PERFORMANCE ENHANCEMENT OF BIG DATA PROCESSING IN HADOOP MAP/REDUCE

A report submitted in partial fulfillment of the requirements for the award of the degree of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

by

PRATEEK GARG
(133050093)

under the guidance of

Prof. D. B. PHATAK

COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
MUMBAI

October, 2014
1. INTRODUCTION

Large scale data analytics has become indispensable as business enterprises need to gain actionable insights from their increasing volumes of data. The explosion of data is being experienced by every sector of the computing industry today. Internet giants such as Google, Amazon, Yahoo!, Facebook and the like have to deal with huge amounts of user generated data in the form of blog posts, photographs, status messages, and audio/video files. There is also a large quantity of data that is indirectly generated by web sites in the form of access log files, click through events etc. Analysis of this data can uncover useful patterns about user behavior. However, analyzing the flood of data has become one of the greatest challenges of recent times, with traditional approaches including relational databases and data warehouses failing to match the scale of data. The current explosion of data collection is increasing at a faster rate than Moore’s law, indicating that data management and analysis will only become more difficult in the future.

Scalable log processing is a crucial facility to ensure efficient use of infrastructure and gain a competitive advantage. Service providers continuously monitor many aspects of their system using semi-structured log data. Increased revenue can be generated by analyzing logs of user click streams for behavioral patterns to provide targeted ads. E-commerce and credit card companies analyze point-of-sales transactions for fraud detection. Infrastructure providers use log data to detect hardware misconfigurations, improve load-balancing across large data centers, and gather statistics about infrastructure usage patterns [18,19].

The emergence of Google’s MapReduce paradigm and its open source implementation Hadoop provide enterprises with a cost effective solution for the analytic needs such as described above. Hadoop is a framework that supports data intensive parallel applications, working with thousands of compute nodes and petabytes of data. In Hadoop, map and reduce functions operate on data stored in HDFS (Hadoop Distributed File System) files. HDFS stores large files as a series of blocks distributed over a cluster of data nodes and takes care of fault tolerance through replication. The data placement policy of Hadoop tries to balance load by placing blocks
randomly; it does not take any data characteristics into account. In particular, HDFS does not provide any means to place relevant data on the same set of nodes.

Also, real time data processing challenges are complex. BigData is commonly characterized by volume, velocity, and variety of the data, and Hadoop like systems are efficient to handle the volume and variety aspect. Along with the volume and variety, the real time system needs to handle the velocity of the data as well. And handling the velocity of Big Data is not an easy task. First, the system should be able to collect the data generated by real time events streams coming in at a rate of millions of events per seconds. Second, it needs to handle the parallel processing of this data as and when it is being collected. Third, it should perform event correlation using a Complex Event Processing engine to extract the meaningful information from this moving stream. These three steps should happen in a fault tolerant and distributed fashion. The real time system should be a low latency system so that the computation can happen with near real time response capabilities.

Hadoop being a strictly batch oriented data processing platform faces the above mentioned two shortcomings for real time data processing.

This report investigates the performance bottleneck created by the default HDFS data placement policy and proposes an extension to Hadoop that allows applications to control where data is stored. Also, to handle velocity of massive amount of data coming in, an integration with Storm forms the front end of the final system.
2. BACKGROUND & RELATED WORK

This chapter gives an overview of the technologies that are used in this work with a purpose of providing a basic and common background to ease the understanding of the rest of the document. Section 2.1 gives a description of the original MapReduce programming model. Section 2.2 describes Hadoop, the current state of the art open source MapReduce framework which this work extends. Section 2.3 gives an overview of Hadoop Distributed File System. Finally, section 2.4 briefly describes related stream processing systems.

2.1 MapReduce

2.1.1 What is Map/Reduce?

Map/Reduce [4] is a “programming model and an associated implementation for processing and generating large data sets”. It is designed to simplify the parallelization of application logic enabling developers to easily leverage clustered resources. MapReduce is an imperative programming model, and is based on the common map and fold functions found in languages like LISP. The model makes it easy for a programmer to implement desired functionality without having to worry about complications like fault tolerance, load balancing, data distribution and task parallelization in a distributed environment.

MapReduce is a linearly scalable programming model. The programmer writes two functions—a map function and a reduce function—which each defines a mapping from one set of key-value pairs to another. These functions are oblivious to the size of the data or the cluster that they are operating on. More important, if the size of the input is doubled, a job will run twice as slow. But if at the same time the size of the clusters is also made double, a job will run as fast as the original one. This is not generally true of SQL queries.

Some of the simple and interesting computations for which Map/Reduce can be used include (see [9] for details on each):

- Distributed Grep - finding patterns in a number of files at the same time.
- Count of URL access frequency.
• Reverse Web-Link Graph - Given a list of \langle target, source \rangle pair of URLs, finding \langle target, list(source) \rangle, i.e., finding all the URLs that link to a given target.
• Construction of Inverted Indices from crawled web pages
• Distributed Sort

The Map/Reduce framework runs multiple instances of map and reduce functions in parallel. The map function processes a key/value pair to generate another key/value pair. A number of such map functions running in parallel on the data that is partitioned across the cluster, produce a set of intermediate key/value pairs. The reduce function then merges all intermediate values that are associated with the same intermediate key

\[
\text{map} \ ( k1, v1 ) \rightarrow k2, v2 \\
\text{reduce} \ ( k2, \text{list}( v2 ) ) \rightarrow v3
\]

where \( k1 \) and \( k2 \) denote keys; \( v1, v2 \) and \( v3 \) denote values.

