parallel DBMS solutions [39].

The *HadoopDB* project[21] is a hybrid system that tries to combine the scalability advantages of MapReduce with the performance and efficiency advantages of parallel databases [1]. 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. Parallely, it tries to achieve the performance of parallel databases by doing most of the query processing inside the database engine.

[20] http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.

[21] http://db.cs.yale.edu/hadoopdb/hadoopdb.html.

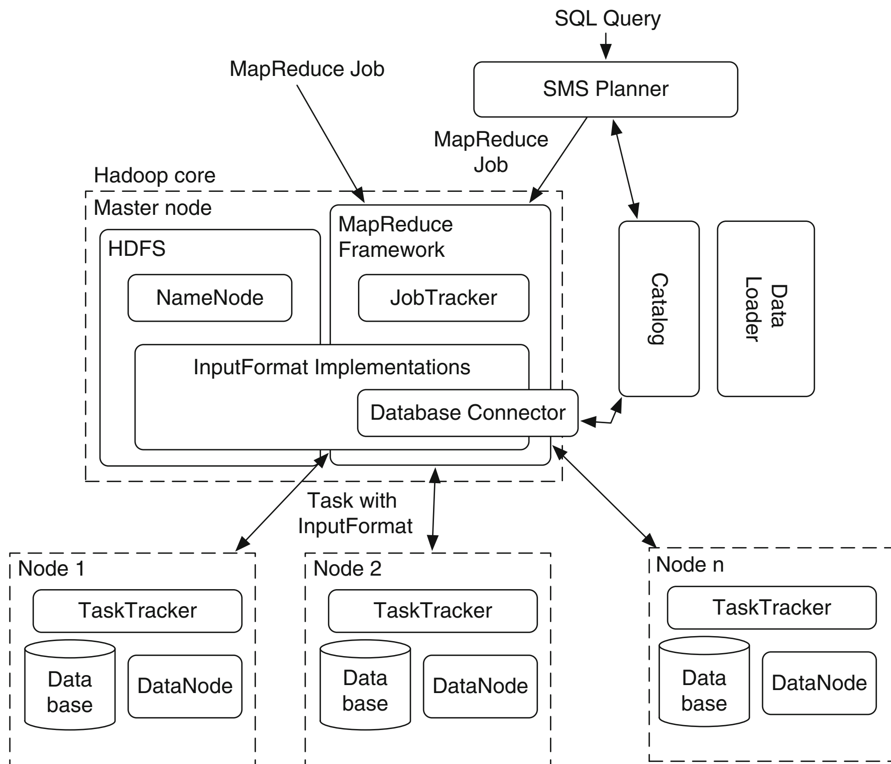

Fig. 1.20 The architecture of HadoopDB [1]

Figure 1.20 illustrates the architecture of HadoopDB, which consists of two layers: (1) a data storage layer or the Hadoop Distributed File System[22] (HDFS), and (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

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

and data partitioning properties. The *Data Loader* component is responsible for globally repartitioning data on a given partition key upon loading and breaking apart single node data into multiple smaller partitions or chunks. The *SMS planner* extends the HiveQL translator [40] and transforms SQL into MapReduce jobs that connect to tables stored as files in HDFS. Abouzeid et al. [2] have demonstrated HadoopDB in action running two different application types: (1) a semantic web application that provides biological data analysis of protein sequences, and (2) a classical business data warehouse.

Teradata [44] has recently started to follow the same approach of integrating Hadoop and parallel databases. It provides a fully parallel load utility to load Hadoop data to its datawarehouse store. Moreover, it provides a database connector for Hadoop, which allows MapReduce programs to directly access Teradata datawarehouses' data via JDBC drivers without the need of any external steps of exporting (from DBMS) and loading data to Hadoop. It also provides a *Table* user-defined function which can be called from any standard SQL query to retrieve Hadoop data directly from Hadoop nodes in parallel. This means that any relational tables can be joined with the Hadoop data that are retrieved by the Table UDF, and any complex business intelligence capability provided by Teradata's SQL engine can be applied to both Hadoop data and relational data. Hence, no extra steps of exporting/importing Hadoop data to/from the Teradata datawarehouse are required.

