Chapter 1. Big Data Technology Primer

Apache Hadoop is a tightly integrated ecosystem of different software products built to provide scalable and reliable distributed storage and distributed processing. The inspiration for much of the Hadoop ecosystem was a sequence of papers published by Google in the 2000s, describing innovations in systems to produce reliable storage (the Google File System), processing (MapReduce, Pregel), and low-latency random-access queries (Bigtable) on hundreds to thousands of potentially unreliable servers. For Google, the primary driver for developing these systems was pure expediency: there simply were no technologies at the time capable of storing and processing the vast datasets it was dealing with. The traditional approach to performing computations on datasets was to invest in a few extremely powerful servers with lots of processors and lots of RAM, slurp the data in from a storage layer (e.g., NAS or SAN), crunch through a computation, and write the results back to storage. As the scale of the data increased, this approach proved both impractical and expensive.

The key innovation, and one which still stands the test of time, was to distribute the datasets across many machines and to split up any computations on that data into many independent, "shared-nothing" chunks, each of which could be run on the same machines storing the data. Although existing technologies could be run on multiple servers, they typically relied heavily on communication between the distributed components, which leads to diminishing returns as the parallelism increases (see Amdahl's law). By contrast, in the distributed-by-design approach, the problem of scale is naturally handled because each independent piece of the computation is responsible for just a small chunk of the dataset. Increased storage and compute power can be obtained by simply adding more servers—a so-called *horizontally scalable* architecture. A key design point when computing at such scales is to design with the assumption of component failure in order to build a reliable system from unreliable components. Such designs solve the problem of cost-effective scaling because the storage and computation can be realized on standard commodity servers.

NOTE

With advances in commodity networking and the general move to cloud computing and storage, the requirement to run computations locally to the data is becoming less critical. If your network infrastructure is good enough, it is no longer essential to use the same underlying hardware for compute and storage. However, the distributed nature and horizontally scalable approach are still fundamental to the efficient operation of these systems.

Hadoop is an open source implementation of these techniques. At its core, it offers a distributed filesystem (HDFS) and a means of running processes across a cluster of servers (YARN). The original distributed processing application built on Hadoop was MapReduce, but since its inception, a wide range of additional software frameworks and libraries have grown up around Hadoop, each one addressing a different use case. In the following section, we go on a whirlwind tour of the core technologies in the Hadoop project, as well as some of the more popular open source frameworks in the ecosystem that run on Hadoop clusters.

WHAT IS A CLUSTER?

In the simplest sense, a *cluster* is just a bunch of servers grouped together to provide one or more functions, such as storage or computation. To users of a cluster, it is generally unimportant which individual machine or set of machines within the cluster performs a computation, stores the data, or performs some other service. By contrast, architects and administrators need to understand the cluster in detail. Figure 1-1 illustrates a cluster layout at a high level.

Figure 1-1. Machine roles in a cluster

Usually we divide a cluster up into two classes of machine: *master* and *worker*.[1] Worker machines are where the real work happens—these machines store data, perform computations, offer services like lookups and searches, and more. Master machines are responsible for coordination, maintaining metadata about the data and services running on the worker machines, and ensuring the services keep running in the event of worker failures. Typically, there are two or three master machines for redundancy and a much larger number of workers. A cluster is scaled up by adding more workers and, when the cluster gets large enough, extra masters.

Often, we want to allow access to the cluster by users and other applications, so we provide some machines to act as *gateway* or *edge* servers. These servers often do not run any services at all but just have the correct client configuration files to access cluster services.

We discuss the various machine types and their purpose in more detail in Chapter 3 and introduce the different types of cluster you might need in Chapter 2.

A Tour of the Landscape

When we say "Hadoop," we usually really mean Hadoop *and* all the data engineering projects and frameworks that have been built around it. In this section, we briefly review a few key technologies, categorized by use case. We are not able to cover every framework in detail—in many cases these have their own full book-level treatments—but we try to give a sense of what they do. This section can be safely skipped if you are already familiar with these technologies, or you can use it as a handy quick reference to remind you of the fundamentals.

