Foundations and Trends® in Databases
Vol. 5, No. 1 (2012) 1–104

DOI: 10.1561/1900000036

now
the essence of knowledge

Massively Parallel Databases and MapReduce Systems

Shivnath Babu
Duke University
shivnath@cs.duke.edu

Herodotos Herodotou
Microsoft Research
herohero@microsoft.com

Contents

Abstract

Timely and cost-effective analytics over "big data" has emerged as a key ingredient for success in many businesses, scientific and engineering disciplines, and government endeavors. Web clicks, social media, scientific experiments, and datacenter monitoring are among data sources that generate vast amounts of raw data every day. The need to convert this raw data into useful information has spawned considerable innovation in systems for large-scale data analytics, especially over the last decade. This monograph covers the design principles and core features of systems for analyzing very large datasets using massively-parallel computation and storage techniques on large clusters of nodes. We first discuss how the requirements of data analytics have evolved since the early work on parallel database systems. We then describe some of the major technological innovations that have each spawned a distinct category of systems for data analytics. Each unique system category is described along a number of dimensions including data model and query interface, storage layer, execution engine, query optimization, scheduling, resource management, and fault tolerance. We conclude with a summary of present trends in large-scale data analytics.

S. Babu and H. Herodotou. *Massively Parallel Databases and MapReduce Systems.* Foundations and Trends® in Databases, vol. 5, no. 1, pp. 1–104, 2012.

DOI: 10.1561/1900000036.

4

MapReduce Systems

MapReduce is a relatively young framework—both a programming model and an associated run-time system—for large-scale data processing [73]. *Hadoop* is the most popular open-source implementation of a MapReduce framework that follows the design laid out in the original paper [72]. A number of companies use Hadoop in production deployments for applications such as Web indexing, data mining, report generation, log file analysis, machine learning, financial analysis, scientific simulation, and bioinformatics research.

Even though the MapReduce programming model is highly flexible, it has been found to be too low-level for routine use by practitioners such as data analysts, statisticians, and scientists [159, 189]. As a result, the MapReduce framework has evolved rapidly over the past few years into a *MapReduce stack* that includes a number of higher-level layers added over the core MapReduce engine. Prominent examples of these higher-level layers include *Hive* (with an SQL-like declarative interface), *Pig* (with an interface that mixes declarative and procedural elements), *Cascading* (with a Java interface for specifying workflows), *Cascalog* (with a Datalog-inspired interface), and *BigSheets* (with a spreadsheet interface). The typical Hadoop stack is shown in Figure 4.1.

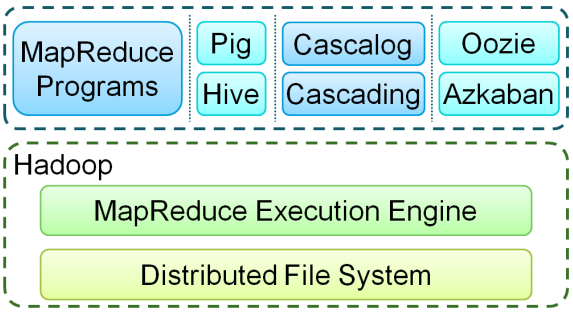

Figure 4.1: Typical Hadoop software stack.

4.1 Data Model and Interfaces

MapReduce systems typically process data directly from files, permitting data to be in any arbitrary format. Hence, MapReduce systems are capable of processing unstructured, semi-structured, and structured data alike.

The MapReduce programming model consists of two functions: $map(k_1, v_1)$ and $reduce(k_2, list(v_2))$ [192]. Users can implement their own processing logic by specifying a customized *map*() and a *reduce*() function written in a general-purpose language like Java or Python. The $map(k_1, v_1)$ function is invoked for every *key-value* pair $\langle k_1, v_1 \rangle$ in the input data to output zero or more key-value pairs of the form $\langle k_2, v_2 \rangle$. The $reduce(k_2, list(v_2))$ function is invoked for every unique key k_2 and corresponding values $list(v_2)$ in the map output. $reduce(k_2, list(v_2))$ outputs zero or more key-value pairs of the form $\langle k_3, v_3 \rangle$. The MapReduce programming model also allows other functions such as (i) $partition(k_2)$, for controlling how the map output key-value pairs are partitioned among the reduce tasks, and (ii) $combine(k_2, list(v_2))$, for performing partial aggregation on the map side. The keys k_1, k_2, and k_3 as well as the values v_1, v_2, and v_3 can be of different and arbitrary types.

