Big Data Primer

Alex Sverdlov
alex@theparticle.com

1 Why Big Data?

Data has value. This immediately leads to: more data has more value, naturally causing datasets to grow rather large, even at small companies. Since storage space is cheap (and getting cheaper), smart companies realize that it is much more lucrative to store everything collected indefinitely. Even if no immediate use for the collected data exists, companies may find a use for it in the future—or perhaps sell it to someone who has a use for it.

Why is data valuable? Often it is business level information, such as purchases, customer interactions, deliveries, payrolls, web page visits, GPS location of phones, web logs, and 'like button' clicks. This data can be used to fit models to predict the future, to cluster and classify entities and events; the possibilities are endless. More data often leads to more accurate models and better features, and more data for cross validation. These models can improve business processes, attract new customers, offer better products, or just spy on everyone. In other words business can grow with a lot less guesswork.

Raw data has raw value. Processed data may have more information in a friendly format, but is not as valuable as raw data—is it often summarized and filtered. Since we cannot know what *might* be important in the future, it is critical to retain the *original raw data*. Processed data is not as valuable since it can always be recreated from the original raw data. For example, bank transactions are important (they indicate changes to accounts), the current balance is not as important, since it can be recalculated by aggregating over all transactions. That is not to say that current balance should not be present somewhere in the dataset, it means we should not replace transactional data with a single number.

Vast majority of companies operate without ever worrying about 'big data'. For example, a medical office that services a hundred patients per day will not have a problem maintaining all of the patient information on a single machine. Processing power and storage space wise, there is more than enough capacity even on a cheap machine. Granted they would do well to replicate the database in a few locations—even if just for safety. This is the realm of traditional databases, such as PostgreSQL.

'Big Data' is *more data than we can comfortably work with*. It obviously varies from place to place, but roughly anything that does not fit (space or processing wise) on one machine is 'big data'.

Many new and old companies are jumping on this 'big data' bandwagon and starting to collect everything they come across. It is not just the search engines or social networks. Often the big data attempts are misguided. We cannot think of big data in traditional database terms—we practically *never delete or update big data*. Collecting and storing 'big data' is not enough: we need to have a strong data science team to extract useful information. It is very easy to waste months of time and money on poor implementation of a 'big data strategy'. Sometimes however (a lot more rarely than anyone wants to admit), at very clever companies, big data reveals a key insight that allows one company to rise well above the competition. That is often reason enough for every company to try to get there first, even if there is no immediate payoff in sight.

2 Hadoop

We have operating systems to abstract away the hardware, and present a uniform interface to the file system. The concept of files is also an abstraction, but let's move past that. The idea is that we don't want to worry about how or what format lands on spinning rust or solid state disk. We just want to create a file and then write data to it.

For single machines, we have Linux, or other less popular choices. We can view distributed operating systems as another abstraction layer. We don't want to worry about what machine is processing or storing our data. We just want to create a file and then write data to it.

With that in mind, below sections present various components of a Hadoop distributed operating system, the leading open source package for Big Data stuff. There are alternatives, and Hadoop is not the best architecture for every problem—but it is there, and it has a great price of *free*.

3 HDFS

HDFS is Hadoop's file system. It abstracts away where and how files are stored—exactly like the file system in your operating system abstracts away where and how files are stored.

Simple view of HDFS is that it has two components: NameNode, and many DataNodes. The NameNode manages the directory tree, file names, locations. When you create a file, an entry is created in the NameNode. NameNode does not deal with the file contents, just the meta data.

The DataNodes manage the actual file contents, stored as blocks of a certain fixed size, such as 64MB. DataNodes don't know which file the data blocks belong to, they simply report back which blocks they contain to the NameNode. The NameNode maintains the association of blocks to files. Because blocks are rather large (often 64 or 128 MB), HDFS is not very efficient at storing large number of tiny files—it is best used to store a tiny number of huge files.

A typical "read a file" scenario is: client contacts the NameNode to find the data blocks and their locations for a particular file. The client then proceeds to contact the DataNodes

directly for the file contents.

A typical "write a file" scenario is: client contacts the NameNode to create the file. If the client machine has a DataNode instance running, then the data block is written to the local DataNode first (this has important implications for locality).

Each data block is replicated on at least three (by default) DataNodes. So simultaneous failure of *two* data nodes will not cause data loss, nor significantly impact data processing performance. Once the NameNode notices less than three replicas of a data block, it directs DataNodes to make more copies, so shortly after failure of a DataNode, there will again be three copies of every data block.

There is also a backup NameNode (in case the primary NameNode fails).

4 YARN

YARN is Hadoop 2.0. Before YARN, Hadoop had two major components: HDFS and MapReduce. If we wanted to do non-MapReduce work, we were stuck coercing our tasks into MapReduce. YARN is a resource manager—similar to how an operating system is a resource manager.

The resources YARN manages are compute cores. A map-reduce job allocates processing resources using YARM.

5 MapReduce

MapReduce is the most popular use case for Hadoop—and until YARN, it was the primary processing mechanism of Hadoop. The basic idea of MapReduce was developed by Google, and it boils down to decomposing algorithms into two functions: *map* and *reduce*.

