BIG DATA PROCESSING BEYOND HADOOP AND MAPREDUCE

Ravi Sharda
Consultant Software Engineer
EMC, India Center of Excellence
Ravi.Sharda@emc.com

EMC²

Table of Contents

1. INTRODUCTION 5

2. INTERACTIVE QUERYING OVER HADOOP 7

2.1. APACHE DRILL: LOW-LATENCY SELF-SERVICE DISTRIBUTED QUERY ENGINE 8

2.1.1. Overview 8

2.1.2. Architecture 9

2.1.3. Discussion 11

2.2. STINGER INITIATIVE: AN INITIATIVE TO IMPROVE APACHE HIVE'S PERFORMANCE 100X 12

2.2.1. Overview 12

2.2.2. Discussion 13

2.3. APACHE TEZ: A DISTRIBUTED EXECUTION ENGINE FOR INTERACTIVE JOBS 13

2.3.1. Background 13

2.3.2. Overview 14

3. ITERATIVE COMPUTATIONS 16

3.1. HALOOP: A MODIFIED VERSION OF HADOOP OPTIMIZED FOR ITERATIVE PROCESSING 18

3.1.1. Overview 18

3.2. APACHE GIRAPH: BULK SYNCHRONOUS PARALLEL GRAPH PROCESSING 18

3.2.1. Background 18

3.2.2. Bulk Synchronous Parallel (BSP) Computing Model 19

3.2.3. Overview 20

4. MOVING BEYOND MAPREDUCE 22

4.1. YET-ANOTHER-RESOURCE-MANAGER (YARN) 22

4.1.1. Background: Workings of Pre-YARN Hadoop MR 22

4.1.2. Enter YARN 23

4.1.3. Discussion 24

4.2. APACHE SPARK: AN ALTERNATIVE CLUSTER COMPUTING PLATFORM 26

4.2.1. Overview 26

4.2.2. Architecture 26

4.2.3. Intersections of Spark and Hadoop 28

4.2.4. Discussion 28

5. CONCLUSION 29

6. ACKNOWLEDGEMENT 30

7 **REFERENCES 32**

Table of Figures

Figure 1 Functional Architecture – Apache Drill 9
Figure 2 Core Modules in a Drillbit 10
Figure 3 Apache Tez in Hadoop 2.0 Stack 14
Figure 4 Typical Dataflow in Iterative Processing with Hadoop MR 16
Figure 5 YARN's Role in the Hadoop Ecosystem 25

1. Introduction

Hadoop and MapReduce (MR) have been de-facto standards for Big Data processing for a long time now, so much so that they are seen by many as synonymous with "Big Data". With MR data processing model and Hadoop Distributed File System at its core, Hadoop is great at storing and processing large amounts of data. It allows a programmer to divide a larger problem into smaller mapper and reducer tasks that can be executed, mostly in parallel, over a network of machines. Hadoop's runtime hides much of the gory details of distributed and parallel data processing from the programmer, such as partitioning input data, breaking down MR jobs into individual tasks, scheduling the tasks for parallel execution, co-locating processing to where the data is (to the extent possible), monitoring the progress of tasks and jobs, handling partial errors and fault-tolerance on unreliable commodity hardware, synchronizing results and tasks when necessary, and so on.

As such, Hadoop and MR lower the entry barrier for Big Data processing, by making data-intensive processing easy and cost-effective. Easy, because programmers need to just write and deploy Mapper and Reducer tasks and Hadoop handles much of the scaffolding. Cheap, because Hadoop MR jobs can work over commodity servers, avoiding the need to deploy specialized (read costly) hardware.

MR and Hadoop are designed for *"batch-oriented"* processing of large-scale data processing rather than for interactive use.

However, many classes of applications do require low-latency analysis. Consider for example a credit card fraud detection application that requires real-time results, or a Business Intelligence (BI) application serving results of ad-hoc queries to a user.

Similarly, interactive/exploratory querying, involving firing multiple queries that are closely related to one another, isn't served well by Hadoop MR. This is because Hadoop fires new MR jobs and loads data from disks for each job irrespective of the data access pattern and history. Factor this with high-latency execution of an individual MR job and users' general expectations of sub-second response times, and it's easy to see why interactive analysis with Hadoop MapReduce is seen as impractical.

Moreover, Hadoop MR is not suitable for computations involving iterative processing, owing to the overheads of repeated fetching of data since Hadoop doesn't store working sets of data in memory. Iterative computations are common in many applications including machine learning and graph processing algorithms.

This article provides an overview of various new and upcoming alternatives to Hadoop MR, with a special focus on how they go about addressing the issues discussed earlier. Some of these alternatives are at the infrastructure-level, replacing the guts of Hadoop while keeping interface-level semantic compatibility with Hadoop. Others choose to re-use existing Hadoop infrastructure, but provide a new higher level interface designed to address some of the limitations discussed earlier. Prominent examples include: Apache Spark, Apache Drill, *Yet Another Resource Navigator (YARN), Apache Tez,* HaLoop, Apache Hama, Apache Giraph, and Open MPI.

2. Interactive Querying over Hadoop

Parallel processing of map/reduce tasks and high-throughput access to data sets via sequential reads enable Hadoop MR jobs to execute in minutes and hours; those jobs would otherwise take days or months using serial processing steps over the same dataset. This works well for batch-oriented jobs that run behind-the-scenes and do not have low-latency expectations.

As such, even the simplest single-stage Hadoop MR job exhibits high latency – relative to interactive response times of a second or less – regardless of the volume of data it operates on and the number of nodes processing tasks in parallel. This is primarily because of; a) Hadoop's design emphasis on efficient processing of batch-oriented jobs, and b) synchronization barriers inherent in its MapReduce programming model. These are elaborated in greater detail in the following paragraphs.

Several aspects of Hadoop's design explicitly emphasize efficiencies at scale for batch-oriented applications/MR jobs. Batch-oriented MR jobs like ETL and aggregation often involve running batch operations on large data sets, implying that the jobs make long streaming reads from disks and large sequential writes to disks [Lin]. Such workloads are better served by avoiding caching intermediate data or results in memory, and that's exactly what Hadoop File System (HDFS) does. Similarly, Hadoop stores intermediate data produced during Map stage on disks, which in turn makes it easier to handle failures at run time. However doing so also adds considerable performance overheads to overall computation time, which is not much of an issue for batch-oriented jobs.