Figure : Map/Reduce Execution Model [4]
2.1.2 Map/Reduce Example

Map/Reduce is better explained with an example below. Given below are the map and reduce function for categorizing a set of numbers as even or odd.

```java
map(String key, Integer values) {
    // key: File name
    // value: list of numbers
    for each word v in values:
        if(v%2 == 0)
            EmitIntermediate("Even", v);
        else
            EmitIntermediate("Odd", v);
}

reduce(String key, Iterator values) {
    // key: Even or Odd
    // value: iterator over list of numbers
    String val
    while(values.hasNext())
        val += values.toString();
    Emit(key, val);
}
```

Figure: Map and Reduce Function examples.

So given a list of numbers as 5,4,2,1,3,6,8,7,9, the final output file would be:

Even 2,4,6,8
Odd 1,3,5,7,9

This is a very simple example where both the map and reduce function do not do anything much interesting. But a programmer has the freedom to write something a lot more complex than these functions.

2.2 Hadoop

Hadoop [12] is the Apache Software Foundation open source and Java-based implementation of the Map/Reduce framework.
Hadoop provides the tools for processing vast amounts of data using the Map/Reduce framework and, additionally, implements the Hadoop Distributed File System (HDFS). Consequently, it renders the advantages of the Map/Reduce available to the users.

Hadoop MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions: the map function and the reduce function.

In Hadoop, a MapReduce job is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks. There are two types of nodes that control the job execution process: a jobtracker and a number of tasktrackers. The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker.

Hadoop divides the input to a MapReduce job into fixed-size pieces called splits or chunks. Hadoop creates one map task for each split, which runs the user defined map function for each record in the split.

Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if the splits are processed in parallel, the processing is better load-balanced if the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine-grained. On the other hand, if splits are too small, then the overhead of managing the splits and of map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of an HDFS block, 64 MB by default, although this can be changed for the cluster (for all newly created files), or specified when each file is created.
Sometimes, all three (default number of replicas of a split) nodes hosting the HDFS block replicas for a map task’s input split are running other map tasks so the job scheduler has to look for a free map slot on a node in the same rack as one of the blocks. Very occasionally even this is not possible, so an off-rack node is used, which results in an inter-rack network transfer.

It should now be clear why the optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node. If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so some of the split would have to be transferred across the network to the node running the map task, which is clearly less efficient than running the whole map task using local data.

Map tasks write their output to the local disk, not to HDFS. This is because map output is intermediate output: it’s processed by reduce tasks to produce the final output, and once the job is complete the map output can be discarded. So storing it in HDFS, with replication, would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop automatically reruns the map task on another node to generate the map output.

Reduce tasks don’t have the advantage of data locality—the input to a single reduce task is normally the output from all mappers. The output of the reduce is normally stored in HDFS for reliability. For each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes.

The number of reduce tasks is not governed by the size of the input, but is specified independently. When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

The data flow for the general case of multiple reduce tasks is illustrated in figure below. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as “the shuffle,” as each reduce task is fed by many map tasks. The shuffle is more
complicated than this diagram suggests, and tuning it can have a big impact on job execution time.

![Data Flow representation for a MapReduce Task](image)

Figure: Data Flow representation for a MapReduce Task

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a combiner function to be run on the map output—the combiner function’s output forms the input to the reduce function. Since the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer.

The contract for the combiner function constrains the type of function that may be used. This is best illustrated with an example. Suppose that for a given metrological data over a span of years, the maximum temperature for each year is to be found. Readings for the year 1950 were processed by two maps (because they were in different splits). Imagine the first map produced the output:

\[(1950, 0) \ (1950, 20) \ (1950, 10)\]

And the second produced:

\[(1950, 25) \ (1950, 15)\]
The reduce function would be called with a list of all the values:

\((1950, [0, 20, 10, 25, 15])\)

with output:

\((1950, 25)\)

since 25 is the maximum value in the list. A combiner function could be used that, just like the reduce function, finds the maximum temperature for each map output independently. The reduce would then be called with:

\((1950, [20, 25])\)

and the reduce would produce the same output as before. More succinctly, the function called on the temperature values in this case may be expressed as:

\(\max(0, 20, 10, 25, 15) = \max(\max(0, 20, 10), \max(25, 15)) = \max(20, 25) = 25\)

Not all functions possess this property. For example, if mean was to be calculated, then the combiner function could not be used, since:

\(\text{mean}(0, 20, 10, 25, 15) = 14\)

but:

\(\text{mean}(\text{mean}(0, 20, 10), \text{mean}(25, 15)) = \text{mean}(10, 20) = 15\)

### 2.3 Hadoop Distributed File System

Hadoop includes a distributed filesystem called *Hadoop Distributed Filesystem* (HDFS). HDFS is Hadoop’s flagship filesystem and is the focus of this section, but Hadoop actually has a general-purpose filesystem abstraction, allowing users to integrate Hadoop with other storage systems.

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware. The statement can be analyzed further as:

- Very large files: ‘Very large’ in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size. There are Hadoop clusters running today that store petabytes of data.
• Streaming data access: HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

• Commodity hardware: Hadoop doesn’t require expensive, highly reliable hardware to run on. It’s designed to run on clusters of commodity hardware for which the chance of node failure across the cluster is not low, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.

The following are some applications for which HDFS does not work so well. While this may change in the future, these are areas where HDFS is not a good fit at present:

• Low-latency data access: Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency.

• Lots of small files: Since the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode. As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if the system deal with one million files, each taking one block, it would need at least 300 MB of memory. While storing millions of files is feasible, billions is beyond the capability of current hardware.

• Multiple writers, arbitrary file modifications: Files in HDFS may be written to by a single writer. Writes are always made at the end of the file. There is no support for multiple writers, or for modifications at arbitrary offsets in the file.