A given MapReduce program may be expressed in one among a variety of programming languages like Java, C++, Python, or Ruby; and then connected to form a workflow using a workflow scheduler such as Oozie [161]. Alternatively, the MapReduce jobs can be generated

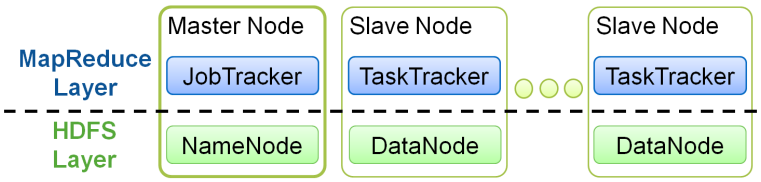

Figure 4.2: Hadoop architecture.

automatically using compilers for higher-level languages like Pig Latin [159], HiveQL [189], JAQL [36], and Cascading [52].

4.2 Storage Layer

The storage layer of a typical MapReduce cluster is an independent distributed file system. Typical Hadoop deployments use the *Hadoop Distributed File System (HDFS)* running on the cluster's compute nodes [176]. Alternatively, a Hadoop cluster can process data from other file systems like the MapR File System [149], CloudStore (previously Kosmos File System) [127], Amazon Simple Storage Service (S3) [13], and Windows Azure Blob Storage [50].

HDFS is designed to be resilient to hardware failures and focuses more on batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access. An HDFS cluster employs a master-slave architecture consisting of a single *NameNode* (the master) and multiple *DataNodes* (the slaves), usually one per node in the cluster (see Figure 4.2). The NameNode manages the file system namespace and regulates access to files by clients, whereas the DataNodes are responsible for serving read and write requests from the file system's clients. HDFS is designed to reliably store very large files across machines in a large cluster. Internally, a file is split into one or more blocks that are replicated for fault tolerance and stored in a set of DataNodes.

The modularity of the storage architecture enables higher-level systems to introduce their own features or extensions on top of the distributed file system. For example, Hive organizes and stores the data into partitioned tables [189]. Hive tables are analogous to ta-

bles in relational databases and are represented using HDFS directories. Partitions are then created using subdirectories whereas the actual data is stored in files. Hive also includes a system catalog—called *Metastore*—containing schema and statistics, which are useful in data exploration and query optimization. Hive's Metastore inspired the creation of HCatalog, a new table and storage management service for data created using Hadoop [24]. HCatalog provides a unified table abstraction and interoperability across data processing tools such as MapReduce, Pig, and Hive.

Indexing in HDFS: Hadoop++ provides indexing functionality for data stored in HDFS by means of an approach called *Trojan Indexes* [80]. This approach is based on user-defined functions so no changes are required to the underlying Hadoop system. The indexing information is stored as additional metadata in the HDFS blocks that are read by map tasks. The information is added when tables are written to HDFS so that no overhead is caused during query processing.

Index creation in Hadoop++ can increase the data loading time. This problem is addressed by HAIL [81] which also improves query processing speeds over Hadoop++. HAIL creates indexes during the I/O-bound phases of writing to HDFS so that it consumes CPU cycles that are otherwise wasted. For fault tolerance purposes, HDFS maintains k replicas for every HDFS block. ($k = 3$ by default.) HAIL builds a different clustered index in each replica. The most suitable index for a query is selected at run-time, and the corresponding replica of the blocks are read by the map tasks in HAIL.