Hadoop's MR programming model also lends itself to certain inherent latencies. For example, Shuffle/reduce tasks can only commence after all of the Map tasks are completed. Similarly, in a multi-stage application comprising of a chain of MR jobs, a job must complete before the next in the chain can start executing [Rentachintala].

Many Big Data applications, like Business Intelligence (BI) dashboards, enterprise reporting, big data visualization, ad-hoc reporting, interactive querying, fraud detection, security threat detection, etc. are latency-sensitive. Some of these applications are latency-sensitive simply because human users are responsible for triggering their execution and tend to demand interactive response times of a sub-second or less. In

other cases, applications may be latency sensitive because the results must be used in-line of a transaction and the transaction itself must complete quickly to be effective. A credit card fraud detection system is an example of this, where a potential fraud may need to be detected in-line of a payment transaction; since payment transactions bear expectations of low-latency, it becomes incumbent on the fraud detection system to compute its decision quickly.

If an individual MR job features high-latency, the problem is only further exacerbated in multi-stage application comprising of a chain of Hadoop MR jobs.

Several solutions have come about, which address the high-latency problems of Hadoop MR applications. Taking a cue from Google's Dremel, Cloudera's Impala and Apache Drill take the approach of bypassing compilation of queries into Hadoop MR jobs (unlike Hive's approach) and instead break down the queries into smaller sub-queries – each of which is executed in parallel. Hortonworks-led Stinger initiative takes the approach of making changes to Hive to make it faster. Apache Tez follows the approach of arranging tasks in a Directed Acyclic Graph (DAG) and optimizing their execution using DAG as the unit of execution. We discuss some of these solutions in the following sub-sections.

2.1. Apache Drill: Low-Latency Self-Service Distributed Query Engine

2.1.1. Overview

Inspired by Google's Dremel – a *"technology for interactive analysis of Web-scale datasets* [Melnik]*"* – Apache Drill is a low-latency SQL query engine for big data sets stored in a variety of data sources, including Hadoop. Apache Drill is not a "database", but is rather a SQL query layer over underlying data sources including Hadoop, NoSQL databases (HBase, MongoDB, etc.), etc. [Hausenblas].

Unlike Hive, which compiles and executes SQL queries into MR jobs, Apache Drill takes the approach of compiling user queries into query fragments that are executed directly over data nodes and then assembling an aggregated response, building on ideas from *"shared-nothing"* style parallel databases or Massively Parallel Processing (MPP) databases to be specific.

Apache Drill graduated from an Apache Incubator to a Top-Level Project (TLP) in early December 2014, which in turn signifies maturity of the technology and an active community supporting the project. Not to mention, the website moved from incubator.apache.org/drill to drill.apache.org, reflecting the change.

2.1.2. Architecture

The foundational component of Drill's architecture is the *"Drillbit"* service. The following figure depicts how a client application's query is processed in Drill.

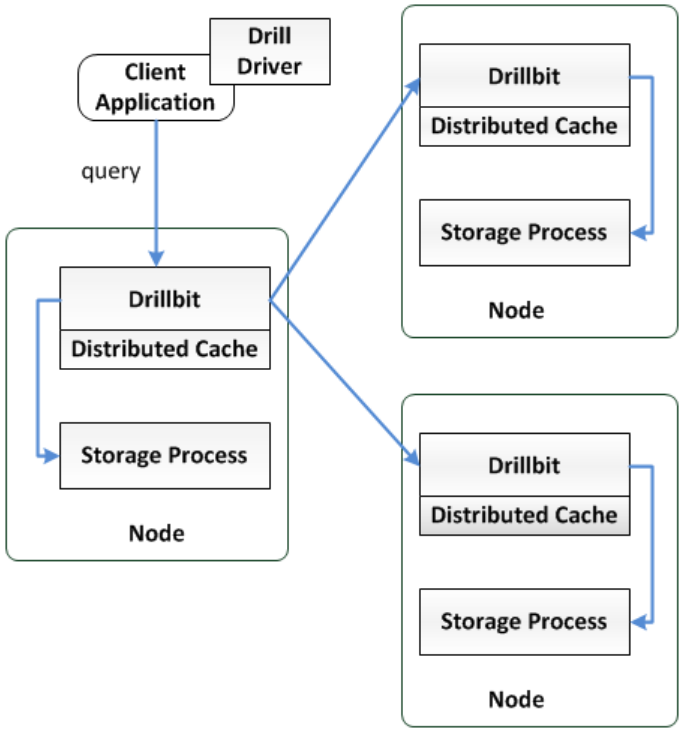

Figure 1: Functional Architecture – Apache Drill

The Drillbit service accepts and responds to client requests. In Hadoop cluster environments, each of the data nodes that need to respond to Drill queries have the Drillbit service installed and running.

A client user or an application issues a query to any of the Drillbit nodes through one of its interfaces: command line shell (Drill Shell), Web UI (Drill Web UI), Open Database Connectivity (ODBC)/Java Database Connectivity (JDBC) libraries or the C++ Application Programming Interface (API). Drill implements full ANSI SQL 2003: this makes it easier for business users to perform interactive analysis as well as integration with third party BI tools via compliant ODBC/JDBC drivers.

Once a Drillbit node accepts a request, it becomes responsible for driving the request through completion including fetching a list of available Drillbit nodes in the cluster (from ZooKeeper) and determining the appropriate nodes to execute the fragments of the execution plan.