2.3.1 Hadoop Concepts

2.3.1.1 Blocks

A disk has a block size, which is the minimum amount of data that it can read or write. Filesystems for a single disk build on this by dealing with data in blocks, which are an integral
multiple of the disk block size. Filesystem blocks are typically a few kilobytes in size, while disk blocks are normally 512 bytes.

HDFS, too, has the concept of a block, but it is a much larger unit—64 MB by default. Like in a filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block’s worth of underlying storage.

2.3.1.2 Why is a block in HDFS so large?

HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. By making a block large enough, the time to transfer the data from the disk can be made to be significantly larger than the time to seek to the start of the block. Thus the time to transfer a large file made of multiple blocks operates at the disk transfer rate.

Having a block abstraction for a distributed filesystem brings several benefits. The first benefit is the most obvious: a file can be larger than any single disk in the network. There’s nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster. In fact, it would be possible, if unusual, to store a single file on an HDFS cluster whose blocks filled all the disks in the cluster.

Second, making the unit of abstraction a block rather than a file simplifies the storage subsystem. Simplicity is something to strive for a distributed system in which the failure modes are so varied.

Furthermore, blocks fit well with replication for providing fault tolerance and availability. To insure against corrupted blocks and disk and machine failure, each block is replicated to a small number of physically separate machines (typically three). If a block becomes unavailable, a copy can be read from another location in a way that is transparent to the client. A block that is no longer available due to corruption or machine failure can be replicated from its alternative locations to other live machines to bring the replication factor back to the normal level. Similarly, some applications may choose to set a high replication factor for the blocks in a popular file to spread the read load on the cluster.
2.3.1.3 Namenodes and Datanodes

An HDFS cluster has two types of nodes operating in a master-worker pattern: a namenode (the master) and a number of datanodes (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree using two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts.

Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

Without the namenode, the filesystem cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes. For this reason, it is important to make the namenode resilient to failure, and Hadoop provides two mechanisms for this.

The first way is to back up the files that make up the persistent state of the filesystem metadata. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. These writes are synchronous and atomic.

It is also possible to run a secondary namenode, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge. It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode’s metadata files that are on NFS to the secondary and run it as the new primary.
2.3.1.4 HDFS high availability

The combination of replicating namenode metadata on multiple filesystems, and using the secondary namenode to create checkpoints protects against data loss, but does not provide high-availability of the filesystem. The namenode is still a single point of failure (SPOF), since if it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping. In such an event the whole Hadoop system would effectively be out of service until a new namenode could be brought online.

To recover from a failed namenode in this situation, a new primary namenode is started with one of the filesystem metadata replicas, and configures datanodes and clients to use this new namenode. The new namenode is not able to serve requests until it has i) loaded its namespace image into memory, ii) replayed its edit log, and iii) received enough block reports from the datanodes to leave safe mode.

The long recovery time is a problem for routine maintenance too. In fact, since unexpected failure of the namenode is so rare, the case for planned downtime is actually more important in practice.

2.4 Real Time Stream Processing

This section gives an insight into one of the most commonly used real time stream processing framework: Apache Storm.

Apache Storm is an open source engine which can process data in realtime using its distributed architecture while being fault tolerant. Storm is simple and flexible. It can also be used with any programming language of the application developers choice.

Storm implements a set of characteristics that define it in terms of performance and reliability. Storm uses ZeroMQ for message passing, which removes intermediate queueing and allows messages to flow directly between the tasks themselves. Under the covers of messaging is an automated and efficient mechanism for serialization and deserialization to Storm's primitive types.
What makes Storm most interesting is its focus on fault tolerance and management. Storm implements guaranteed message processing such that each tuple is fully processed through the topology; if a tuple is discovered not to have been processed, it is automatically replayed from the spout. Storm also implements fault detection at the task level, where upon failure of a task, messages are automatically reassigned to quickly restart processing. Storm includes more intelligent process management than Hadoop, where processes are managed by supervisors to ensure that resources are adequately used.

A Storm cluster consists of the following components:

1. Nimbus Node: The master node (similar to JobTracker in Hadoop)
2. Supervisor Nodes: Start/Stop workers & communicate with Nimbus through ZooKeeper
3. ZooKeeper Nodes: Coordinates the storm cluster.

The following discussion is based on the following terminology:

1. Tuples: An ordered list of elements e.g a ‘4-tuple’ might be (7,1,3,7).
2. Streams: The stream is the core abstraction in Storm. A stream is an unbounded sequence of tuples that is processed and created in parallel in a distributed fashion. Streams are defined with a schema that names the fields in the stream’s tuples.
3. Spouts: A spout is a source of streams in a topology. Generally spouts read tuples from an external source and emit them into the topology. Spouts can either be reliable or unreliable. A reliable spout is capable of replaying a tuple if it failed to be processed by Storm, whereas an unreliable spout forgets about the tuple as soon as it is emitted. Spouts can emit more than one stream.
4. Bolts: All processing in topologies is done in bolts. Bolts can do anything from filtering, functions, aggregations, joins, talking to databases, and more. Bolts can do simple stream transformations. Doing complex stream transformations often requires multiple steps and thus multiple bolts.
5. Topologies: The logic for a realtime application is packaged into a Storm topology. A Storm topology is analogous to a MapReduce job. One key difference is that a MapReduce job eventually finishes, whereas a topology runs forever (or until it is killed). A topology is a graph of spouts and bolts that are connected with stream groupings.
2.4.1 The Storm Model

Storm implements a data flow model in which data flows continuously through a network of transformation entities (see figure below). The abstraction for a data flow is called a stream, which is an unbounded sequence of tuples. The tuple is like a structure that can represent standard data types (such as ints, floats, and byte arrays) or user-defined types with some additional serialization code. Each stream is defined by a unique ID that can be used to build topologies of data sources and sinks. Streams originate from spouts, which flow data from external sources into the Storm topology.