The zoo of frameworks, and how they relate to and depend on each other, can appear daunting at first, but with some familiarization, the relationships become clearer. You may have seen representations similar to Figure 1-2, which attempt to show how different components build on each other. These diagrams can be a useful aid to understanding, but they don't always make all the dependencies among projects clear. Projects depend on each other in different ways, but we can think about two main types of dependency: data and control. In the *data plane*, a component depends on another component when reading and writing data, while in the *control plane*, a component depends on another component for metadata or coordination. For the graphically inclined, some of these relationships are shown in Figure 1-3. Don't panic; this isn't meant to be scary, and it's not critical at this stage that you understand exactly how the dependencies work between the components. But the graphs demonstrate the importance of developing a basic understanding of the purpose of each element in the stack. The aim of this section is to give you that context.

Figure 1-2. Standard representation of technologies and dependencies in the Hadoop stack

Figure 1-3. Graphical representation of some dependencies between components in the data and control planes

NOTE

Where there are multiple technologies with a similar design, architecture, and use case, we cover just one but strive to point out the alternatives as much as possible, either, in the text or in "Also consider" sections.

Core Components

The first set of projects are those that form the core of the Hadoop project itself or are key enabling technologies for the rest of the stack: HDFS, YARN, Apache ZooKeeper, and the Apache Hive Metastore. Together, these projects form the foundation on which most other frameworks, projects, and applications running on the cluster depend.

HDFS

The Hadoop Distributed File System (HDFS) is the scalable, fault-tolerant, and distributed filesystem for Hadoop. Based on the original use case of analytics over large-scale datasets, HDFS is optimized to store very large amounts of immutable data with files being typically accessed in long sequential scans. HDFS is *the* critical supporting technology for many of the other components in the stack.

When storing data, HDFS breaks up a file into *blocks* of configurable size, usually something like 128 MiB, and stores a *replica* of each block on multiple servers for resilience and data parallelism. Each worker node in the cluster runs a daemon called a *DataNode* which accepts new blocks and persists them to its local disks. The DataNode is also responsible for serving up data to clients. The DataNode is only aware of blocks and their IDs; it does not have knowledge about the file to which a particular replica belongs. This information is curated by a coordinating process, the *NameNode*, which runs on the master servers and is responsible for maintaining a mapping of files to the blocks, as

well as metadata about the files themselves (things like names, permissions, attributes, and replication factor).

Clients wishing to store blocks must first communicate with the NameNode to be given a list of DataNodes on which to write each block. The client writes to the first DataNode, which in turn streams the data to the next DataNode, and so on in a pipeline. When providing a list of DataNodes for the pipeline, the NameNode takes into account a number of things, including available space on the DataNode and the location of the node—its *rack locality*. The NameNode insures against node and rack failures by ensuring that each block is on at least two different racks. In Figure 1-4, a client writes a file consisting of three blocks to HDFS, and the process distributes and replicates the data across DataNodes.

Figure 1-4. The HDFS write process and how blocks are distributed across DataNodes

Likewise, when reading data, the client asks the NameNode for a list of DataNodes containing the blocks for the files it needs. The client then reads the data directly from the DataNodes, preferring replicas that are local or close, in network terms.

The design of HDFS means that it does not allow in-place updates to the files it stores. This can initially seem quite restrictive until you realize that this immutability allows it to achieve the required horizontal scalability and resilience in a relatively simple way.

HDFS is fault-tolerant because the failure of an individual disk, DataNode, or even rack does not imperil the safety of the data. In these situations, the NameNode simply directs one of the DataNodes that is maintaining a surviving replica to copy the block to another DataNode until the required replication factor is reasserted. Clients reading data are directed to one of the remaining replicas. As such, the whole system is self-healing, provided we allow sufficient capacity and redundancy in the cluster itself.

HDFS is scalable, given that we can simply increase the capacity of the filesystem by adding more DataNodes with local storage. This also has

the nice side effect of increasing the available read and write throughput available to HDFS as a whole.