Data collocation: A serious limitation of MapReduce systems is that HDFS does not support the ability to collocate data. Because of this limitation, query processing systems on top of MapReduce—e.g., Hive, Pig, and JAQL—cannot support collocated join operators. Recall from Chapter 2 that collocated joins are one of the most efficient ways to do large joins in parallel database systems. Hadoop++ and CoHadoop [84] provide two different techniques to collocate data in MapReduce systems.

Hadoop++ uses the same "trojan" approach as in trojan indexes in order to co-partition and collocate data at load time [80]. Thus,

blocks of HDFS can now contain data from multiple tables. With this approach, joins can be processed at the map side rather than at the reduce side. Map-side joins avoid the overhead of sorting and shuffling data.

CoHadoop adds a file-locator attribute to HDFS files and implements a file layout policy such that all files with the same locator are placed on the same set of nodes. Using this feature, CoHadoop can collocate any related pair of files, e.g., every pair of joining partitions across two tables that are both hash-partitioned on the join key; or, a partition and an index on that partition. CoHadoop can then run joins very efficiently using a map-side join operator that behaves like the collocated join operator in parallel database systems. CoHadoop relies on applications to set the locator attributes for the files that they create.

Data layouts: It is also possible to implement columnar data layouts in HDFS. Systems like *Llama* [140] and *CIF* [89] use a pure column-oriented design while *Cheetah* [64], Hadoop++ [80], and *RCFile* [105] use a hybrid row-column design based on *PAX* [10]. Llama partitions attributes into vertical groups like the projections in C-Store and Vertica (recall §3.2.1). Each vertical group is sorted based on one of its component attributes. Each column is stored in a separate HDFS file, which enables each column to be accessed independently to reduce read I/O costs, but may incur run-time costs for tuple reconstruction.

CIF uses a similar design of storing columns in separate files, but its design is different from Llama in many ways. First, CIF partitions the table horizontally and stores each horizontal partition in a separate HDFS directory for independent access in map tasks. Second, CIF uses an extension of HDFS to enable collocation of columns corresponding to the same tuple on the same node. Third, CIF supports some late materialization techniques to reduce tuple reconstruction costs [89].

Cheetah, Hadoop++, and RCFile use a layout where a set of tuples is stored per HDFS block, but a columnar format is used within the HDFS block. Since HDFS guarantees that all the bytes of an HDFS block will be stored on a single node, it is guaranteed that tuple reconstruction will not require data transfer over the network. The intra-block data layouts used by these systems differ in how they use com-

pression, how they treat replicas of the same block, etc. For example, Hadoop++ can use different layouts in different replicas, and choose the best layout at query processing time. Queries that require a large fraction of the columns should use a row-oriented layout, while queries that access fewer columns should use a column-oriented layout [122].

Almost all the above data layouts are inspired by similar data layouts in classic parallel database systems and columnar database systems. *HadoopDB* [5] takes this concept to the extreme by installing a centralized database system on each node of the cluster, and using Hadoop primarily as the engine to schedule query execution plans as well as to provide fine-grained fault tolerance. The additional storage system provided by the databases gives HadoopDB the ability to overcome limitations of HDFS such as lack of collocation and indexing. HadoopDB introduces some advanced partitioning capabilities such as reference-based partitioning which enable multi-way joins to be performed in a collocated fashion [30]. The HadoopDB architecture is further discussed in §4.3.

HDFS alternatives: A number of new distributed file systems are now viable alternatives to HDFS and offer full compatibility with Hadoop MapReduce. The MapR File System [149] and Ceph [191] have similar architectures to HDFS but both offer a distributed metadata service as opposed to the centralized NameNode on HDFS. In MapR, metadata is sharded across the cluster and collocated with the data blocks, whereas Ceph uses dedicated metadata servers with dynamic subtree partitioning to avoid metadata access hot spots. In addition, MapR allows for mutable data access and is NFS mountable. The Quantcast File System (QFS) [162], which evolved from the Kosmos File System (KFS) [127], employs erasure coding rather than replication as its fault tolerance mechanism. Erasure coding enables QFS to not only reduce the amount of storage but to also accelerate large sequential write patterns common to MapReduce workloads.