Figure: Implementing MapReduce using Storm.

Figure: Data Flow model used by Storm.
The sinks (or bolts) implement a single transformation on a stream and all processing within a Storm topology. Bolts can implement traditional things like MapReduce functionality or more complex actions (single-step functions) like filtering, aggregations, or communication with external entities such as a database. A typical Storm topology implements multiple transformations and therefore requires multiple bolts with independent tuple streams.

Storm can be easily used to implement MapReduce functionality for word frequency. As shown in figure below, a spout generates the stream of textual data, and a bolt implements the Map function (to tokenize the words of a stream). The resulting stream from the "map" bolt then flows into a single bolt that implements the Reduce function (to aggregate the words into counts).

Note that bolts can stream data to multiple bolts as well as accept data from multiple sources. Storm has the concept of stream groupings, which implement shuffling (random but equal distribution of tuples to bolts) or field grouping (stream partitioning based upon the fields of the stream). Other stream groupings exist, including the ability for the producer to route tuples using its own internal logic.

But one of the most interesting features in the Storm architecture is the concept of guaranteed message processing. The Storm framework provides reliability by ensuring that every spout tuple will be fully processed by the topology. It does this by tracking the tree of tuples triggered by every spout tuple and determining when that tree of tales has been successfully completed. Every topology has a “message timeout” associated with it. If Storm fails to detect that a spout tuple has been completed within that timeout, then it fails the tuple and replays it later.

In addition to supporting reliable messaging, Storm uses ZeroMQ to maximize messaging performance (removing intermediate queueing and implementing direct passing of messages between tasks). ZeroMQ incorporates congestion detection and alters its communication to optimize the available bandwidth.
Stream Groupings

A stream grouping tells a topology how to send tuples between two components. Remember, spouts and bolts execute in parallel as many tasks across the cluster. A topology executing at the task level looks something like this:

![Executing topology in Storm Framework](image)

When a task for Bolt ‘A’ emits a tuple to Bolt ‘B’, which task should it send the tuple to? A ‘Stream Grouping’ answers this question by telling Storm how to send tuples between set of tasks. There’s a few different kinds of stream groupings.

1. **Shuffle Grouping**: Tuples are randomly distributed across the bolt’s tasks in a way such that each bolt is guaranteed to get an equal number of tuples.

2. **Fields Grouping**: The stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by the “user-id” field, tuples with the same “user-id” will always go to the same task, but tuples with different “user-id”’s may go to different tasks.

3. **All Grouping**: The stream is replicated across all the bolt’s tasks. It is to be used with care.

4. **Global Grouping**: The entire stream goes to a single one of the bolt’s tasks. Specifically it goes to the task with the lowest id.

5. **None Grouping**: This grouping specifies that it is not bothered how the stream is grouped. Currently, none groupings are equivalent to shuffle groupings. Eventually though, Storm will
push down bolts with none groupings to execute in the same thread as the bolt or spout they subscribe from (when possible).

6. Direct grouping: This is a special kind of grouping. A stream grouped this way means that the producer of the tuple decides which task of the consumer will receive this tuple. Direct groupings can only be declared on streams that have been declared as direct streams.

7. Local or shuffle grouping: If the target bolt has one or more tasks in the same worker process, tuples will be shuffled to just those in-process tasks. Otherwise, this acts like a normal shuffle grouping.

2.5 Related Work

Some of the earlier work [10] related to the proposed system does not support co-placement of data at all, while others [1, 6] perform heavy-weight changes to co-place the related data partitions: HadoopDB [1] stores the data in a local DBMS and hence disrupts the dynamic scheduling and fault tolerance of Hadoop.

Hadoop++ [6] co-groups the two input files by creating a special “Trojan” file. Although this approach does not require a modification of Hadoop, it is a static solution that requires users to reorganize their input data. In fact, Hadoop++ can only colocate two files that are created by the same job, and requires reorganization of the data as new files are ingested into the system. In applications such as log processing, where data arrive incrementally and continuously, it is important (1) to co-place many files, not just two, and (2) to co-place newly ingested files incrementally with existing data.

In comparison with Hadoop++ [6], which is the closest to this work, in that it also exploits data pre-partitioning and colocation. In this work, focus is emphasized on only the join query since it is the one considered in [6]. The HDFS block size is set to 256MB.

Recent benchmarks have identified a performance gap between Hadoop and parallel databases [14, 10]. There has been considerable interest [1, 6, 10] in enriching Hadoop with techniques from parallel databases, while retaining Hadoop’s flexibility. Jiang et al. [10] conducted an intensive benchmark of various parts of Hadoop’s processing pipeline. It was found that (among others) indexing and map-side “partition joins” can greatly improve Hadoop’s
performance. In contrast to this extension work, they do not co-place partitioned data fragments. HadoopDB [1] and Hadoop++ [6] are the closest to this proposal in spirit because they also try to co-place data. However, they change the physical layout: HadoopDB replaces HDFS by full-fledged relational databases, whereas Hadoop++ injects indexes and co-partitiones data directly into raw data files. HadoopDB benefits from the power of DBMSs in query optimization and use of indexes at the cost of breaking the programming model and simplicity of MapReduce; it can be viewed as “another parallel database” [6]. Hadoop++ is less intrusive: co-placed data (such as indexes and co-partitions for joins) are stored as “Trojans” within HDFS files and splits; no changes to Hadoop itself are required. In contrast to the proposed approach, however, co-placement in Hadoop++ is static and done at load time: any change of desired indexes, co-partitioning, or even arrival of new data forces Hadoop++ to re-organize the entire dataset. Moreover, their colocation is geared toward joins and hence they can only colocate two files, whereas this approach is pretty flexible in terms of the queries it can support, and number of files it can co-place.