Both *map* and *reduce* operate on key-value pairs. The usual map looks like:

```
map(key, value, context) {
    // ... process key/value into outkey/outvalue
    context.write(outkey, outvalue);
}
```

The mapper could do one-to-one output, or it could iterate over input key-value pair and generate zero or many results. The reducer is:

```
reduce(key, list_of_values, context) {
    // ... iterate over list_of_values and generate outkey/outvalue.
    context.write(outkey, outvalue);
}
```

The key part to notice is that the framework can have multiple map tasks running at the same time—since no map task needs to communicate with any other map task. In Hadoop, the map tasks are started on HDFS DataNodes, and they mostly operate on *local* data blocks

of that DataNode (so data does not need to be moved across the network). This is the key concept of moving processing code to the data, instead of the other way around.

Output of a map task (running on individual network nodes) gets split into N buckets (one for each reducer). The splitting can be automated or user controlled. Once each bucket reaches a certain size, it is transfered to the appropriate reducer. Since this is happening from all mappers to all reducers, this is called the "shuffle" operation.

The reducer gets data from many map tasks. As soon as it gets data, it sorts it (to make sure data is in the key order sequence). Once all mappers are complete, the reducer starts running, making sure that each call to reduce function gets the key, and all the values for that key. The output of the reducer goes back to HDFS (with each reducer writing an HDFS data block to its *local* DataNode; N reducers will often create N output files in HDFS, one for each reducer)

If at any time a mapper fails, the map-reduce framework can restart it on another machine—since HDFS keeps three copies of data blocks, a mapper can often start on another machine that houses a copy of the same data block.

If a reducer fails, the framework can restart it anywhere, and it will pull mapper results from all mappers. This may actually cause a mapper to restart (since the mappers and reducers often utilize the same cluster nodes).

Things to notice about map-reduce: it is batch data processing. It has high latency, but very high throughput. It requires writing mappers and reducers in a programming language, often in Java. It is very durable (it has very good reliability in failure situations).

In the best case scenario (no computational overhed, IO bound jobs), mappers operate at sequential disk read speeds (often around 100MB per second per node for spinning disks). The shuffle operates at network speeds, which are often at least gigabit speeds (about 100MB per second per node). Reducers operate at disk write speeds. File formats must be setup to allow for these throughput rates—do not compress data with bzip2 unless decompression can happen at 100MB/second.

6 Hive

Not long after Hadoop became popular, Facebook created Hive, a SQL layer on top of Hadoop. The initial versions translated SQL into a series of map-reduce tasks. For example:

```
select * from whatever where somefield = 123;
```

These are translated into a map task that would filter on the condition, and pass-through reduce task (reduce task that does nothing). Queries that have aggregates would utilize the reduce task to calculate aggregates.

Joins of small-table-to-large table are handled as map-side-joins: the map task reads the smaller table, hashes it, and then iterates over the larger table implementing the join.

Joins of two large tables are implemented as co-located joins: the map task outputs both tables on the same key, the reduce task gets records from both tables and joins them (this is often slower than map-side-join).

Hive is not a real-time system, and is still firmly batch mode. It is best viewed as a SQL interface to MapReduce. Nobody should write MapReduce jobs by hand these days—SQL is a much better alternative.

To speed up Hive, there is Tez execution engine. Instead of translating SQL into a series of MapReduce jobs, Tez translates SQL into a DAG (directed graph). Each node does some processing, and edges can be implemented as network pipes. The key feature though is that Tez does not store intermediate results in HDFS. For example, if a query requires two MapReduce steps, the old mechanism would persist the intermediate results in HDFS (each map-reduce phrase reads from HDFS and saves data in HDFS). Tez keeps intermediate results in memory or temporary files on disk—this makes Tez a bit more brittle for long running jobs.

Latest Hive version has support for advanced SQL features, including analytical (windowing) functions. There is still lack of support for many 'standard' SQL things, such as sequences, though that will likely change in the near future. Latest iterations are adding transactional features.

7 Spark

The basic MapReduce mechanism operates on HDFS files and produces other HDFS files. This can be slow, since it involves disk.

Spark is essentially a framework that abstracts away memory objects in the same way that Hadoop abstracts away HDFS files. With Spark, we can create an RDD (Resilient Distributed Dataset) and then do operations on it, with strong preference of keeping the RDD in memory. The operations are very similar to what map-reduce jobs do—except they operate on memory objects (and not HDFS files).

RDDs are result sets that are sharded across multiple machines. Reliability is achieved by maintaining historical lineage for each RDD, so it can be recalculated if that is ever required. They can also be persisted to local disk, or to HDFS.

RDDs are immutable, so not all memory heavy processing tasks are applicable for Spark. For example, we cannot maintain a hash table as an RDD, since we wouldn't be able to alter it without making another copy of the whole RDD.

Many map-reduce tasks are *much* faster when rewritten using Spark. The Spark API supports joins between RDDs, filtering, sorting, and many other SQL-like capabilities. The primary mechanism to interact with RDDs is via Scala or Java. There are SQL extentions to Spark, but they're nowhere near Hive SQL capability. Spark can be deployed as part of Hadoop or standalone (on its own cluster).