It is important to note, however, that HDFS does not achieve this resilience and scaling all on its own. We have to use the right servers and design the layout of our clusters to take advantage of the resilience and scalability features that HDFS offers—and in large part, that is what this book is all about. In Chapter 3, we discuss in detail how HDFS interacts with the servers on which its daemons run and how it uses the locally attached disks in these servers. In Chapter 4, we examine the options when putting a network plan together, and in Chapter 12, we cover how to make HDFS as highly available and fault-tolerant as possible.

One final note before we move on. In this short description of HDFS, we glossed over the fact that Hadoop abstracts much of this detail from the client. The API that a client uses is actually a *Hadoop-compatible filesystem*, of which HDFS is just one implementation. We will come across other commonly used implementations in this book, such as cloud-based object storage offerings like Amazon S3.

YARN

Although it's useful to be able to store data in a scalable and resilient way, what we really want is to be able to derive insights from that data. To do so, we need to be able to compute things from the data, in a way that can scale to the volumes we expect to store in our Hadoop filesystem. What's more, we need to be able to run lots of different computations at the same time, making efficient use of the available resources across the cluster and minimizing the required effort to access the data. Each computation processes different volumes of data and requires different amounts of compute power and memory. To manage these competing demands, we need a centralized cluster manager, which is aware of all the available compute resources and the current competing workload demands.

This is exactly what YARN (Yet Another Resource Negotiator) is designed to be. YARN runs a daemon on each worker node, called a *NodeManager*, which reports in to a master process, called the *ResourceManager*. Each NodeManager tells the ResourceManager

how much compute resource (in the form of virtual cores, or *vcores*) and how much memory is available on its node. Resources are parceled out to applications running on the cluster in the form of *containers*, each of which has a defined resource demand—say, 10 containers each with 4 vcores and 8 GB of RAM. The NodeManagers are responsible for starting and monitoring containers on their local nodes and for killing them if they exceed their stated resource allocations.

An application that needs to run computations on the cluster must first ask the ResourceManager for a single container in which to run its own coordination process, called the ApplicationMaster (AM). Despite its name, the AM actually runs on one of the worker machines. ApplicationMasters of different applications will run on different worker machines, thereby ensuring that a failure of a single worker machine will affect only a subset of the applications running on the cluster. Once the AM is running, it requests additional containers from the ResourceManager to run its actual computation. This process is sketched in Figure 1-5: three clients run applications with different resource demands, which are translated into different-sized containers and spread across the NodeManagers for execution.

Figure 1-5. YARN application execution.

The ResourceManager runs a special thread, which is responsible for scheduling application requests and ensuring that containers are allocated equitably between applications and users running applications on the cluster. This scheduler strives to allocate cores and memory fairly between tenants. Tenants and workloads are divided into hierarchical *pools*, each of which has a configurable share of the overall cluster resources.

It should be clear from the description that YARN itself does not perform any computation but rather is a framework for launching such applications distributed across a cluster. YARN provides a suite of APIs for creating these applications; we cover two such implementations, MapReduce and Apache Spark, in "Computational Frameworks".

You'll learn more about making YARN highly available in Chapter 12.

APACHE ZOOKEEPER

The problem of *consensus* is an important topic in computer science. When an application is distributed across many nodes, a key concern is getting these disparate components to agree on the values of some shared parameters. For example, for frameworks with multiple master processes, agreeing on which process should be the *active* master and which should be in *standby* is critical to their correct operation.

Apache ZooKeeper is the resilient, distributed configuration service for the Hadoop ecosystem. Within ZooKeeper, configuration data is stored and accessed in a filesystem-like tree of nodes, called *znodes*, each of which can hold data and be the parent of zero or more child nodes. Clients open a connection to a single ZooKeeper server to create, read, update and delete the znodes.