Cheetah [3] and Hive [19] are two data warehousing solutions on Hadoop, and borrow many ideas from parallel databases. But, neither supports co-placement and its exploitation. GridBatch [11] is another extension to Hadoop with several new operators, as well as a new file type, which is partitioned by a user-defined partitioning function. GridBatch allows applications to specify files that need to be co-placed as well. Their solution intermixes partitioning and colocation at the file system level, whereas this method decouples them so that different applications can use different methods to define related files.

The proposed approach is heavily inspired by the more advanced partitioning features of parallel database systems [20], such as IBM DB2, TeraData, Aster Data. In these systems, tables are co-partitioned, and the query optimizer exploits this fact to generate efficient query plans [7]. This approach adapts these ideas to the MapReduce infrastructure, while retaining Hadoop’s dynamicity and flexibility. To achieve this, proposed approach differs from parallel databases in that proposed system performs co-placement at the file-system level and in a best-effort manner: When space constraints or failures prevent co-placement, high availability and fault tolerance are given higher priority.
3. PROBLEM STATEMENT

Consider the figure above which depicts a timeline in the course of a Hadoop job. Also, consider that, at present, the system is at the position marked by “Now”. As shown in the figure, the Hadoop job takes quite long; at present the Hadoop is still in process and hence incomplete. The amount of data that has actually been processed till now is shown by the green line while the amount of data yet to be processed is depicted using the red line. Clearly, the amount of data that is yet to be processed is greater than a full period interval.

The point being emphasized here is that Hadoop is not very real time in nature. Also, due to the batch oriented nature of Hadoop jobs, it is unable to process high velocity incoming data in near real time.

4. PROPOSED APPROACH

Specific to the case of log processing, data are accumulated in batches from event logs, such as clickstreams, phone call records, application logs or a sequence of transactions. Each batch of data is ingested into Hadoop and stored in one or more HDFS files at regular intervals. Two common operations in log analysis are 1) joining the log data with some reference data, and 2) sessionization, i.e. analyzing user sessions. Join performance can be improved by co-partitioning the input files. In these cases, the performance gain is due to the elimination of data
shuffling and the reduce phase of MapReduce. With co-placed data partitions, joins can be executed using a map-only join algorithm with no remote I/O. Some of the earlier work does not support co-placement at all, while others [1,6] perform heavy weight changes to co-place the related data partitions.

In contrast to previous approaches, the proposed system decouples the problem of co-placing related files from the applications that exploit this property. The extension requires minimal changes to HDFS. A new file property to identify related data files and modify the data placement policy of HDFS to co-place all copies of those related files is introduced. These changes retain the benefits of Hadoop, including load balancing and fault tolerance. The decoupling also enables a wide variety of applications to exploit data co-placement by simply specifying related files.

The contributions of this extension can be summarized as follows:
1. A flexible, dynamic and light weight approach to co-place related data files, which is implemented directly in HDFS is proposed.
2. Identification of two use cases in log processing, i.e. join and sessionization, where co-partitioning files and co-placing them speeds up the query processing significantly.
3. Fault tolerance, data distribution and data loss properties of the proposed system are studied using a stochastic model.

The objective of HDFS default data placement policy is to achieve load balancing by distributing the data evenly across the datanodes, independently of the intended use of data. This simple data placement policy works well with most Hadoop applications that access just a single file, but applications that process data from different files can get a significant boost in performance with customized strategies. The proposed extension allows applications to easily define and exploit such customized strategies.

Research in parallel databases [20] as well as recent work in Hadoop have shown that careful data organization enables efficient algorithms for query processing. Although partitioning is easy to achieve in Hadoop, co-placement is not. This is because Hadoop provides no
principled way for applications to control where the data is stored\textsuperscript{1}. Without co-placement data shuffling costs and network overhead reduces the effectiveness of partitioning. To overcome this and related problems, the proposed extension provides a generic mechanism that allows applications to control data placement at the file system level, if needed.

To achieve co-placement, HDFS is extended with a new file level property called a \textbf{locator}, and Hadoop’s data placement policy is modified so that it makes use of this locator property. Each locator may be represented with an integer value. There is an N:1 relationship between files and locators. Each file in HDFS is assigned to at most one locator and many files can be assigned the same locator. File with the same locator are placed on the same set of datanodes, whereas files with no locator are placed using Hadoop’s default strategy. Co-placement deals with all data blocks including the replicas.

To manage the locator information, and keep track of co-placed files, a new data structure called the \textbf{locator table} is introduced into the namenode of HDFS. The locator table stores a mapping of locators to the list of files that share this locator. Similar to the block-to-node mapping in HDFS, the locator table is not synchronized to disk. Instead, it is maintained dynamically in memory while the namenode is running and reconstructed from the file system image when the namenode is restarted. To facilitate reconstruction, the locator of each file is stored in its INode.

Data placement is modified in the following way: Whenever a new file $f$ with locator $l$ is created, the locator table is queried to check whether it contains an entry for $l$ or not. If not, a new entry $(l, f)$ is added to the locator table and the default placement policy of HDFS is used to chose the datanodes to store the replicas. Otherwise, the locator $l$ is known and the list of all files with the same locator value are obtained from the locator table. For each file, the BlockMap data structure of the namenode is queried to get the list of its blocks and their storage locations(set of datanodes). To store $r$ replicas of the new file, $r$ datanodes have to be chosen from this list. $r$ datanodes that store the highest number of blocks with locator $l$ and have enough

\textsuperscript{1} HDFS’ Write Affinity- which places the first copy of a newly written file on the node that created the file- can be used to influence data placement in a limited way, but doing so cumbersome and error prone as it requires interaction with Hadoop’s scheduler.
space for the new file are chosen. If less than \( r \) datanodes have sufficient space, more nodes are selected based on the HDFS’ default data placement policy. Hence, the modified data placement policy is *best effort*: the creation of new files always succeeds, and data are not moved around to enforce strict co-placement. This approach is chosen as it retains the HDFS’ fault tolerance properties and does not incur reorganization costs.