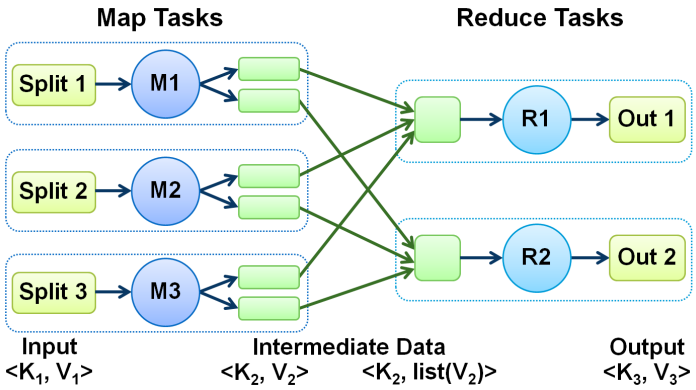

Figure 4.3: Execution of a MapReduce job.

4.3 Execution Engine

MapReduce execution engines represent a new data-processing framework that has emerged in recent years to deal with data at massive scale [72]. Users specify computations over large datasets in terms of Map and Reduce functions, and the underlying run-time system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disk bandwidth.

As shown in Figure 4.2, a Hadoop MapReduce cluster employs a master-slave architecture where one master component (called *JobTracker*) manages a number of slave components (called *TaskTrackers*). Figure 4.3 shows how a MapReduce job is executed on the cluster. Hadoop launches a MapReduce job by first splitting (logically) the input dataset into data *splits*. Each data split is then scheduled to one TaskTracker node and is processed by a map task. A *Task Scheduler* is responsible for scheduling the execution of map tasks while taking data locality into account. Each TaskTracker has a predefined number of task execution *slots* for running map (reduce) tasks. If the job will execute more map (reduce) tasks than there are slots, then the map (reduce) tasks will run in multiple *waves*. When map tasks complete, the run-time system groups all intermediate key-value pairs using an

external sort-merge algorithm. The intermediate data is then *shuffled* (i.e., transferred) to the TaskTrackers scheduled to run the reduce tasks. Finally, the reduce tasks will process the intermediate data to produce the results of the job.

Higher-level systems like Pig [94] and Hive [189] use the underlying MapReduce execution engine in similar ways to process the data. In particular, both Pig and Hive will compile the respective Pig Latin and HiveQL queries into logical plans, which consist of a tree of logical operators. The logical operators are then converted into physical operators, which in turn are packed into map and reduce tasks for execution. A typical query will result in a directed acyclic graph (DAG) of multiple MapReduce jobs that are executed on the cluster.

The execution of Pig Latin and HiveQL queries as MapReduce jobs results in Pig and Hive being well suited for batch processing of data, similar to Hadoop. These systems thrive when performing long sequential scans in parallel, trying to utilize the entire cluster to the fullest. In addition, both Pig and Hive are read-based, and therefore not appropriate for online transaction processing which typically involves a high percentage of random write operations. Finally, since MapReduce is the basic unit of execution, certain optimization opportunities like better join processing can be missed (discussed further in §4.4).

4.3.1 Alternative MapReduce Execution Engines

As discussed earlier in §4.2, HadoopDB is a hybrid system that combines the best features of parallel database systems and MapReduce systems [5]. HadoopDB runs a centralized database system on each node and uses Hadoop primarily as the execution plan scheduling and fault-tolerance layer. HadoopDB introduced the concept of *split query execution* where a query submitted by a user or application will be converted into an execution plan where some parts of the plan would run as queries in the database and other parts would run as map and reduce tasks in Hadoop [30]. The best such splitting of work will be identified during plan generation. For example, if two joining tables are stored partitioned and collocated, then HadoopDB can perform an efficient collocated join like in parallel database systems.