For resilience, ZooKeeper instances should be deployed on different servers as an *ensemble*. Since ZooKeeper operates on majority consensus, an odd number of servers is required to form a *quorum*. Although even numbers can be deployed, the extra server provides no extra resilience to the ensemble. Each server is identical in functionality, but one of the ensemble is elected as the *leader* node—all other servers are designated *followers*. ZooKeeper guarantees that data updates are applied by a majority of ZooKeeper servers. As long as a majority of servers are up and running, the ensemble is operational. Clients can open connections to any of the servers to perform reads and writes, but writes are forwarded from follower servers to the leader to ensure consistency. ZooKeeper ensures that all state is consistent by guaranteeing that updates are always applied in the same order.

TIP

In general, a quorum with n members can survive up to floor((n–1)/2) failures and still be operational. Thus, a four-member ensemble has the same resiliency properties as an ensemble of three members.

As outlined in Table 1-1, many frameworks in the ecosystem rely on ZooKeeper for maintaining highly available master processes, coordinating tasks, tracking state, and setting general configuration

parameters. You'll learn more about how ZooKeeper is used by other components for high availability in Chapter 12.

Project	Usage of ZooKeeper
HDFS	Coordinating high availability
HBase	Metadata and coordination
Solr	Metadata and coordination
Kafka	Metadata and coordination
YARN	Coordinating high availability
Hive	Table and partition locking and high availability

Table 1-1. ZooKeeper dependencies

APACHE HIVE METASTORE

We'll cover the querying functionality of Apache Hive in a subsequent section when we talk about analytical SQL engines, but one component of the project—the Hive Metastore—is such a key supporting technology for other components of the stack that we need to introduce it early on in this survey.

The Hive Metastore curates information about the structured datasets (as opposed to unstructured binary data) that reside in Hadoop and organizes them into a logical hierarchy of databases, tables, and views. Hive tables have defined schemas, which are specified during table creation. These tables support most of the common data types that you know from the relational database world. The underlying data in the storage engine is expected to match this schema, but for HDFS this is checked only at runtime, a concept commonly referred to as *schema on read*. Hive tables can be defined for data in a number of storage engines, including Apache HBase and Apache Kudu, but by far the most common location is HDFS.

In HDFS, Hive tables are nothing more than directories containing files. For large tables, Hive supports partitioning via subdirectories within the table directory, which can in turn contain nested partitions, if necessary. Within a single partition, or in an unpartitioned table, all files should be stored in the same format; for example, comma-delimited text files or a binary format like Parquet or ORC. The metastore allows tables to be defined as either *managed* or *external*. For managed tables, Hive actively controls the data in the storage engine: if a table is created, Hive builds the structures in the storage engine, for example by making directories on HDFS. If a table is dropped, Hive deletes the data from the storage engine. For external tables, Hive makes no modifications to the underlying storage engine in response to metadata changes, but merely maintains the metadata for the table in its database.

Other projects, such as Apache Impala and Apache Spark, rely on the Hive Metastore as the single source of truth for metadata about structured datasets within the cluster. As such it is a critical component in any deployment.

GOING DEEPER

There are some very good books on the core Hadoop ecosystem, which are well worth reading for a thorough understanding. In particular, see:

- *Hadoop: The Definitive Guide*, 4th Edition, by Tom White (O'Reilly)
- *ZooKeeper*, by Benjamin Reed and Flavio Junqueira (O'Reilly)
- *Programming Hive*, by Dean Wampler, Jason Rutherglen, and Edward Capriolo (O'Reilly)

Computational Frameworks

With the core Hadoop components, we have data stored in HDFS and a means of running distributed applications via YARN. Many frameworks have emerged to allow users to define and compose arbitrary computations and to allow these computations to be broken up into smaller chunks and run in a distributed fashion. Let's look at two of the principal frameworks.

HADOOP MAPREDUCE

MapReduce is the original application for which Hadoop was built and is a Java-based implementation of the blueprint laid out in Google's MapReduce paper. Originally, it was a standalone framework running on the cluster, but it was subsequently ported to YARN as the Hadoop project evolved to support more applications and use cases. Although superseded by newer engines, such as Apache Spark and Apache Flink, it is still worth understanding, given that many higher-level frameworks compile their inputs into MapReduce jobs for execution. These include:

- Apache Hive
- Apache Sqoop
- Apache Oozie
- Apache Pig

NOTE

The terms *map* and *reduce* are borrowed from functional programming, where a map applies a transform function to every element in a collection, and a reduce applies an aggregation function to each element of a list, combining them into fewer summary values.