1.6 Case Studies

MapReduce-based systems are increasingly being used for large-scale data analysis. There are several reasons for this [28], such as:

- *The interface of MapReduce is simple yet expressive.* Although MapReduce only involves two functions, map and reduce, a number of data analytical tasks, including traditional SQL query, data mining, machine learning and graph processing, can be expressed with a set of MapReduce jobs.
- *MapReduce is flexible.* MapReduce is designed to be independent of storage systems and is able to analyze various kinds of data, structured and unstructured.
- *MapReduce is scalable*. An installation of MapReduce can run over thousands of nodes on a shared-nothing cluster, while keeping to provide fine-grain fault tolerance whereby only tasks on failed nodes need to be restarted.

The above-mentioned advantages have triggered several research efforts that aim at applying the MapReduce framework for solving challenging data processing problems on large scale datasets in a wide spectrum of domains. For example, *Mahout*[23] is an apache project which is designed with the aim of building scalable machine learning libraries using the MapReduce framework. *Ricardo* [15] is a

[23] http://mahout.apache.org/.

scalable platform for applying sophisticated statistical methods over huge data repositories. It is designed to facilitate the *trading* between *R* (a famous statistical software) and Hadoop, where each trading partner performs the tasks that it does best. In particular, this trading is performed in a way that *R* sends aggregation-processing queries to Hadoop, while Hadoop sends aggregated data to *R* for advanced statistical processing or visualization.

MapDupReducer [43] is a MapReduce-based system which has been developed for supporting the problem of near duplicate detection over massive datasets. Vernica et al. [42] have proposed an approach to efficiently perform set-similarity joins in parallel using the MapReduce framework. In particular, they have proposed a 3-stage approach for end-to-end set-similarity joins. The approach takes as input a set of records and outputs a set of joined records based on a set-similarity condition. It partitions the data across nodes in order to balance the workload and minimize the need for replication. Morales et al. [21] have presented two matching algorithms, *GreedyMR* and *StackMR*, which are geared for the MapReduce paradigm and aim to distribute content from information suppliers to information consumers on social media applications. In particular, they seek to maximize the overall relevance of the matched content from suppliers to consumers, while regulating the overall activity.

Surfer [14] is a large scale graph processing engine which is designed to execute in the cloud. Surfer provides two basic primitives for programmers: *MapReduce* and *propagation*. In this engine, MapReduce processes different key-value pairs in parallel, and propagation is an iterative computational pattern that transfers information along the edges from a vertex to its neighbors in the graph. In principle, these two primitives are complementary in graph processing where MapReduce is suitable for processing flat data structures (e.g., vertex-oriented tasks), while propagation is optimized for edge-oriented tasks on partitioned graphs.

Lattanzi et al. [30] have presented an approach for solving graph problems using the MapReduce framework. In particular, they present parallelized algorithms for minimum spanning trees, maximal matchings, approximate weighted matchings, approximate vertex and edge covers and minimum cuts. Cary et al. [12] presented an approach for applying the MapReduce model in the domain of spatial data management. In particular, they focus on the bulk-construction of R-Trees and aerial image quality computation, which involves vector and raster data.

Abouzeid et al. [2] have demonstrated that *HadoopDB* in conjunction with a column-oriented database can provide a promising solution for supporting efficient and scalable semantic web applications. Ravindra et al. [36] have presented an approach for parallelizing the processing of analytical queries on RDF graph models. In particular, they extended the function library of *Pig Latin* to include functions that aid in operator-coalescing and look-ahead processing to reduce the I/O costs that arise from repeated processing and materialization of intermediate results.

1.7 Discussion and Conclusions

MapReduce has emerged as a popular way to harness the power of large clusters of computers. Currently, MapReduce serves as a platform for a considerable amount of massive data analysis. 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. Gu and Grossman [25] have reported the following important lessons, which they have learned from their experiments with the MapReduce framework:

- *The importance of data locality*. Locality is a key factor, especially when relying on inexpensive commodity hardware.
- *Load balancing and the importance of identifying hot spots*. With poor load balancing, the entire system can be waiting for a single node. Thus, it is important to eliminate any "hot spots", which can be caused by data access (accessing data from a single node) or network I/O (transferring data into or out of a single node).
- *Fault tolerance comes with a price*. In some cases, fault tolerance introduces extra overhead in order to replicate the intermediate results. For example, in the cases of running on small to medium sized clusters, it might be reasonable to favor performance and re-run any failed intermediate task when necessary.
- *Streams are important*. Streaming is important in order to reduce the total running time of MapReduce jobs.