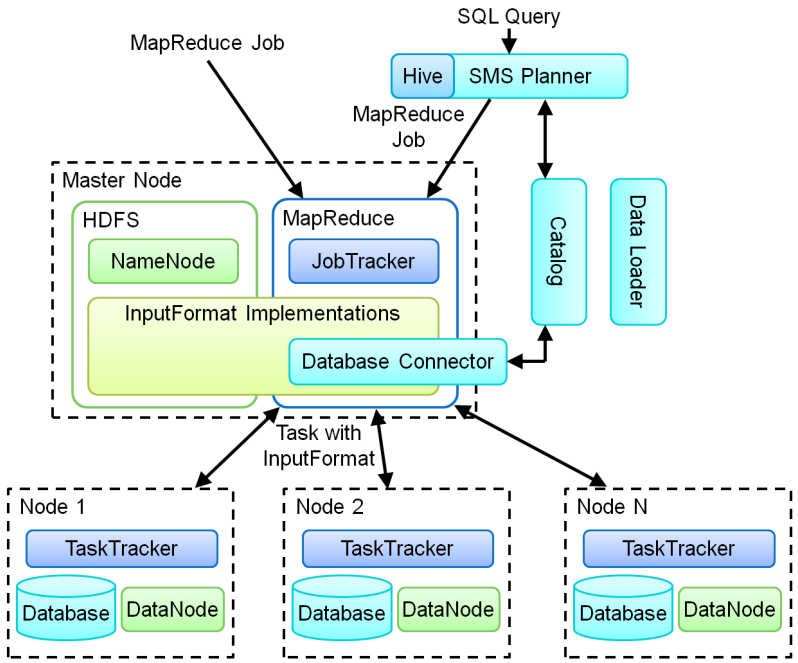

Figure 4.4: Architecture of the HadoopDB system, which has a hybrid design that combines MapReduce systems and centralized databases.

Figure 4.4 shows the architecture of HadoopDB. The database connector is an interface between the TaskTrackers in Hadoop and the individual database systems. The database connector can connect to a database, execute a SQL query, and return the query result in the form of key-value pairs. Metadata about the databases is stored in the system catalog. The catalog maintains system metadata such as connection parameters, schema and statistics of the tables stored, locations of replicas, and data partitioning properties.

The SMS (SQL to MapReduce to SQL) planner extends Hive and produces split-execution query plans that can exploit features provided by the available database systems. The data loader globally repartitions tables based on a partition key, breaks down single-node partitions further into smaller partitions, and bulk loads the single-node databases with these smaller partitions.

Clydesdale is a system that was built to demonstrate that a simple architecture that leverages existing techniques from parallel database systems can provide substantial performance benefits for query processing in MapReduce systems [123]. Clydesdale is aimed at workloads where the data fits a star schema. The fact table is stored on HDFS using CIF [89]. Copies of the dimension tables are also stored in HDFS, but are replicated to the local storage on each node in the cluster.

SQL queries submitted to Clydesdale are converted into a query plan composed of MapReduce jobs. Join processing is done in the map phase such that map tasks apply predicates to the dimension tables and build hash tables in the setup phase. Then, the map tasks join the tuples in the fact table by probing the hash tables. The reduce phase is responsible for grouping and aggregation. Clydesdale draws on several strategies for performance improvement: careful use of multi-core parallelism, employing a tailored star-join plan instead of joining tables in a pairwise manner, and columnar storage. All these techniques together give considerable improvement over processing the queries directly on a MapReduce system.

Sailfish [170] is an alternative MapReduce framework for large scale data processing whose design is centered around aggregating intermediate data, i.e., data produced by map tasks and consumed later by reduce tasks. In particular, the output of map tasks (which consists of key/value pairs) is first partitioned by key and then aggregated on a per-partition basis using a new file system abstraction, called *I-files* (Intermediate data files). I-files support batching of data written by multiple writers and read by multiple readers. This intermediate data is sorted and augmented with an index to support key-based retrieval. Based on the distribution of keys across the I-files, the number of reduce tasks as well as the key-range assignments to tasks can be determined dynamically in a data-dependent manner.

4.4 Query Optimization

Being a much newer technology, MapReduce engines significantly lack principled optimization techniques compared to database systems.