8 HBase

Hadoop and MapReduce were built as an efficient batch processing mechanism for big data. They were never meant for real-time data reads/writes. HBase is a real time key-value

database that is independent of map-reduce. HBase often uses HDFS as a backend to store data files, though it is capable of using non-HDFS file systems as well.

HBase stores key-values. This is a bit simplistic. There are *tables*, and each table has *column families*. A row-key uniquely identifies a record within a table across multiple column families. HBase stores several versions of the value for each row key. The values within the same column family are stored together—it is very efficient to retrieve same column families for subsequent row keys.

The entire space of keys is partitioned into disjoint key regions, naturally hosted on separate region servers. To add a key-value (for a particular column family) to HBase, the client figures out which region will get the key. The region server then writes the key to a write-ahead log (append only file hosted on HDFS), and adds it to the memory store. From that point forward, that region server will respond correctly to queries regarding that key.

Once the memory store fills up to a pre-configured size, it is flushed to HDFS in key-sorted-order, followed by an index for a file footer. The files are blocky (with 64kb block size being common); the end-of-file index only needs to point to the correct block. Each column family is stored separately.

If a failure occurs after the flush, then HDFS ensures no data gets lost. If region server fails before flushing memory to HDFS, during the next startup the write-ahead log (hosted on HDFS) is used to recreate the region server memory store.

When region grows beyond a certain size, it is split into two. Similarly, small regions are merged if they become too small.

All updates (new keys inserted, or removed), are added first to write-ahead log, then to memory store. Occasionally the updates are merged with the disk data by a complete rewrite of the HDFS file—potentially compacting old files or splitting them. Deletes create a tombstone marker for a particular key, but do not actually remove the key from the disk until the next data rewrite.

Note that all these reads/writes are mostly happening on the *local* HDFS blocks.

9 MPP

MPP (Massively Parallel Processing) databases are another way of working with big data. These include databases such as Netezza and Greenplum. These are multi-node clusters designed to run database software, and not much of anything else.

The primary feature is distribution of data—data is sharded across all the nodes, often on a distribution key. Locality of data is maintained by distribution key—so joins that contain distribution keys can be done locally without any data movements.

Beyond this data locality aspect, the data processing mechanisms are very similar to MapReduce. We can filter records in a table (Map operation), aggregate (Reduce operation), and do joins (map side or reduce side joins).

MPP databases often have big corporations behind them, with amazing SQL support (great optimization, full support for analytical functions and other SQL things). They're often quite expensive, especially compared to *free* Hive.

10 HAWQ

Pivotal, the company behind Greenplum has a hybrid database named HAWQ. It is essentially Greenplum with an HDFS back end—similar architecture to Hive, except with a bit more performance—mostly due to Greenplum's high performance guts compared to Hive's Java implementation.

11 Hardware

It is a common misconception that Hadoop requires expensive big servers. It thrives on many cheap machines. Instead of spending $10k per node, it is best to get 10 nodes for $1k each—instead of having a cluster of 40 machines, it is better to have a cluster of 400 machines (or 4000!).

Keep in mind what matters: IO speed, and CPU speed. 100 machines with 100 disks and 100 CPUs are almost always faster (more IO and CPU) than more expensive 10 machines. That said, is it still a great idea to max out the RAM, and depending on needs, go with solid state disks.

12 Cloud, AWS & Others

Hadoop is a great 'distributed operating system' for big data processing. It does require a cluster of many machines—often hundreds. For many companies, maintaining a large local cluster is cost prohibitive—especially if processing is only needed periodically. It is very expensive to maintain a data center presence with hundreds of machines.

For such situations there's Amazon EC2 (Elastic Compute Cloud)—rent as many machines as needed, and pay by the hour. There are various pricing scales, and an auction process, resulting in fairly cheap computational resources.

Security can often be achieved by encrypting data in flight (for example, use SSH or SSL for data transfers), and encrypting resting data (encrypt the local EC2 disk, and throw away the encryption key). The EC2 machine is a blank slate as soon as it shuts down—and there is no danger of the powered off disks showing up anywhere.

Amazon also has S3 (Simple Storage Service) which is a key-value store for big data—best way to view it is as a file system. Data in S3 can be encrypted by customers or Amazon (with Amazon managed keys).

AWS provides mechanisms to easily process S3 data using EC2 instances. For example, Hive can create external tables using S3 objects.

Because customers don't know exactly where (nor how, nor how many copies) their data is stored, nor where (exactly which physical machines) are processing their data—it is stored "out there somewhere" processed by clusters of "virtual machines" that they rent by the hour; this kind of arrangement is often referred to as "the cloud".

The Cloud solves many traditional IT problems, such as maintaining a data center, but it also often creates its own set of new problems. A lot of the data management, administration and development tasks are quite a bit more complicated and require a lot more effort—not less.

Amazon is not the only provider—it just happens to be the more popular one at the moment. There are offerings from Google, Rackspace and Microsoft, and surely many others, all worth exploring.