Co-placement has some effect on the data distribution over the cluster. Data distribution depends on several factors, including the number of files assigned to each locator, the order of file creation, file sizes, disk capacity of datanodes etc. If too many files are assigned the same locator so that the current set of of \( r \) datanodes runs out of space, system shall switch to a new set of datanodes to store subsequent files with this locator. When this happens, only subsets of the files with the same locator are actually co-placed. The extended proposal, hence, sacrifices co-placement for availability. For this reason, the creation order of files in the extended system may also effect the data distribution since the first file that is ingested determines the set of nodes to store subsequent files with the same locator.

Next, a comparison of the default Hadoop setup and the proposed extension is discussed theoretically.

### 4.1 Log Processing on Plain Hadoop

In log processing, a log of events—such as a clickstream, a log of phone call records, application logs, or a sequence of transactions—is continuously collected. To analyze this data, the log files are moved to a distributed file system such as HDFS and analytical queries are run on the so-obtained collection of log files. In addition, reference data—such as user and account information—are often brought in to enrich the log data during the analysis. Two common queries for log processing are described below

#### 4.1.1 Sessionization

The most common query in log processing is sessionization, in which log records are divided into user sessions. This is done by grouping the records in the logs by an account or user id, sorting each of the groups by time stamp, and finally dividing the sorted lists into sessions.
The standard MapReduce algorithm for this query is the repartition-based solution described below.

Basic MapReduce Solution: The mappers read the log records from multiple log files and output the extracted group key (the account id) along with the original record as a key-value pair. The map output is grouped by key in the shuffling phase and fed to the reducers, where the log records for each group are sorted (by time stamp) and divided into sessions.

4.1.2 Join

Another important query for log processing is an equi-join between log and reference data; e.g., joining transaction logs with corresponding account information. Although various MapReduce join algorithms have been proposed [2, 13], the most widely used evaluation plan for this query in MapReduce is a repartition join.

Basic MapReduce Solution: The mappers read both log records (from multiple log files) and reference records, tag each record to indicate whether it is a log or reference record, and output the extracted join key (account id) and the tagged record as a key-value pair. This output is grouped by the join key and fed to the reducers, where a cross product between the log records and reference records of each group is performed.

4.2 Improved Log Processing On Extended System

As described above, the basic MapReduce solutions to both sessionization and join queries are repartition-based. However, repartitioning the data is an expensive operation in MapReduce because it requires local sorting in each mapper, shuffling data across the network, and merging of sorted files in each reducer. However, it is well-known that data repartitioning can be avoided if the data are already organized into partitions that correspond to the query. Figure below shows an example with two log files of transactions(A & B) and a reference file of account information (R). Instead of directly copying the files to HDFS, a preprocessing step at load time is introduced in which both log files and the reference file are partitioned by account id. Notice that unlike Hadoop++ [6], which requires all the input files to be loaded and partitioned by a single MapReduce job, this approach can load files incrementally and in
different jobs by reusing the same partitioning function across the jobs. After partitioning the data, sessionization and join queries can be evaluated by a map-only job. For sessionization, each mapper reads the corresponding partitions from $A$ and $B$, groups all the records by account id, and divides each group into sessions. For join, each mapper takes as input the corresponding partitions from $A$, $B$ and $R$ (e.g., $A1$, $B1$ and $R1$) and performs a join of the $R$ partition with the union of partitions $A$ and $B$.

Pre-partitioning the data improves the performance by eliminating the expensive data shuffling operation, but each mapper still has to pay some network overhead. The default data placement policy of HDFS arbitrarily places partitions across the cluster so that mappers often have to read the corresponding partitions from remote nodes. This network overhead can be eliminated by co-placing the corresponding partitions, i.e., storing them on the same set of data-nodes.
4.2.1 Partitioning and Co-placing Data

Data partitioning is implemented using a single MapReduce job. Each log file is partitioned using the same partitioning function; this ensures that the generated partitions contain the same key ranges. To co-place corresponding partitions (same key range) from all files, a new locator for each key range is created and assigned to all corresponding partitions. Given the desired number of partitions, this is done by creating and passing a list of locators to Hadoop’s job configuration; the first locator in the list is assigned to the first key range and so on. A specially designed output format assigns the correct locators to each output partition when written to HDFS by a reducer. Since the same list of locators is used to partition each log file, corresponding partitions are assigned the same locator and are thus co-placed. When additional log files are ingested, the described job is run on the new data, using the same partitioning function and list of locators.

One issue to consider while partitioning the data is the size of each partition. On one hand, a large partition size produces a smaller number of partitions and hence smaller number of files. But recall that each mapper accesses all corresponding partitions. Mappers working on large partitions thus process a lot of data, which increases the cost of recovery in the case of failure. On the other hand, a small partition size would fit better with Hadoop’s basic flow and recovery model, but will create many files. This issue shall be dealt with when it comes to actual implementation. Another issue that may arise is skew in the partition sizes due to a skewed distribution of the partitioning key. Handling highly-skewed data requires more sophisticated algorithms and techniques, which are orthogonal to co-placement and beyond the scope of this work.