Hence, the MapReduce stack (see Figure 4.1) can be poorer in performance compared to a database system running on the same amount of cluster resources [90, 166]. In particular, the reliance on manual tuning and absence of cost-based optimization can result in missed opportunities for better join processing or partitioning. A number of ongoing efforts are addressing this issue through optimization opportunities arising at different levels of the MapReduce stack.

For higher levels of the MapReduce stack that have access to declarative semantics, many optimization techniques inspired by database query optimization and workload tuning have been proposed. Hive and Pig employ *rule-based* approaches for a variety of optimizations such as filter and projection pushdown, shared scans of input datasets across multiple operators from the same or different analysis tasks [157], reducing the number of MapReduce jobs in a workflow [137], and handling data skew in sorts and joins. The *AQUA* system supports System-R-style join ordering [193]. Improved data layouts [122, 140] and indexing techniques [80, 81] inspired by database storage have also been proposed, as discussed in §4.2.

Lower levels of the MapReduce stack deal with workflows of MapReduce jobs. A MapReduce job may contain *black-box* map and reduce functions expressed in programming languages like Java, Python, and R. Many heavy users of MapReduce, ranging from large companies like Facebook and Yahoo! to small startups, have observed that MapReduce jobs often contain black-box UDFs to implement complex logic like statistical learning algorithms or entity extraction from unstructured data [146, 159]. One of the optimization techniques proposed for this level—exemplified by *HadoopToSQL* [121] and *Manimal* [49]—does static code analysis of MapReduce programs to extract declarative constructs like filters and projections. These constructs are then used for database-style optimization such as projection pushdown, column-based compression, and use of indexes.

Finally, the performance of MapReduce jobs is directly affected by various configuration parameter settings like degree of parallelism and use of compression. Choosing such settings for good job performance is a nontrivial problem and a heavy burden on users [27]. Starfish has

introduced a cost-based optimization framework for MapReduce systems for determining configuration parameter settings as well as the cluster resources to meet desired requirements on execution time and cost for a given analytics workload [110]. Starfish's approach is based on: (i) collecting monitoring information in order to learn the run-time behavior of workloads (profiling), (ii) deploying appropriate models to predict the impact of hypothetical tuning choices on workload behavior, and (iii) using efficient search strategies to find tuning choices that give good workload performance [107, 109].

4.5 Scheduling

The primary goal of scheduling in MapReduce is to maximize data-locality; that is, to schedule the MapReduce tasks to execute on nodes where data reside or as close to those nodes as possible. The original scheduling algorithm in Hadoop was integrated within the JobTracker and was called FIFO (First In, First Out). In FIFO scheduling, the JobTracker pulled jobs from a work queue in order of arrival and scheduled all the tasks from each job for execution on the cluster. This scheduler had no concept of priority or size of the job, but offered simplicity and efficiency.

As MapReduce evolved into a multi-tenant data-processing platform, more schedulers were created in an effort to maximize the workload throughput and cluster utilization. The two most prominent schedulers today are the *Fair Scheduler* [16] and the *Capacity Scheduler* [15], developed by Facebook and Yahoo!, respectively. The core idea behind the Fair Scheduler is to assign resources to jobs such that on average over time, each job gets an equal share of the available resources. Users may assign jobs to pools, with each pool allocated a guaranteed minimum number of Map and Reduce slots. Hence, this scheduler lets short jobs finish in reasonable time while not starving long jobs. The Capacity Scheduler shares similar principles with the Fair Scheduler, but focuses on enforcing cluster capacity sharing among users, rather than among jobs. In capacity scheduling, several queues are created, each with a configurable number of map and reduce slots. Queues that

contain jobs are given their configured capacity, while free capacity in a queue is shared among other queues. Within a queue, scheduling operates based on job priorities.

There exists an extensive amount of research work improving the scheduling policies in Hadoop [168]. Delay scheduling [198] is an extension to the Fair Scheduler that temporarily relaxes fairness to improve data locality by asking jobs to wait for a scheduling opportunity on a node with local data. Dynamic Proportional Share Scheduler [172] supports capacity distribution dynamically among concurrent users, allowing them to adjust the priority levels assigned to their jobs. However, this approach does not guarantee that a job will complete by a specific deadline. Deadline Constraint Scheduler [124] addresses the issue of deadlines but focuses more on increasing system utilization. Finally, Resource Aware Scheduler [196] considers resource availability on a more fine-grained basis to schedule jobs.