Jiang et al. [28] have conducted an in-depth performance study of MapReduce using its open source implementation, Hadoop. As an outcome of this study, they identified some factors that can have significant performance effect on the MapReduce framework. These factors are described as follows:

- Although MapReduce is independent of the underline storage system, it still requires the storage system to provide efficient I/O modes for scanning data. The experiments of the study on HDFS show that direct I/O outperforms streaming I/O by 10–15 %.
- MapReduce can utilize three kinds of indices, namely range-indices, block-level indices and database indexed tables, in a straightforward way. The experiments of the study show that the range-index improves the performance of MapReduce by a factor of 2 in the selection task and a factor of 10 in the join task when selectivity is high.
- There are two kinds of decoders for parsing the input records: mutable decoders and immutable decoders. The study claims that only immutable decoders introduce performance bottleneck. To handle database-like workloads, MapReduce users should strictly use mutable decoders. A mutable decoder is faster than an immutable decoder by a factor of 10, and improves the performance of selection by a factor of 2. Using a mutable decoder, even parsing the text record is efficient.
- Map-side sorting exerts negative performance effect on large aggregation tasks, which require nontrivial key comparisons and produce millions of groups.

Therefore, fingerprinting-based sort can be used to significantly improve the performance of MapReduce on such aggregation tasks. The experiments show that fingerprinting-based sort outperforms direct sort by a factor of 4–5, and improves overall performance of the job by 20–25%.
- The scheduling strategy affects the performance of MapReduce, as it can be sensitive to the processing speed of slave nodes, and slows down the execution time of the entire job by 25–35% [48].

The experiments of the study show that with proper engineering for these factors, the performance of MapReduce can be improved by a factor of 2.5–3.5, and approaches the performance of Parallel Databases.

In general, to run a single program in a MapReduce framework, a number of tuning parameters (e.g. memory allocation, concurrency, I/O optimization, network bandwidth usage) have to be set by users or system administrators. In practice, users may often run into performance problems because they do not know how to set these parameters. In addition, as MapReduce is a relatively new technology, it is not easy to find qualified administrators. Babu [6] has proposed some techniques to *automate* the setting of tuning parameters for MapReduce programs. The aim of these techniques is to provide good out-of-the-box performance for ad hoc MapReduce programs that run on large datasets. Babu suggested the following research agenda to automatically configure the parameters for MapReduce jobs:

- There is a need to conduct a comprehensive empirical study with a representative class of MapReduce programs and different cluster configurations to understand (and potentially model) parameter impacts, interactions, and response surfaces.
- Developing cost models that are useful to recommend good parameter settings for MapReduce job configuration parameters.
- Tune the performance of a MapReduce program that is run repeatedly (e.g., for daily report generation) and whose current performance is unsatisfactory.
- Developing mechanisms that can automatically generate an execution plan, which is composed of one or more MapReduce jobs for a higher-level operation, like join.

The cluster-level energy management of the MapReduce framework is another interesting research direction. Lang and Patel [29] have investigated the approach to power down (and power up) MR nodes in order to save energy in periods of low utilization. In particular, they compared between two strategies for MR energy management: (1) Covering Set (CS) strategy that keeps only a small fraction of the nodes powered up during periods of low utilization, and (2) All-In Strategy (AIS) that uses all the nodes in the cluster to run a workload and then powers down the entire cluster. The comparison shows that there are two crucial factors that affect the effectiveness of these two methods: (1) the computational complexity of the workload, and (2) the time taken to transition nodes to and from a low power (deep hibernation) state to a high performance state. The comparison evaluation also shows that *CS* is more effective than *AIS* only when the computational complexity of the workload is low (e.g., linear), and that the time it takes for the hardware to

transition a node to and from a low power state is a relatively large fraction of the overall workload time (i.e., the workload execution time is small). In all other cases, the *AIS* shows better performance over *CS* in terms of energy savings and response time performance.