4.2.2 Sessionization Query

The sessionization query is implemented using a merge-based algorithm as a map-only job. The query takes $m$ log datasets $D_1, D_2, \ldots, D_m$ as input. Each dataset $D_i$ is partitioned into $s$ partitions $D_{i1}, D_{i2}, \ldots, D_{is}$. As corresponding partitions from different datasets contain the same key ranges, each single mapper $M_j$ needs to process one of the $s$ partitions from each of the $m$ input datasets ($D_{1j}, D_{2j}, \ldots, D_{mj}$) simultaneously. After assigning partitions to mappers,
record readers start consuming the input records. Here we exploit the fact that the partitions are sorted by the partitioning key: Record readers dynamically merge the sorted partitions into an sorted input stream. This ensures that the key-value pairs seen by mappers are already sorted by the partitioning key. The map function simply gathers all the log records for a given key, sorts by time stamp, and divides the log records into sessions.

4.2.3 Join Query

The join query is implemented using a hash-based map-side join algorithm [2]. For this query, there is an extra reference dataset \( R \) in addition to the log datasets \( D_1, D_2, \ldots, D_m \). The reference data \( R \) is partitioned in the same way as the log datasets; to obtain \( R_1, R_2, \ldots, R_s \). Each input split for the join algorithm consists of a reference partition plus the corresponding set of log partitions. In each input split, the records from all the log partitions need to be joined with the reference records. Each mapper builds an in-memory hash table using the keys from the reference dataset, and probes this hash-table using the keys from the log datasets to produce the join output.

4.3 Stochastic Model Explaining Improvement

This section provides a probabilistic model of block placement to compare the fault tolerance and data distribution characteristics of HDFS with and without the proposed extension. The notations used in the model are as follows:

- \( n \): total number of nodes
- \( k \): number of failed nodes
- \( m \): number of datasets to be placed on the same datanodes
- \( s \): number of partitions per dataset
- \( r \): replication factor of the splits
- \( D_i \): dataset \( i \), \( i \in \{1, \ldots, m\} \)
$D_{ij}$ partition $j$ of dataset $D_i$

$L_{lij}$ indicator variable for the event that node $l$ stores a replica of partition $D_{ij}$

$R_{ij}$ number of alive replicas of partition $D_{ij}$

$X$ total number of partitions lost, $X = \sum_{ij} I_{R_{ij}=0}$

$C_l$ I/O load of node $l$ without failures

$C_l^{\text{fail}}$ I/O load of node $l$ after 1 failure

Suppose that the cluster consists of $n$ nodes and that store $m$ datasets $D_i$ of $s$ partitions $D_{ij}$ each, i.e., $D_i = \{D_{i1}, ..., D_{is}\}$ for $1 \leq i \leq m$. Each partition is stored in a separate file. For each $j$, we co-place partitions $D_{1j}$ through $D_{mj}$ for efficient processing. Let $r$ be the desired number of replicas and let $L_{lij}$ be an indicator variable for the event that node $l$ stores a replica of partition $D_{ij}$. Initially, we have $\sum_l L_{lij} = r$ for all $i, j$. It is assumed throughout that whenever a partition $D_{ij}$ is read, one of its replicas is chosen at random. When multiple partitions are accessed in a single job, replicas are selected independently or from a single node. Finally, it is assumed that each node has sufficient disk space to store all the partitions assigned to it, that all partitions have equal size, and that partitions are small enough so that their files are not split across multiple nodes.

The following section analyzes the probability and amount of data loss when $k \geq r$ nodes of the cluster fail.

Default Hadoop: Set $L^*_{ij} = (L_{1ij}, ..., L_{nij})$; this vector indicates for each node whether it stores a replica of partition $D_{ij}$. By default, HDFS randomly distributes the replicas such that each set of $r$ distinct nodes is selected equally likely. Thus $L^*_{ij}$ has distribution:

$$
Pr[L^*_{ij} = l] = \begin{cases} \frac{n!}{r!} & \text{if } \sum_l L_l = r \\ 0 & \text{otherwise} \end{cases}
$$

for $l \in \{0, 1\}^n$. Partition $D_{ij}$ is lost when all of its $r$ replicas are location on the same set of failing nodes, or equivalently, when
\[ \sum_{i=1}^{k} L_{lij} = r \]

We obtain the probability of data loss

\[
Pr[X > 0] = 1 - \prod_{ij} (1 - Pr[\sum_{l=1}^{k} L_{lij} = r])
\]

\[
= 1 - \left[1 - \left(\frac{k}{r}\right) / \left(\frac{n}{r}\right)\right]^{m_s}
\]

where \( X \) denotes the number of lost partitions. By linearity of expectation, the expected number of partitions lost is given by:

\[
E[X] = \sum_{i,j} Pr[\sum_{l=1}^{k} L_{lij} = r] = m_s \left(\frac{k}{r}\right) / \left(\frac{n}{r}\right)
\]

Extended system: The analysis is similar as before, but the losses of partitions \( D_1, D_2, \ldots, D_m \) are now correlated: either all or none are lost. Thus a data loss occurs if and only if \( D_{1j} \) is lost for some \( j \).

\[
Pr[X' > 0] = 1 - \left[\left(\frac{k}{r}\right) / \left(\frac{n}{r}\right)\right]^{s}
\]

where the prime symbol is used to mark variables that involve co-placed data. The expected number of files lost remains unchanged i.e \( E[X'] = E[X] \)
Inference:

The table below gives an example for \( n = 40, s = 1000, \) and \( r = 3. \) Note that since \( s \) is kept constant, the total amount of data stored on the cluster increases with \( m. \) As mentioned above, the probability of losing data without co-placement is significantly smaller than losing data with out co-placement. The probability of loss of data in Hadoop increases as more data is stored on the cluster (since data is more spread out). The table also shows the the expected loss \( E[ X \mid X > 0] = E[X] / Pr[ X > 0] \) given that a data loss occurs. Since the expected loss is the same for Hadoop and the proposed technique, the expected amount of data loss given that data is lost is higher when the data is co-placed. Overall, co-placement does not have a negative effect on data loss: multiplying probability and conditional expected loss gives the same expected loss for both approaches.