One potential issue with the Hadoop platform is that by dividing the tasks across multiple nodes, it is possible for a few slow nodes to rate-limit the rest of the job. To overcome this issue, Hadoop uses a process known as *speculative execution*. In particular, Hadoop schedules redundant copies of some tasks (typically towards the end of a job) for execution across several nodes that are not currently busy. Whichever copy of a task finishes first becomes the definitive copy and the other copies are abandoned.

As discussed earlier, a typical HiveQL or Pig Latin query is parsed, possibly optimized using a rule-based or cost-based approach, and converted into a *MapReduce workflow*. A MapReduce workflow is a directed acyclic graph (DAG) of multiple MapReduce jobs that read one or more input datasets and write one or more output datasets. The jobs in a workflow can exhibit *dataflow dependencies* because of producer-consumer relationships and hence, must be scheduled serially in order of their dependencies. Jobs without such dataflow dependencies can be scheduled to be run concurrently and are said to exhibit *cluster resource dependencies* [139]. The scheduling of jobs within a workflow is typically handled by the higher-level system itself (i.e., Hive and Pig).

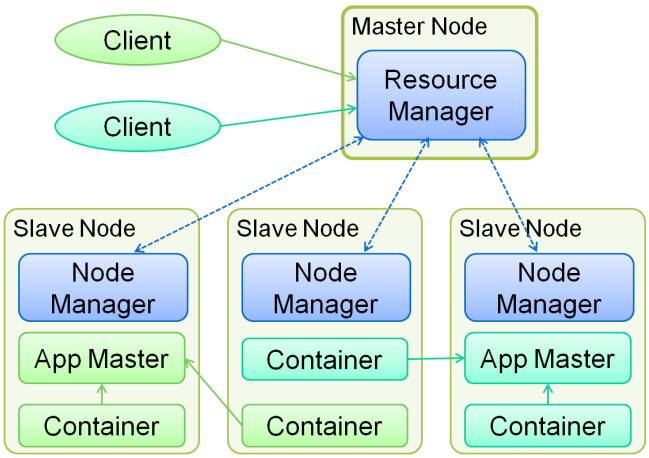

Figure 4.5: Hadoop NextGen (YARN) architecture.

4.6 Resource Management

In the original version of Hadoop, each node in a cluster is statically assigned a predefined number of map and reduce *slots* for running map and reduce tasks concurrently [192]. Hence, the allocation of cluster resources to jobs is done in the form of these slots. This static allocation of slots has the obvious drawback of lowered cluster utilization since slot requirements vary during the MapReduce job life cycle. Typically, there is a high demand for map slots when the job starts, whereas there is a high demand for reduce slots towards the end. However, the simplicity of this scheme simplified the resource management performed by the JobTracker, in addition to the scheduling decisions.

Overall, the Job Tracker in Hadoop has the dual role of managing the cluster resources as well as scheduling and monitoring MapReduce jobs. *Hadoop NextGen* (also known as *MapReduce 2.0* or *YARN*) [22] separates the above two functionalities into two separate entities: a global *ResourceManager* is responsible for allocating resources to running applications, whereas a per-application *ApplicationMaster* manages the application's life-cycle (see Figure 4.5). There is also a per-machine *NodeManager* that manages the user processes on that node.

The ApplicationMaster is a framework-specific library that negotiates resources from the ResourceManager and works with the NodeManager(s) to execute and monitor the tasks. In the YARN design, MapReduce is just one application framework; the design permits building and deploying distributed applications using other frameworks as well.

The resource allocation model in YARN addresses the static allocation deficiencies of the previous Hadoop versions by introducing the notion of *resource containers*. A container represents a specification of node resources—called *attributes*—such as CPU, memory, disk bandwidth, and network bandwidth. In this model, only a minimum and a maximum for each attribute are defined, and ApplicationMasters can request containers with attribute values as multiples of the minimum. Hence, different ApplicationMasters have the ability to request different container sizes at different times, giving rise to new research challenges on how to efficiently and effectively allocate resources among them.