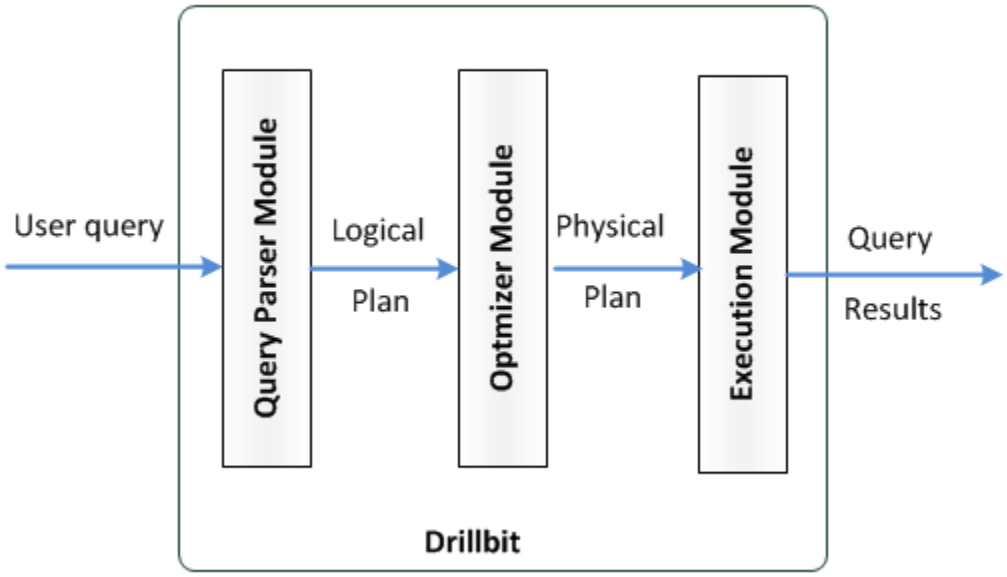

Figure 2: Core Modules in a Drillbit

The figure above illustrates the data flow through major modules within the Drillbit service. A *"Parser"* module parses and transforms the incoming user query into a logical plan. The logical plan represents the query as a dataflow program in a DAG, and typically lives in memory in the form of Java objects [Hausenblas].

An *"Optimizer"* module, then transforms the logical plan into a physical plan that - among other things - represents allocation of corresponding query fragments to individual Drillbit

nodes. By default, a cost-based optimizer is used, but there is also an extension point for plugging-in other optimizers. The default cost-based optimizer takes several things into account including topological and data locality considerations. For example, a query fragment will ideally execute on the same data node that hosts the sub-set of the data that it reads.

An *"Execution"* module within the driving Drillbit service then schedules execution of query fragments according to the physical plan and keeps track of their execution statuses.

Finally, the driving Drillbit node receives results from all the Drillbit nodes involved in processing parts of the user query, collates/merges those results, and returns an aggregated response to client users/applications.

2.1.3. Discussion

Low-latency access to data is a key requirement for workloads involving ad-hoc, interactive, and exploratory data analysis over Hadoop-resident big data sets. What design aspects of Apache Drill enable that?

First, by avoiding compiling queries into Hadoop MR jobs a number of performance pitfalls of Hadoop MR are avoided. These include performance issues caused by barrier synchronization steps and persistence steps in Hadoop MR. Examples of the former are: waiting for completion of all Map tasks prior to commencement of shuffle reduce tasks, or, waiting for completion of a job before executing the next job in a chain of multi-stage jobs. Examples of the latter are: persistence of intermediate data between different phases of a MR job to disk and corresponding de-serialization steps between different phases.

Second, distributing queries into finer-grained query fragments throws up more opportunities for optimizing query execution. For example, when Drill scans *"whole tables"*, it spreads out scanning into smaller fragments that can be executed, which in turn can be executed in parallel and largely operating over local data sets over a network of Drillbit services.

Additionally, Apache Drill optimizes query execution for Columnar Storage and execution. For example, when working with data persisted in columnar formats such as Parquet, Drill avoids accessing columns that are not part of the query and processes directly on columnar data avoiding row materialization [Bevens], lowering memory footprints, and improving performance of BI/analytic queries. Moreover, vectorized processing in Drill enables partitioning of data into "Record Batches": Record Batches form the unit of work for the query system which in turn lend themselves well to efficient processing by leveraging *"modern chip-technology with deep-pipelines CPUs* [Bevens]".

But that's not all. Apache Drill also provides a query interface that allows a user to build a query without knowing in advance what queries to issue. Support for ANSI SQL:2003 makes it easier for business users to leverage their pre-existing knowledge for composing interactions with the system. Additionally, support for standard ANSI SQL allows for integration with third party BI/analytics tools via standard ODBC/JDBC interfaces and drivers.

2.2. Stinger Initiative: An Initiative to Improve Apache Hive's Performance 100X

2.2.1. Overview

Since its introduction, Apache Hive (along with its HiveQL interface) has become the de facto SQL-like query interface for Hadoop, and is used primarily for large-scale operational batch processing geared toward enterprise reporting, data mining, and data preparation use cases. Hive compiles queries into Hadoop MR jobs and therefore inherits much of the latency problems Hadoop MR jobs suffer.

An increasing demand for real-time and interactive analytical querying led to the commencement of Hortonworks-led Stinger Initiative that had an initial aim of improving Hive's performance dramatically: reduce latency by 100x, to be specific.

While the initiative is not part of the Hadoop project ecosystem, improvements made as part of the initiative go into pre-existing Hadoop projects, making the benefits available to the community at large.

The original initiative is claimed to have largely met its objective. Stinger.next is a continuation of the initiative with the aim of further improving performance of Hive as well as introducing new SQL features that add to the analytical querying capability of Hive.

2.2.2. Discussion

Among the major changes made in the original Stinger initiative toward reducing latencies of Hive queries were:

- Optimizing Hive query execution infrastructure so that queries can run much faster. Examples of such changes include removing redundant operators from Hive's Map/Reduce plans, predicate pushdown (filtering at the storage layer, rather than in the SQL frontend), vectorized query execution that batches operations, caching of hot tables, introduction of in-memory joins, etc.

- A new column store-based file format for storage that enables better performance of analytical queries.

- Drive evolution of Apache Tez for faster execution of MR jobs: Hive compiles user queries into MR jobs, and making compiled MR jobs efficient has a direct bearing on Hive's performance. In fact, one may argue that Apache Tez was born out of the requirements of this initiative. We discuss more on Tez in the next section.

There are many more changes in the pipeline, which are expected to make Hive queries even more efficient.

2.3. Apache Tez: A Distributed Execution Engine for Interactive Jobs

2.3.1. Background

As we already know, Hadoop MR programs are written using just two primary data processing primitives: "*map*" and "*reduce*". Every application running on Hadoop MR is expressed using these primitives, ultimately yielding either a single MR job or a multi-stage job comprising a chain of successive MR jobs.