| \( m \) | C-placement ? | Number of failed nodes (\( k \)) |
|---|---|---|---|---|
| | 2 | 3 | 4 | 5 |
| 2 | No | 0% | 18% (1.1) | 55% (1.5) | 86% (2.3) |
| | Yes | 0% | 10% (2.1) | 33% (2.4) | 63% (3.2) |
| 4 | No | 0% | 33% (1.2) | 80% (2.0) | 98% (4.1) |
| | Yes | 0% | 10% (4.2) | 33% (4.9) | 63% (6.3) |
| 8 | No | 0% | 55% (1.5) | 96% (3.4) | 100% (8.1) |
| | Yes | 0% | 10% (8.4) | 33% (9.7) | 63% (12.7) |

Table: Probability of data loss and expected number of partitions lost given that a loss occurs(in parenthesis)
**Data Distribution:**

To measure the impact of co-placement on data distribution over the cluster, a simple measure of file system load is derived. The measure is based on the assumption that all files are read with equal frequency, and it ignores CPU and network cost. This measure is chosen since it keeps the analysis tractable; more elaborate models can be built in a similar fashion. Suppose that a randomly chosen replica of each partition is read. Then, the expected number of partitions read from node $\ell$ is given by:

$$C_\ell = \sum_{i,j} L_{lij}/R_{ij}$$

where $R_{ij} = \sum L_{lij}$ denotes the number of active replicas of partition $D_{ij}$. In an ideal setting, all nodes would be guaranteed to have the same load. (With assumed measure of load, this happens only when $r = n$.)

Default Hadoop: Suppose that there are no failures, i.e., $R_{ij} = r$. From (1), $E[L_{lij}] = r/n$ and thus

$$E[C_\ell] = \sum_{i,j} E[L_{lij}]/r = ms/n$$

Observe that the number of replicas does not affect the expected load per node. To analyze variance, observe that

$$Var[L_{lij}] = E[L_{lij}^2] - E[L_{lij}]^2 = p(1 - p)$$

with $p = r/n$. Since different files are placed independently, $L_{lij}$ and $L_{l'i'j'}$ are independent for $i \neq i'$ and/or $j \neq j'$ and thus

$$Var[C_\ell] = \sum_{i,j} Var[L_{lij}/r] = \frac{ms}{r^2} p(1 - p)$$
As expected, \( \text{Var}[C_i] \) is monotonically decreasing with increasing replication factor; it reaches 0 when \( r = n \).

Now suppose that (without loss of generality) node 1 fails. Then \( R_{ij} = r - L_{1i} \), i.e. the replication factor of the files stores at node 1 decreases. Denoting by \( C_{\text{fail}} \), the load of the remaining node \( l \neq 1 \) after failure. Using the law of total expectation,

\[
E[C_{\text{fail}}^l] = \sum_{i,j} E[L_{ij}/(r - L_{1ij})] = \frac{ms}{n - 1}
\]

as expected, To derive the variance (using the law again), observe that

\[
\text{Var}[\frac{L_{ij}}{(r - L_{1ij})}] = \frac{n^2(r - 1) - n(r^2 - r - 1) - r}{(n - 1)^2n(r - 1)r}
\]

We obtain,

\[
\text{Var}[C_{\text{fail}}^l] = \frac{ms}{r(r - 1)}p'(1 - p') + \frac{n - r}{n(n - 1)(r - 1)}
\]

\[
\approx \frac{ms}{r(r - 1)}p'(1 - p')
\]

where \( p' = \frac{r - 1}{n - 1} \)

For large \( n \), the difference in variance between non-failure and failure cases is mainly determined by the replication factor \( r \); the variance decreases with increasing replication factor.
Extended System: With co-placement, all replicas of partitions $D_{ij}$, $1 \leq i \leq m$ are placed on the same set of nodes. In the notation used, this means that $L'_{1ij} = … = L'_{mij}$ and $R'_{1j} = … = R'_{mj}$.

By linearity of expectation, the expected load remains unaffected

$$E[C'_l] = \frac{ms}{n} \quad \text{and} \quad E[C'_l^{\text{fail}}] = \frac{ms}{n-1}$$

The terms of variance do change slightly: $m$ is replaced by $m^2$ and

$$Var[C'_l] = \sum_j Var[mL'_{1lj}/r] \approx \frac{m^2s}{r^2}p(1 - p)$$

is obtained.

and

$$Var[C'_l^{\text{fail}}] \approx \frac{m^2s}{r(r - 1)}p'(1 - p')$$

**Interpretation:** For both non-failure and failure case, co-placement does not affect the expected load $C_l$ on each node. However, the variance of the load increases by a factor of $m$ when co-placement is used. The coefficient of variance (COV) is used as a mean independent measure of variation.

$$CV[C_l] = \frac{\sqrt{Var[C_l]}}{E[C_l]}$$

A value significantly less than one indicated good balancing, whereas values larger than one indicate a lot of variation. As shown above, the COV increases by a factor of $\sqrt{m}$ when co-placement is used. With co-placement the COV decreases as the number of datasets increases; the increased number of files makes an uneven distribution less likely. With co-placement, the COV remains unaffected by the number of co-placed datasets since the location of the set is decide by the first copy of that set.
5. PLAN OF ACTION

The work carried out till now has been theoretical with very less focus given to implementation and its details. In the next stage of the project the implementation of the proposed approach shall be carried out as an extension to Hadoop 2.2.0.

Also, as mentioned in the introduction, integration of Storm shall also be carried out with the extended Hadoop in order to provide a framework to process high velocity log streams.
6. REFERENCES


http://hadoop.apache.org
   http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html


[20] Storm: Distributed and fault tolerant real-time computation
   https://storm.incubator.apache.org