Essentially, MapReduce divides a computation into three sequential stages: map, shuffle, and reduce. In the map phase, the relevant data is read from HDFS and processed in parallel by multiple independent map *tasks*. These tasks should ideally run wherever the data is located—usually we aim for one map task per HDFS block. The user defines a `map()` function (in code) that processes each record in the file and produces key-value outputs ready for the next phase. In the shuffle phase, the map outputs are fetched by MapReduce and shipped across the network to form input to the reduce tasks. A user-defined `reduce()` function receives all the values for a key in turn and aggregates or combines them into fewer values which summarize the inputs. The essentials of the process are shown in Figure 1-6. In the example, files are read from HDFS by mappers and shuffled by key according to an ID column. The reducers aggregate the remaining columns and write the results back to HDFS.

Figure 1-6. Simple aggregation performed in MapReduce

Sequences of these three simple linear stages can be composed and combined into essentially any computation of arbitrary complexity; for example, advanced transformations, joins, aggregations, and more. Sometimes, for simple transforms that do not require aggregations, the reduce phase is not required at all. Usually, the outputs from a MapReduce job are stored back into HDFS, where they may form the inputs to other jobs.

Despite its simplicity, MapReduce is incredibly powerful and extremely robust and scalable. It does have a couple of drawbacks, though. First, it is quite involved from the point of view of a user, who needs to code and compile `map()` and `reduce()` functions in Java, which is too high a bar for many analysts—composing complex processing pipelines in MapReduce can be a daunting task. Second, MapReduce itself is not particularly efficient. It does a lot of disk-based I/O, which can be expensive when combining processing stages together or doing iterative operations. Multistage pipelines are composed from individual MapReduce jobs with an HDFS I/O barrier in between, with no recognition of potential optimizations in the whole processing graph.

Because of these drawbacks, a number of successors to MapReduce have been developed that aim both to simplify development and to make processing pipelines more efficient. Despite this, the conceptual underpinnings of MapReduce—that data processing should be split up into multiple independent tasks running on different machines (maps), the results of which are then shuffled to and grouped and collated together on another set of machines (reduces)—are fundamental to all distributed data processing engines, including SQL-based frameworks. Apache Spark, Apache Flink, and Apache Impala, although all quite different in their specifics, are all essentially different implementations of this concept.

APACHE SPARK

Apache Spark is a distributed computation framework, with an emphasis on efficiency and usability, which supports both batch and streaming computations. Instead of the user having to express the necessary data

manipulations in terms of pure `map()` and `reduce()` functions as in MapReduce, Spark exposes a rich API of common operations, such as filtering, joining, grouping, and aggregations directly on *Datasets*, which comprise rows all adhering to a particular type or schema. As well as using API methods, users can submit operations directly using a SQL-style dialect (hence the general name of this set of features, Spark SQL), removing much of the requirement to compose pipelines programmatically. With its API, Spark makes the job of composing complex processing pipelines much more tractable to the user. As a simple example, in Figure 1-7, three datasets are read in. Two of these unioned together and joined with a third, filtered dataset. The result is grouped according to a column and aggregated and written to an output. The dataset sources and sinks could be batch-driven and use HDFS or Kudu, or could be processed in a stream to and from Kafka.

Figure 1-7. A typical simple aggregation performed in Spark

A key feature of operations on datasets is that the processing graphs are run through a standard query optimizer before execution, very similar to those found in relational databases or in massively parallel processing query engines. This optimizer can rearrange, combine, and prune the processing graph to obtain the most efficient execution pipeline. In this way, the execution engine can operate on datasets in a much more efficient way, avoiding much of the intermediate I/O from which MapReduce suffers.