We believe that this survey of the MapReduce family of approaches would be useful for the future development of MapReduce-based data processing systems. In addition, we are convinced that there is still room for further optimization and advancement in several directions on the spectrum of the MapReduce framework that is still required to bring forward the vision of providing large scale data analysis as a commodity for novice end-users.

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Rasin, D.A., Silberschatz, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. PVLDB **2**(1), 922–933 (2009)
2. Abouzeid, A., Bajda-Pawlikowski, K., Huang, J., Abadi, D., Silberschatz, A.: HadoopDB in action: building real world applications. In: SIGMOD, Indianapolis, 2010, pp. 1111–1114
3. Afrati, F., Ullman, J.: Optimizing joins in a map-reduce environment. In: EDBT, Lausanne, 2010, pp. 99–110
4. Alvaro, P., Hellerstein, J., Elmeleegy, K., Condie, T., Conway, N., Sears, R.: MapReduce online. In: NSDI, San Jose, 2010
5. Armbrust, M., Fox, A., Rean, G., Joseph, A., Katz, R., Konwinski, A., Gunho, L., David, P., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: a Berkeley view of cloud computing, Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Tech. Rep. UCB/EECS, vol. 28, 2009
6. Babu, S.: Towards automatic optimization of MapReduce programs. In: SoCC, Indianapolis, 2010, pp. 137–142
7. Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Paulson, E.: HadoopDB in action: efficient processing of data warehousing queries in a split execution environment. In: SIGMOD, Athens, 2011, pp. 1165–1176
8. Bell, G., Gray, J., Szalay, A.: Petascale computational systems. IEEE Comput. **39**(1), 110–112 (2006)
9. Beyer, K., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M., Kanne, C., Ozcan, F., Shekita, E.: Jaql: a scripting language for large scale semistructured data analysis. PVLDB **4**(11), 1272–1283 (2011)
10. Blanas, S., Patel, J., Ercegovac, V., Rao, J., Shekita, E., Tian, Y.: A comparison of join algorithms for log processing in MapReduce. In: SIGMOD, Indianapolis, 2010, pp. 975–986
11. Bu, Y., Howe, B., Balazinska, M., Ernst, M.: HaLoop: efficient iterative data processing on large clusters. PVLDB **3**(1), 285–296 (2010)
12. Cary, A., Sun, Z., Hristidis, V., Rishe, N.: Experiences on processing spatial data with MapReduce. In: SSDBM, New Orleans, 2009, pp. 302–319
13. Chaiken, R., Jenkins, B., Larson, P., Ramsey, B., Shakib, D., Weaver, S., Zhou, J.: SCOPE: easy and efficient parallel processing of massive data sets. PVLDB **1**(2), 1265–1276 (2008)
14. Chen, R., Weng, X., He, B., Yang, M.: Large graph processing in the cloud. In: SIGMOD, Indianapolis, 2010, pp. 1123–1126
15. Das, S., Sismanis, Y., Beyer, K., Gemulla, R., Haas, P., McPherson, J.: Ricardo: integrating R and Hadoop. In: SIGMOD, Indianapolis, 2010, pp. 987–998

16. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI, San Francisco, 2004, pp. 137–150
17. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
18. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)
19. Dittrich, J., Quiane-Ruiz, J., Jindal, A., Kargin, Y., Setty, V., Schad, J.: Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). PVLDB **3**(1), 518–529 (2010)
20. Eltabakh, M., Tian, Y., Ozcan, F., Gemulla, R., Krettek, A., McPherson, J.: CoHadoop: flexible data placement and its exploitation in Hadoop. PVLDB **4**(9), 575–585 (2011)
21. Francisci Morales, G., Gionis, A., Sozio, M.: Social content matching in MapReduce. PVLDB **4**(7), 460–469 (2011)
22. Friedman, E., Pawlowski, P., Cieslewicz, J.: SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions. PVLDB **2**(2), 1402–1413 (2009)
23. Gates, A., Natkovich, O., Chopra, S., Kamath, P., Narayanam, S., Olston, C., Reed, B., Srinivasan, S., Srivastava, U.: Building a highlevel data ow system on top of MapReduce: the pig experience. PVLDB **2**(2), 1414–1425 (2009)
24. Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. In: SOSP, Bolton Landing, 2003, pp. 29–43
25. Gu, Y., Grossman, R.: Lessons learned from a year's worth of benchmarks of large data clouds. In: SC-MTAGS, Portland, 2009
26. Hey, T., Tansly, S., Tolle, K. (eds.): The Fourth Paradigm: Data-Intensive Scientific Discovery. Microsoft Research, Redmond (2009)
27. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: EuroSys, Lisbon, 2007, pp. 59–72
28. Jiang, D., Chin Ooi, B., Shi, L., Wu, S.: The performance of MapReduce: an in-depth study. PVLDB **3**(1), 472–483 (2010)
29. Lang, W., Patel, J.: Energy management for MapReduce clusters. PVLDB **3**(1), 129–139 (2010)
30. Lattanzi, S., Moseley, B., Suri, S., Vassilvitskii, S.: Filtering: a method for solving graph problems in MapReduce. In: SPAA, San Jose, 2011, pp. 85–94
31. Murray, D., Hand, S.: Scripting the cloud with Skywriting. In: HotCloud, USENIX Workshop, Boston, 2010
32. Nykiel, T., Potamias, M., Mishra, C., Kollios, G., Koudas, N.: MRShare: sharing across multiple queries in MapReduce. PVLDB **3**(1), 494–505 (2010)
33. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD, Vancouver, 2008, pp. 1099–1110
34. Pavlo, A., Paulson, E., Rasin, A., Abadi, D., DeWitt, D., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: SIGMOD, Providence, 2009, pp. 165–178
35. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: parallel analysis with Sawzall. Sci. Program. **13**(4), 277–298 (2005)
36. Ravindra, P., Deshpande, V., Anyanwu, K.: Towards scalable RDF graph analytics on MapReduce. In: MDAC, Raleigh, 2010
37. Sakr, S., Liu, A., Batista, D., Alomari, M.: Hive – a survey of large scale data management approaches in cloud environments. IEEE Commun. Surv. Tutor. **13**(3), 311–336 (2011)
38. Stonebraker, M.: The case for shared nothing. IEEE Database Eng. Bull. **9**(1), 4–9 (1986)
39. Stonebraker, M., Abadi, D., DeWitt, D., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: MapReduce and parallel DBMSs: friends or foes? Commun. ACM **53**(1), 64–71 (2010)
40. Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive – a warehousing solution over a map-reduce framework. PVLDB **2**(2), 1626–1629 (2009)

41. Thusoo, A., Sarma, J., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive – a petabyte scale data warehouse using Hadoop. In: ICDE, Long Beach, 2010, pp. 996–1005
42. Vernica, R., Carey, M., Li, C.: Efficient parallel set-similarity joins using MapReduce. In: SIGMOD, Indianapolis, 2010, pp. 495–506
43. Wang, C., Wang, J., Lin, X., Wang, W., Wang, H., Li, H., Tian, W., Xu, J., Li, R.: MapDupReducer: detecting near duplicates over massive datasets. In: SIGMOD, Indianapolis, 2010, pp. 1119–1122
44. Xu, Y., Kostamaa, P., Gao, L.: Integrating Hadoop and parallel DBMS. In: SIGMOD, Indianapolis, 2010, pp. 969–974
45. Yang, H., Parker, D.: Traverse: simplified indexing on large map-reduce-merge clusters. In: DASFAA, Brisbane, 2009, pp. 308–322
46. Yang, H., Dasdan, A., Hsiao, R., Parker, D.: Map-reduce-merge: simplified relational data processing on large clusters. In: SIGMOD, Beijing, 2007, pp. 1029–1040
47. Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P., Currey, J.: DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In: OSDI, San Diego, 2008, pp. 1–14
48. Zaharia, M., Konwinski, A., Joseph, A., Katz, R., Stoica, I.: Improving MapReduce performance in heterogeneous environments. In: OSDI, San Diego, 2008, pp. 29–42
49. Zhou, J., Larson, P., Chaiken, R.: Incorporating partitioning and parallel plans into the SCOPE optimizer. In: ICDE, Long Beach, 2010, pp. 1060–1071