One way to run a multi-stage application is to execute each job in a chain as a separate MR job, leveraging no knowledge of what came before it in the chain. Doing so also implies that any job in a chain must wait for all of the jobs that come before it to finish, since there can be inter-job dependencies in a chain – dependencies such as output of one job is input to another. But running a multi-stage application in a sequential fashion like that can make it hugely inefficient.

An alternative approach to handling a multi-stage application is to represent such a complex job as a Directed Acyclic Graph (DAG) and to execute the DAG as a whole unit. A DAG, in this context, refers to a directed and acyclic graph of job steps (such as map/reduce tasks) as vertices and producer/consumer connections as edges. Order of execution of the steps is as per directionality of the graph edges. “Acyclic” property of a DAG implies that there are no cycles in the graph.

2.3.2. Overview

Support for third-party ApplicationMasters in YARN/Hadoop 2.0 made it possible to plug-in execution engines other than MapReduce (more on this later). Apache Tez leverages that extension point to plug in a new execution engine that can handle traditional MR jobs as well as DAG-based jobs.

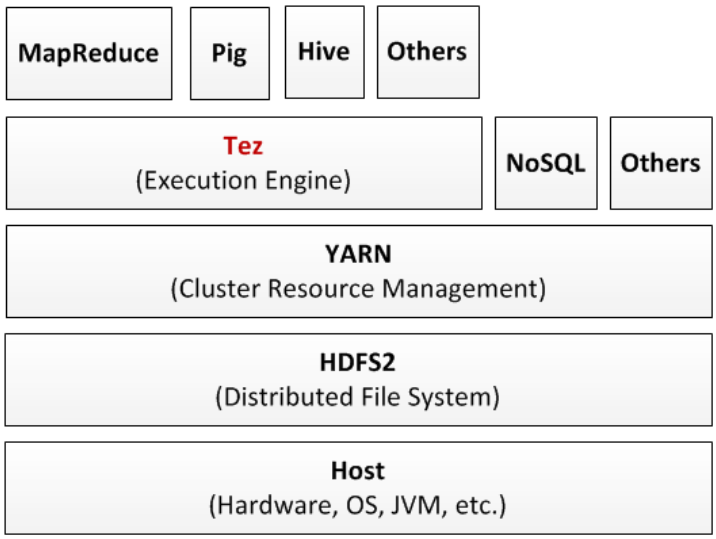

Figure 3: Apache Tez in Hadoop 2.0 Stack

A key short-term goal of Tez is to provide an execution engine that allows for expressing complex query plans generated by Hive in an expressive and efficient manner, and to support deterministic optimizations and high performance execution at runtime. Using Tez's API for defining DAGs, Hive expresses its query plans in the form of DAGs. Vertices represent Map or Reduce tasks and edges represent producer/consumer relationships among the tasks. Each DAG represents a query plan and is on Tez as a Tez job.

Executing a multi-stage job as a DAG has several benefits. One is that tasks representing independent vertices, i.e. those that have no incoming edge, can run in parallel (as opposed to sequential). Second, some of the scheduling overheads of executing MR jobs individually can be avoided. Similarly, since all of the steps involved in running a query can be fully represented in a DAG at runtime and the flow from step to step can be determined upfront, the execution engine can keep intermediate results in memory for as long as necessary, thereby improving the performance of the overall query/job.

In summary, Tez (which in the Hindi language means speed) makes Hive queries (as well as other types of jobs running on top of it, such as Pig jobs) much more efficient, thereby supporting faster response times of analyzes.

3. Iterative Computations

Many data processing applications involve iterative computations where data is processed iteratively until the computation satisfies some convergence criteria (also called termination condition). Algorithms with iterative structures can be found in domains such as machine learning, dimension reduction, link analysis, neural networks, social network analysis, and network traffic analysis. Many of the algorithms in these domains are really recursive structures, but the only way to express them in a programming model like MR is in the form of iterative computations.

Hadoop, on its own, does not natively support iterative computations. Instead, to achieve the effect of iterative computations, the application developer must implement such computations as a program – a driver program or an external script or an external workflow hosted on a workflow engine like Oozie – that implements its own iteration logic, as illustrated in the following figure.

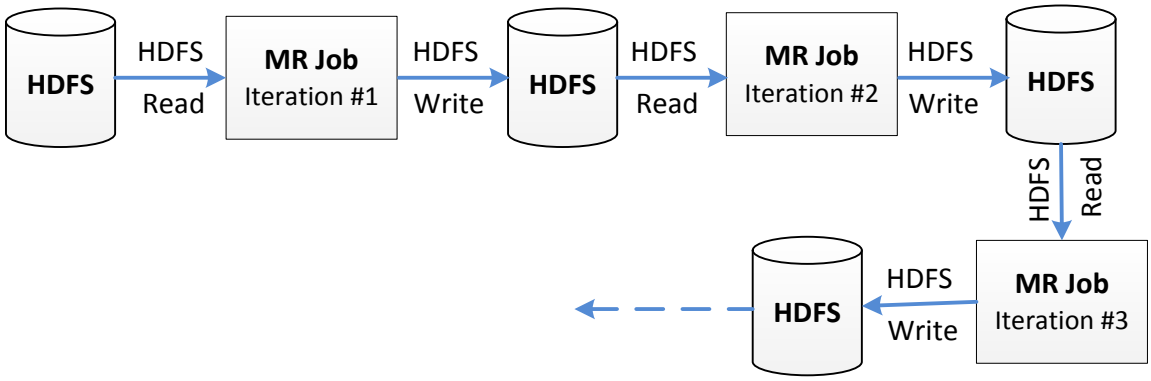

Figure 4: Typical Dataflow in Iterative Processing with Hadoop MR

The program would need to issue the same job afresh repeatedly until the relevant termination condition is satisfied and orchestrate individual jobs across iteration boundaries in a way where shared data is persisted to or read from the file system at output and input points respectively.