One of the principal design goals for Spark was to take full advantage of the memory on worker nodes, which is available in increasing quantities on commodity servers. The ability to store and retrieve data from main memory at speeds which are orders of magnitude faster than those of spinning disks makes certain workloads radically more efficient. Distributed machine learning workloads in particular, which often operate on the same datasets in an iterative fashion, can see huge benefits in runtimes over the equivalent MapReduce execution. Spark allows datasets to be cached in memory on the executors; if the data does not fit entirely into memory, the partitions that cannot be cached are spilled to disk or transparently recomputed at runtime.

Spark implements stream processing as a series of periodic microbatches of datasets. This approach requires only minor code differences in the transformations and actions applied to the data—essentially, the same or very similar code can be used in both batch and streaming modes. One drawback of the micro-batching approach is that it takes at least the interval between batches for an event to be processed, so it is not suitable for use cases requiring millisecond latencies. However, this potential weakness is also a strength because microbatching allows much greater data throughput than can be achieved when processing events one by one. In general, there are relatively few streaming use cases that genuinely require subsecond response times. However, Spark's *structured streaming* functionality promises to bring many of the advantages of an optimized Spark batch computation graph to a streaming context, as well as a low-latency continuous streaming mode.

Spark ships a number of built-in libraries and APIs for machine learning. Spark MLlib allows the process of creating a machine learning model (data preparation, cleansing, feature extraction, and algorithm execution) to be composed into a distributed pipeline. Not all machine learning algorithms can automatically be run in a distributed way, so Spark ships with a few implementations of common classes of problems, such as clustering, classification and regression, and collaborative filtering.

Spark is an extraordinarily powerful framework for data processing and is often (rightly) the de facto choice when creating new batch processing, machine learning, and streaming use cases. It is not the only game in town, though; application architects should also consider options like Apache Flink for batch and stream processing, and Apache Impala (see "Apache Impala") for interactive SQL.

Going deeper

Once again, *Hadoop: The Definitive Guide*, by Tom White, is the best resource to learn more about Hadoop MapReduce. For Spark, there are a few good references:

- The Spark project documentation
- *Spark: The Definitive Guide*, by Bill Chambers and Matei Zaharia (O'Reilly)

- *High Performance Spark*, by Holden Karau and Rachel Warren (O'Reilly)

Analytical SQL Engines

Although MapReduce and Spark are extremely flexible and powerful frameworks, to use them you do need to be comfortable programming in a language like Java, Scala, or Python and should be happy deploying and running code from the command line. The reality is that, in most enterprises, SQL remains the lingua franca of analytics, and the largest, most accessible skill base lies there. Sometimes you need to get things done without the rigmarole of coding, compiling, deploying, and running a full application. What's more, a large body of decision support and business intelligence tools interact with data stores exclusively over SQL. For these reasons, a lot of time and effort has been spent developing SQL-like interfaces to structured data stored in Hadoop. Many of these use MapReduce or Spark as their underlying computation mechanism, but some are computation engines in their own right. Each engine is focused on querying data that already exists in the storage engine or on inserting new data in bulk into those engines. They are designed for large-scale analytics and not for small-scale transactional processing. Let's look at the principal players.

APACHE HIVE

Apache Hive is the original data warehousing technology for Hadoop. It was developed at Facebook and was the first to offer a SQL-like language, called HiveQL, to allow analysts to query structured data stored in HDFS without having to first compile and deploy code. Hive supports common SQL query concepts, like table joins, unions, subqueries, and views. At a high level, Hive parses a user query, optimizes it, and compiles it into one or more chained batch computations, which it runs on the cluster. Typically these computations are executed as MapReduce jobs, but Hive can also use Apache Tez and Spark as its backing execution engine. Hive has two main components: a metadata server and a query server. We covered the Hive Metastore earlier, so we focus on the querying functionality in this section.