Hadoop also supports dynamic node addition as well as decommissioning of failed or surplus nodes. When a node is added in the cluster, the TaskTracker will notify the JobTracker of its presence and the JobTracker can immediately start scheduling tasks on the new node. On the other hand, when a node is removed from the cluster, the JobTracker will stop receiving heartbeats from the corresponding TaskTracker and after a small period of time, it will stop scheduling tasks on that node.

Even though newly-added nodes can be used almost immediately for executing tasks, they will not contain any data initially. Hence, any map task assigned to the new node will most likely not access local data, introducing the need to *rebalance* the data. HDFS provides a tool for administrators that rebalances data across the nodes in the cluster. In particular, the HDFS Rebalancer analyzes block placement and moves data blocks across DataNodes in order to achieve a uniform distribution of data, while maintaining data availability guarantees.

Hadoop On Demand (HOD) [17] is a system for provisioning and managing virtual clusters on a larger shared physical cluster. Each virtual cluster runs its own Hadoop MapReduce and Hadoop Distributed File System (HDFS) instances, providing better security and performance isolation among the users of each cluster. Under the covers,

HOD uses the Torque resource manager [178] to do the node allocation. On each node, HOD will automatically prepare the configuration files and start the various Hadoop components. HOD is also adaptive in that it can shrink a virtual cluster when the workload changes. In particular, HOD can automatically deallocate nodes from a cluster after it detects no jobs were running for a given time period. This behavior permits the most efficient use of the overall physical cluster resources.

4.7 Fault Tolerance

Fault tolerance features are built in all the layers of the Hadoop stack, including HDFS, MapReduce, and higher-level systems like Pig and Hive. HDFS is designed to reliably store very large files across machines in a large cluster by replicating all blocks of a file across multiple machines [176]. For the common case, when the replication factor is three, HDFS's default placement policy is to put one replica on one node in the local rack and the other two replicas on two different nodes in a different rack. In case of a failure—a DataNode becoming unavailable, a replica becoming corrupted, or a hard disk on a DataNode failing—the NameNode will initiate re-replication of the affected blocks in order to ensure that the replication factor for each block is met. Data corruption is detected via the use of checksum validation, which is performed by the HDFS client by default each time the data is accessed.

The primary way that the Hadoop MapReduce execution engine achieves fault tolerance is through restarting tasks [192]. If a particular task fails or a TaskTracker fails to communicate with the JobTracker after some period of time, then the JobTracker will reschedule the failed task(s) for execution on different TaskTrackers. Failed map tasks are automatically restarted (in other nodes) to process their part of the data again (typically a single file block). Intermediate job data are persisted to disk so that failed reduce tasks can also be restarted without requiring the re-execution of map tasks.

4.8 System Administration

System administration for MapReduce systems is a less established area compared to database administration as done by DBAs. MapReduce system deployments are administered predominantly by cluster operations personnel, some of whom may specialize per system layer (e.g., HDFS operations staff). Some notable differences between parallel database systems and MapReduce systems make it harder to tune MapReduce systems for high performance. For example, it is common in MapReduce systems to interpret data (lazily) at processing time, rather than (eagerly) at loading time. Hence, properties of the input data (e.g., schema) may not be known.

Furthermore, there are many possible ways in which a MapReduce job J in a workload that runs on a MapReduce system could have been generated. A user could have generated J by writing the map and reduce functions in some programming language like Java or Python. J could have been generated by query-based interfaces like Pig or Hive that convert queries specified in some higher-level language to a workflow of MapReduce jobs. J could have been generated by program-based interfaces like Cascading or *FlumeJava* [55] that integrate MapReduce job definitions into popular programming languages. This spectrum of choice in MapReduce job generation increases the complexity of administrative tasks like system monitoring, diagnosing the cause of poor performance, performance tuning, and capacity planning.