Manual orchestration of iterative computations by stringing together a bunch of MR jobs is ill-suited for iterative computations for the following reasons:

- First, it makes it much harder for application programmers to express such computations efficiently.
- Scheduling overhead of individual MR jobs with respect to latencies means that the overall application suffers more and more latencies in direct proportion to the number of jobs involved in the chain.
- Even though much of the data is unchanged from iteration to iteration, the data must still be serialized and written to disks at the end of iteration and read from disks and re-processed in another – wasting I/O, network bandwidth, and CPU resources [Bu].
- Checking for terminating condition at the end of each iteration may involve fixed-point verification – i.e. checking that the results of the last two iterations haven't changed. While fixed-point verification would mean different things in different iterative applications, it would typically involve running an additional MR job at the end of each iteration for comparing the results, adding to performance overhead.

There are a number of solutions that address these problems, including HaLoop, Giraph, Twister, and iHadoop. We discuss some of these in the following sections.

Apache Spark – discussed later in this article – also addresses needs of iterative computations, and it does so by providing programming abstractions that makes it easier to express iterative computations, as well as by making such computations much more efficient.

3.1. HaLoop: A Modified Version of Hadoop Optimized for Iterative Processing

3.1.1. Overview

HaLoop is a modified version of Hadoop that extends MR paradigm for use in iterative computations, along two major dimensions.

First, it provides programming abstractions for expressing iterative computations more naturally, as opposed to the "stringing together of MR jobs" approach explained earlier. It introduces new programming abstractions in the form of new functions such as a) functions that allow defining loop bodies, like "AddMap" and "AddReduce", b) functions for checking termination conditions, like "SetFixedPointThreshold" and "SetMaxNumOfIterations", and c) functions that distinguish loop-variant and loop-invariant data, like "AddStepInput" and "AddInvariantTable" [Li].

Second, it makes iterative computations a lot more efficient. It does so using mechanisms such as ([Bu], [Li])

- Caching invariant data that is shared across iterations, and utilizing the cache to avoid reading of that data from disks.
- Caching of fixed-point verification data, avoiding the need for spinning a separate MR job just for that purpose.
- Using a new purpose-built loop-aware task scheduler that leverages data locality to generate more efficient schedules for iterative computations.

Refer to [Bu, Li] for more details on the above.

3.2. Apache Giraph: Bulk Synchronous Parallel Graph Processing

3.2.1. Background

A Graph is simply a collection of nodes/vertices and edges/links. A vertex represents an entity such as a person. An edge connects two vertices, representing a certain

relationship between the two edges – for example two persons who are friends of each other. As such Graphs are efficient in expressing relationships among entities and widely used in areas such as social network analysis.

Graph algorithms can be expressed as a chain of MR jobs where the entire state of the graph is passed from one job to next. However, this approach can lead to poor performance, like in the case of other types of iterative computations (discussed earlier). Those problems are actually magnified in a big way in graph processing applications because graph applications tend to have much larger number of iterations than non-graph iterative processing.

3.2.2. Bulk Synchronous Parallel (BSP) Computing Model

It is difficult to discuss Giraph without delving into the Bulk Synchronous Parallel (BSP) computing model, since Giraph is primarily based on that model. The following paragraphs in this section provide a brief overview.

Introduced by Leslie Valiant in the 1980's, BSP is a much older parallel computing model that the popular MapReduce model.

In BSP, a program executes as a sequence of parallel supersteps separated by barrier synchronization, each of which is comprised of three ordered phases [BSP]:

1. A local and concurrent *computation* phase, where each processor performs its computation using values available locally (usually in memory), and requests data transfers to/from other processors. This phase may overlap with the communication phase.

2. A *communication* phase, where the BSP network delivers the data transfers based on requests made during computation phase [Hassan].

3. A global *barrier synchronization* phase, which waits for all data transfers to complete, making the transferred data available for the next superstep [Hassan]. Also, a processor that has entered the barrier waits for others to enter the barrier too.

Once a processor has come out of the barrier it can operate independently on data it has received through messages from other processors.

Apache Giraph, Apache Hama, and Stanford Graph Processing System (GPS), are realizations of the BSP paradigm.

3.2.3. Overview

Apache Giraph (https://giraph.apache.org/) is arguably the most popular scalable and large-scale graph processing framework. It is an open-source clone of Google's Pregel, and is designed to work on top of Hadoop: Giraph jobs are launched as normal Hadoop MR jobs leveraging the underlying Hadoop infrastructure. Giraph is already in use at companies like Facebook and PayPal in applications like network analysis involving massive data sets (billions of vertices and edges).

Before the advent of YARN/Hadoop 2.0, Giraph used the underlying Hadoop MR programming model, but with the introduction of YARN in Hadoop 2.0, Giraph is no longer tied to the MR model that was inherently inefficient for large-scale graph processing.

Giraph takes its input through sources such as HDFS or Hive tables as a graph composed of vertices and edges (stored in the input source or adjacency matrix, etc.). Depending on the application domain, vertices can be entities or program units. Like Pregel, Giraph implements a vertex-oriented graph processing engine based on the BSP parallel computing model.

Graph Computations are executed as a series of supersteps. Each vertex executes a user-defined computing function, which does local computation (similar to the BSP model) of data available at the vertex, using locally available data, such as data received from other vertices, vertex, and outgoing edge values. Each vertex sends messages to other vertices post execution passing any computational results to other vertices. Thus, the program is designed from the local execution capability of the vertex, and message passing between vertices without offering any inter-vertex data access. There is also a barrier between consecutive supersteps, wherein messages from the current superstep get delivered to the destination vertices, and they start computing after every vertex has

completed computing the current superstep. It is also possible to mutate the graph by adding or removing vertices or edges, during a superstep. Computation halts after all the vertices have voted to halt and there are no messages to be delivered.

Giraph's vertex-oriented approach is a more natural way of modeling graph problems. For example, an application developer implements a vertex, rather than MR jobs. Giraph's BSP implementation allows much of graph processing to occur in memory, avoiding the need for launching too many MR jobs or reading from/writing to disks and thereby making graph processing more performant than otherwise possible with Hadoop MR.

4. Moving Beyond MapReduce

We have discussed several solutions that address limitations of Hadoop/MR. Arguably, none of those solutions are as foundational as Hadoop YARN and Spark, especially with respect to addressing many of the inherent limitations of Hadoop MR (some of which we have discussed so far). The following sub-sections describe YARN and Spark, and how they set out to solve those limitations.