Users who want to run SQL queries do so via the query server, called HiveServer2 (HS2). Users open *sessions* with the query server and

submit queries in the HQL dialect. Hive parses these queries, optimizes them as much as possible, and compiles them into one or more batch jobs. Queries containing subqueries get compiled into multistage jobs, with intermediate data from each stage stored in a temporary location on HDFS. HS2 supports multiple concurrent user sessions and ensures consistency via shared or exclusive locks in ZooKeeper. The query parser and compiler uses a cost-based optimizer to build a query plan and can use table and column statistics (which are also stored in the metastore) to choose the right strategy when joining tables. Hive can read a multitude of file formats through its built-in serialization and deserialization libraries (called *SerDes*) and can also be extended with custom formats.

Figure 1-8 shows a high-level view of Hive operation. A client submits queries to a HiveServer2 instance as part of a user session. HiveServer2 retrieves information for the databases and tables in the queries from the Hive Metastore. The queries are then optimized and compiled into sequences of jobs (J) in MapReduce, Tez, or Spark. After the jobs are complete, the results are returned to the remote client via HiveServer2.

Figure 1-8. High-level overview of Hive operation

Hive is not generally considered to be an interactive query engine (although recently speed improvements have been made via long-lived processes which begin to move it into this realm). Many queries result in chains of MapReduce jobs that can take many minutes (or even hours) to complete. Hive is thus ideally suited to offline batch jobs for extract, transform, load (ETL) operations; reporting; or other bulk data manipulations. Hive-based workflows are a trusted staple of big data clusters and are generally extremely robust. Although Spark SQL is increasingly coming into favor, Hive remains—and will continue to be—an essential tool in the big data toolkit.

We will encounter Hive again when discussing how to deploy it for high availability in Chapter 12.

Going deeper

Much information about Hive is contained in blog posts and other articles spread around the web, but there are some good references:

- The Apache Hive wiki (contains a lot of useful information, including the HQL language reference)
- *Programming Hive*, by Dean Wampler, Jason Rutherglen, and Edward Capriolo (O'Reilly)

NOTE

Although we covered it in "Computational Frameworks", Spark is also a key player in the analytical SQL space. The Spark SQL functionality supports a wide range of workloads for both ETL and reporting and can also play a role in interactive query use cases. For new implementations of batch SQL workloads, Spark should probably be considered as the default starting point.

APACHE IMPALA

Apache Impala is a massively parallel processing (MPP) engine designed to support fast, interactive SQL queries on massive datasets in Hadoop or cloud storage. Its key design goal is to enable multiple concurrent, ad hoc, reporting-style queries covering terabytes of data to complete within a few seconds. It is squarely aimed at supporting analysts who wish to execute their own SQL queries, directly or via UI-driven business intelligence (BI) tools.

In contrast to Hive or Spark SQL, Impala does not convert queries into batch jobs to run under YARN. Instead it is a standalone service, implemented in C++, with its own worker processes which run queries, the Impala *daemons*. Unlike with Hive, there is no centralized query server; each Impala daemon can accept user queries and acts as the *coordinator* node for the query. Users can submit queries via JDBC or ODBC, via a UI such as Hue, or via the supplied command-line shell. Submitted queries are compiled into a distributed query plan. This plan is an operator tree divided into *fragments*. Each fragment is a group of plan nodes in the tree which can run together. The daemon sends different *instances* of the plan fragments to daemons in the cluster to

execute against their local data, where they are run in one or more threads within the daemon process.

Because of its focus on speed and efficiency, Impala uses a different execution model, in which data is streamed from its source through a tree of distributed *operators*. Rows read by scan nodes are processed by fragment instances and streamed to other instances, which may be responsible for joining, grouping, or aggregation via exchange operators. The final results from distributed fragment instances are streamed back to the coordinator daemon, which executes any final aggregations before informing the user there are results to fetch.

The query process is outlined in Figure 1-9. A client chooses an Impala daemon server to which to submit its query. This coordinator node compiles and optimizes the query into remote execution fragments which are sent to the other daemons in the cluster (query initialization). The daemons execute the operators in the fragments and exchange rows between each other as required (distributed execution). As they become available, they stream the results to the coordinator, which may perform final aggregations and computations before streaming them to the client.