4.1. Yet-Another-Resource-Negotiator (YARN)

4.1.1. Background: Workings of Pre-YARN Hadoop MR

As discussed earlier, Hadoop's MR programming model and the distributed execution engine greatly simplified processing of large-scale data sets. But as Hadoop became more and more popular, some of its major limitations also came to the fore: a) the computational inefficiencies of MR jobs, and b) scalability and reliability limitations caused by the core design of the distributed computation engine. We discussed the former in greater length under other sections of this document. Let's delve into the latter, in order to make sense of why and how YARN set out to solve those problems.

In Hadoop 1.x, clients launch MapReduce applications to a centralized JobTracker process/daemon. The process accepts an MR application as a job from a client and assigns parts of that job – Map and Reduce tasks – for execution on the "slave" TaskTracker daemons spread across the shared cluster. To be able to do so, the JobTracker keeps track of which of the TaskTrackers are alive, as well as, available Map and Reduce slots on the Tasktracker. The JobTracker also monitors task execution on TaskTrackers and makes the status and other details of the job available for clients. It helps to think of the JobTracker as a *"shared resource across jobs, across users* [Murthy]". Each of the TaskTracker daemons spawn child processes for executing individual tasks, and updates the TaskTracker with the status of tasks it runs.

Having a centralized JobTracker shared across jobs and users led to several scalability and reliability issues - especially in large clusters. The next paragraphs in this sub-section explain some of those issues.

A JobTracker keeps in-memory representations of a job and its tasks, including dynamic values associated with them such as job counters and configurations, as well as static values. Therefore, memory utilized as part of processing a job is practically unbounded [Murthy]. Hence, memory consumed as part of a single memory-intensive job could potentially lead to runaway memory usage, in turn leading in to JVM pauses and even crashes. A JobTracker being a shared resource, if the JobTracker needed to be restarted, the entire cluster would experience downtime. As such, limits to scalability were encountered in large deployments, placing a practical limit of about a few thousand nodes and about 40000 tasks running concurrently within a cluster..

Similarly, if a Map or Reduce task consumed a lot of memory on a node, it may negatively impact other processes like tasks running on behalf other jobs and even Hadoop daemons like TaskTracker and DataNode. If the TaskTracker or the DataNode goes down, the node becomes unusable as a Hadoop cluster resource.

4.1.2. Enter YARN

In Hadoop 1.0, as indicated in the previous sub-section, the JobTracker had two distinct functions: resource management and Job scheduling/monitoring. These functions, with Hadoop 2.x YARN architecture are now assigned to separate daemons.

Resource management functions in Hadoop 2.x are now fulfilled by a generic resource management framework comprising of a global ResourceManager (RM) and a per-node NodeManager (NM) - acting as RM's slave.

- The RM is responsible for arbitrating division of resources among applications running on the cluster.

- The NM is responsible for allocating containers to applications on the computational nodes of the cluster, based on requests from the RM. It also manages the containers' life cycle, monitors them for resource usage, and reports resource usage metrics back to the RM. Essentially, it is the focal point of resource management on individual computational node in a Hadoop cluster.

Allocation of computational resources to applications – resources being memory, CPU, disk and network – is done by a pluggable scheduler on the RM, of course, subject to various constraints. The pluggable scheduler enables use of a variety of scheduling algorithms such as those focused on specific use cases or programming models.

Note the reference to *"application"* as opposed to a job in the preceding paragraph. In Hadoop 2.x, an *"application*" is either a job such as an individual MapReduce job, or a collection of jobs organized as a Directed Acyclic Graph (DAG) of jobs [Wadkar].

Moreover, YARN introduces an application-specific ApplicationMaster that replaces the older dedicated and single JobTracker, with respect to job-oriented functionality. It is application-specific in the sense that when a user submits an application, an instance of ApplicationMaster is started to coordinate the execution of that application, and all application-framework specific code lies within the ApplicationMaster. Its functions include: negotiating appropriate resource containers with the RM, monitoring the containers (tasks), restarting failed tasks, speculatively running slow tasks, and so on.

Each application, via the per-application ApplicationMaster instance, gets its share of resources via the "container" abstraction, which in turn is a result of a RM granting a resource request made by it. Responsibilities for monitoring applications, restarting failed tasks, and so on have now been shifted to ApplicationMaster. Each application having its own instance of the ApplicationMaster implies that the ApplicationMaster by itself is rarely a bottleneck.

4.1.3. Discussion

Prior to the advent of YARN/Hadoop 2.0, scheduling of jobs through JobTracker/TaskTrackers was exclusively tied to the MR programming model: one could only run MR jobs over data stored in HDFS.

YARN separated the resource management layer (RM, NM, and containers) from the application layer (ApplicationMaster, etc.), paving the way for breaking the tie up between Hadoop and its MapReduce programming model. YARN allows for plugging in third-party ApplicationMasters. The ApplicationMaster itself is merely an instance of a framework-specific library, and encapsulates all application framework-specific code.

The resource management layer (RM, NM, and containers) are oblivious to the type of task. This makes it possible for an ApplicationMaster to run any type of task on a container, not just the Map and Reduce tasks supported by the erstwhile JobTracker/TaskTracker. For example, a graph-processing Giraph ApplicationMaster can run a Giraph task instead. If one so desires, one may write a new ApplicationMaster that runs whatever tasks that one feels necessary.

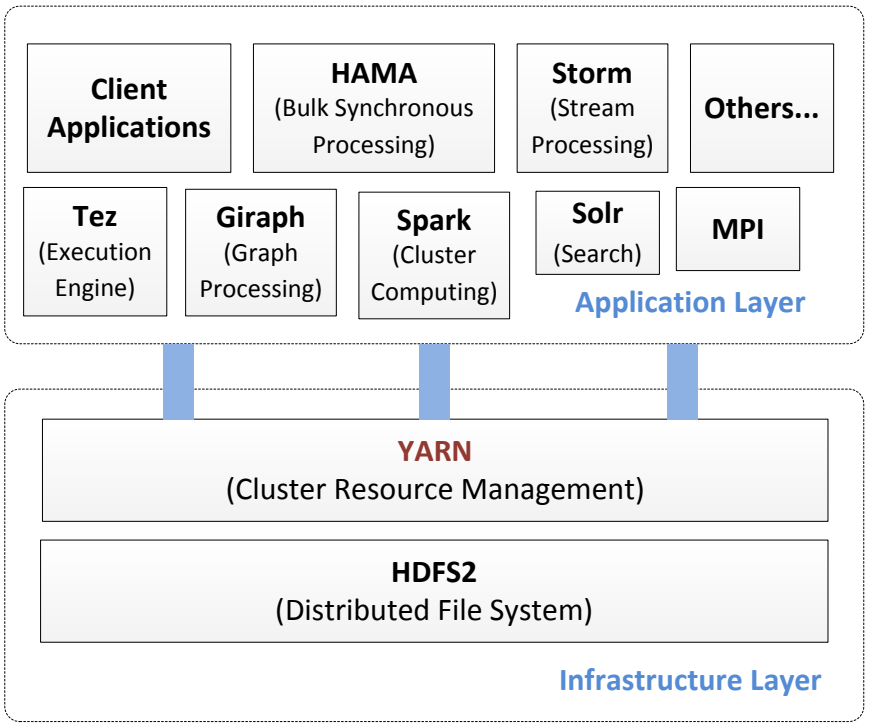

Figure 5: YARN's Role in the Hadoop Ecosystem

Even the erstwhile MapReduce in Hadoop is now just an application that runs on YARN. As such, YARN enables use of additional programming and computing models for processing data stored in Hadoop, as shown in the above figure.

4.2. Apache Spark: An Alternative Cluster computing Platform

4.2.1. Overview

Apache Spark was initially designed for interactive and iterative applications where keeping and processing data in-memory could make processing large data sets efficient, both with respect to latency of execution and development time. In fact, performance and ease of development remain its two key selling points to this day. We discuss these in greater detail in later sub-sections.

It has steadily evolved as a general purpose cluster computing platform that can run a variety of workloads - individually and in combination - within a single platform: these include batch applications, ad-hoc and interactive querying, iterative algorithms, graph processing algorithms, as well as stream processing. Moreover, the same API can be used to invoke any of those workloads, greatly simplifying application development.

Many see Apache Spark as a vast improvement over Hadoop MR and consider it the most likely successor of Hadoop. It started out as a research project at the University of California, AMP Lab, but is now a widely-used open source project with a lot of momentum as well as a thriving community with a growing list of contributors. Accordingly, Spark attained Apache Top-Level Project status in February 2014. All major vendors of Hadoop such as Cloudera, MapR, Hortonworks, Pivotal, and IBM bundle Apache Spark in their distributions - a clear recognition of its maturity and momentum.

4.2.2. Architecture

At its core, Apache Spark introduces two main parallel programming abstractions: Resilient Distributed Data Sets (RDD) and parallel operations over RDDs.

An RDD is a read-only, immutable and partitioned collection of objects, which can be stored in memory or disk. An RDD is automatically split into multiple partitions, and data represented by an RDD is automatically distributed across cluster nodes along the axis of partition. Operations performed on an RDD are automatically parallelized over the underlying data set along the partitioning axis.

All work is expressed as operations on RDD. Operations are of two types: transformations and actions. Transformations are used to create new RDDs from old ones. New RDDs can also be created by loading external files such as HDFS InputFormats. Examples of "transformation" operations include:

- map(<func>) – returns a "new data set that is formed by passing each element of the data set through function 'func' [Spark_Programming_Guide]"
- filter(<func>) – returns a *"new dataset formed by passing each element of the dataset through function func* [Spark_Programming_Guide]*"*.
- join()
- groupBy().

Actions are used to compute something from the RDD, the results of which are returned to the driver or stored on disk (such as HDFS). Examples of actions on RDDs include:

- reduce(<f>) – aggregates elements of the dataset using function f. According to the Spark Programming Guide, this *"function should be commutative and associative so that it can be computed correctly in parallel".*
- count() – returns count of elements in the dataset
- save() – saves a data set onto disk
- collect() – returns the elements of the data set as is
- foreach()

Programmers write programs that start with defining one or more RDD through transformations on data. These RDDs are then used in one or more actions arranged as a dataflow pipeline. Additionally, programmers can choose to persist the RDDs, say for use by other programs: RDDs marked for persistence are held in memory if enough memory is available; otherwise they are spilled over to the disk.

Spark also provides users control over certain aspects of data persistence and partitioning. For example, a user can choose whether an RDD is reused or whether to store the RDD in memory or on disk. Similarly, a user may specify that an RDD's data be partitioned based on a certain key present in each of the data items in the RDD. Refer to [Zaharia] for more details on the workings of the RDD.

In summary, Spark provides a general programming model that allows one to compose an application combining RDDs as higher-level abstractions of data sets and operators (like mappers, reducers, filters, group by's, foreach, and so on) that perform a set of actions on the RDDs.

4.2.3. Intersections of Spark and Hadoop

Like Hadoop, Apache Spark is a general-purpose cluster computing platform, and can operate on its own. It is also compatible with Hadoop in the following ways:

- It can access data sets via Hadoop Input Formats such as sequence files, Avro, Parquet, text files, etc., making it easy to transition from Hadoop ecosystem solutions [Wadkar].

- It can operate as an alternative to MR in Hadoop, running seamlessly over Hadoop 2.x YARN cluster, while leveraging Hadoop's resource management and data fabric (data over HDFS).

4.2.4. Discussion

Unlike Hadoop, Spark provides an abstraction (RDD) that allows users to leverage distributed memory for caching results across multiple computations (multiple jobs). Computations involved in interactive and exploratory querying, iterative processing, and graph processing, all benefit from reusing cached data: shared data through in-memory cache is much faster than sharing the same through disk writes, since writing to/reading from disks tend to be much slower than writing to and reading from memory.

5. Conclusion

Hadoop, along with its primary programming model MapReduce (MR), has remained the de-facto standard for processing large data sets for a long time. It is increasingly recognized that Hadoop and MR by themselves are ill-suited for several types of workloads in common use: interactive, ad-hoc and exploratory querying, iterative computations, graph processing, stream processing, and so on.

Recognizing that Hadoop MR only goes only so far, both open-source and commercial solutions have come up in the last few years, along interesting directions.

Some, like Apache Tez and Apache Giraph (non-YARN implementation) extend the Hadoop platform while continuing to follow the MR model, to meet efficiency needs of those workloads. Within this space, Apache Giraph takes the approach of implementing a specialized parallel computing model BSP for making graph processing efficient at scale.

Others like Apache Drill and Cloudera Impala bypass the MR model altogether. Impala focused its efforts around efficient querying over data resident in Hadoop, while Apache Drill is designed to work with multiple data sources such as Hadoop, NoSQL databases, and others.

HaLoop takes a different approach: it is a modified version of Hadoop and extends MR paradigm for use in iterative computations, which implies that changes made in Hadoop are not automatically reflected in HaLoop.

With YARN, Hadoop underwent a rather substantial overhaul of its core execution engine. YARN single-handedly opened up Hadoop for a diverse set of workloads and programming models beyond just batch-oriented MR applications. Splitting up the core execution engine and providing extension points for plugging in implementations of other programming models and execution engines allowed them use of the underlying Hadoop's resource management and data fabric.

Apache Spark – seen by many as the most likely successor of Hadoop – took yet another interesting direction of providing a more efficient and feature-rich cluster

computing platform that is seen as an alternative to Hadoop, while still complying with Hadoop environments via YARN.

Overall, these are interesting times for those of us involved in the Big Data processing world. With the “beyond Hadoop/MR” world taking several interesting directions, making technology choices isn’t expected to get any easier – at least any time soon. But having many choices for solving a given problem isn’t such a bad thing after all.

6. Acknowledgement

I gratefully acknowledge the contributions of David Broeckelman-Post in reviewing this article. David is the Chief Architect of the Advanced Security Operations Portfolio at RSA, The Security Division of EMC.

7. References

[Grolinger]	Grolinger, Katarina, et. al., “Challenges for MapReduce in Big Data”, Proc. of the IEEE 10th 2014 World Congress on Services (SERVICES 2014), Alaska, USA, June 27-July 2, 1014
[Metz]	Metz, Cade. “Open Source Superstar Rewrites Future of Big Data”, June 2013, retrieved from http://www.wired.com/2013/06/yahooamazonamplabspark/all/
[Leskovec]	Leskovec, Jure; Rajaraman, Anand; D. Ullman, Jeffrey. “Mining of Massive Datasets”
[Loshin]	David Loshin, Big Data analytics - From Strategic Planning to Enterprise Integration with Tools, Techniques, NoSQL and Graph, Morgan Kaufmann
[Jorgensen]	Jorgensen et al., Adam. Microsoft Big Data Solutions. John Wiley & Sons. © 2014
[FacebookScheduling]	Under the Hood: Scheduling MapReduce jobs more efficiently with Corona, https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920, Nov. 2012
[Olson_MRSpark]	Mike Olson, MapReduce and Spark, (Retrieved from http://vision.cloudera.com/mapreduce-spark/ in Jan. 2015)
[Elmeleegy]	Khaled Elmeleegy, Piranha: Optimizing Short Jobs in Hadoop, Proceedings of the VLDB Endowment, Vol. 6, No. 11, 2013
[Ekanayake]	Jaliya Ekanayake et. al., “Twister: A Runtime for Iterative MapReduce”, retrieved from http://www.iterativemapreduce.org/hpdc-camera-ready-submission.pdf on January 4th, 2015

[Lublinsky] http://www.infoq.com/articles/ApacheYARN Boris Lublinsky

[Murthy] Arun C. Murthy et. al., “Apache Hadoop YARN: Moving Beyond MapReduce and Batch processing with Apache Hadoop 2”, Addison-Wesley, 2014

[Wadkar] Wadkar, Sameer, Jason Venner, and Madhu Siddalingaiah. "Chapter 2 - Hadoop Concepts". Pro Apache Hadoop, Second Edition. Apress. 2014

[Agneeswaran] Vijay Srinivas Agneeswaran, Big Data Analytics Beyond Hadoop: Real-Time Applications with Storm, Spark, and More Hadoop Alternative

[Hausenblas] Michael Hausenblas and Jacques Nadeau, “Apache Drill: Interactive Ad-Hoc Analysis at Scale”, DOI: 10.1089/big.2013.0011

[Melnik] Sergey Melnik et. al., “Dremel: Interactive Analysis of WebScale Datasets”, Proceedings of the VLDB Endowment, Vol. 3, No. , 2010

[Rentachintala] Neeraja Rentachintala, “Building Highly Flexible, High Performance query engines - Highlights from Apache Drill project”, retrieved from http://www.slideshare.net/MapRTechnologies/apache-con-for-upload in Jan 2015

[Bevens] Bridget Bevens, “Performance” (in Architecture Highlights), retrieved from https://cwiki.apache.org/confluence/display/DRILL/Performance in Jan 2015

[Zaharia] Matei Zaharia et. al., “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for

In-Memory Cluster Computing”, retrieved from https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf in Jan 2015

[Spark_Programming_Guide] http://spark.apache.org/docs/latest/programming-guide.html

[Lin] Jimmy Lin and Chris Dyer, “Data-Intensive Text Processing with MapReduce”

[Olson] Mile Olson, http://vision.cloudera.com/mapreduce-spark/#sthash.oKNfd4BB.dpuf

[Bu] Yingyi Bu et. al., “HaLoop: Efficient Iterative Data Processing on Large Clusters”, VLDB Endowment, Vol. 3, No. 1, 2010

[Li] Feng Li, et. al., “Distributed Data Management Using MapReduce”, ACM Computing Surveys, Volume 46 Issue 3, January 2014

[Malewicz] Grzegorz Malewicz, et. al., “Pregel: A System for Large-Scale Graph Processing” SIGMOD’10, ACM

[BSP] “The BSP Programming Model” (retrieved from http://groups.csail.mit.edu/cag/bayanihan/papers/javapdc99/html/node2.html in Jan 2015)

[Hassan] M. Al Hajj Hassan, M. Bamha, “Parallel Processing of “Group-By Join” Queries on Shared Nothing Machines”, Software and Data Technologies, Communications in Computer and Information Science Volume 10, 2008, pp 230-241

[TezDesign] Tez Design v1.1, https://issues.apache.org/jira/browse/TEZ-65