CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop

[1]Mohamed Y. Eltabakh*, Yuanyuan Tian*, Fatma Özcan*
Rainer Gemulla+, Aljoscha Krettek#, John McPherson*
*IBM Almaden Research Center, USA {myeltaba, ytian, fozcan, jmcphers}@us.ibm.com
+Max Planck Institut für Informatik, Germany rgemulla@mpi-inf.mpg.de
#IBM Germany aljoscha.krettek@de.ibm.com

ABSTRACT

Hadoop has become an attractive platform for large-scale data analytics. In this paper, we identify a major performance bottleneck of Hadoop: its lack of ability to colocate related data on the same set of nodes. To overcome this bottleneck, we introduce CoHadoop, a lightweight extension of Hadoop that allows applications to control where data are stored. In contrast to previous approaches, CoHadoop retains the flexibility of Hadoop in that it does not require users to convert their data to a certain format (e.g., a relational database or a specific file format). Instead, applications give hints to CoHadoop that some set of files are related and may be processed jointly; CoHadoop then tries to colocate these files for improved efficiency. Our approach is designed such that the strong fault tolerance properties of Hadoop are retained. Colocation can be used to improve the efficiency of many operations, including indexing, grouping, aggregation, columnar storage, joins, and sessionization. We conducted a detailed study of joins and sessionization in the context of log processing—a common use case for Hadoop—, and propose efficient map-only algorithms that exploit colocated data partitions. In our experiments, we observed that CoHadoop outperforms both plain Hadoop and previous work. In particular, our approach not only performs better than repartition-based algorithms, but also outperforms map-only algorithms that do exploit data partitioning but not colocation.

1. INTRODUCTION

Large-scale data intensive analytics has become indispensable to businesses as enterprises need to gain actionable insights from their increasing volumes of data. The emergence of Google's MapReduce paradigm [4] and its open-source implementation Hadoop [17] provide enterprises with a cost-effective solution for their analytics needs. Hadoop is a framework that supports data-intensive parallel applications, working with 1000s of compute nodes and petabytes of data.

In Hadoop, map and reduce functions operate on data stored in HDFS files [18]. HDFS stores large files as a series of blocks distributed over a cluster of data nodes and takes care of replication for fault tolerance. The data placement policy of HDFS tries to balance load by placing blocks randomly; it does not not take any data characteristics into account. In particular, HDFS does not provide any means to colocate related data on the same set of nodes. To address this shortcoming, we propose CoHadoop, an extension of Hadoop with a lightweight mechanism that allows applications to control where data are stored.

We studied the benefits of data colocation within the context of log processing, a common usage scenario for Hadoop. In this scenario, data are accumulated in batches from event logs, such as clickstreams, phone call records, application logs, or a sequences 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., computing user sessions. Our experiments suggest that colocation can significantly increase performance of these common data processing tasks.

As shown by [1, 16, 10, 6], join performance can be improved by copartitioning the two input files. In these cases, the performance gain is due to the elimination of data shuffling and the reduce phase of MapReduce. With colocated data partitions, joins can be executed using a map-only join algorithm with no remote I/O. Some of the earlier work [10] does not support colocation at all, while others [1, 6] perform heavy-weight changes to colocate 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] cogroups 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 colocate many files, not just two, and (2) to colocate newly ingested files incrementally with existing data.

As contrast to earlier approaches, we decouple the problem of colocating related files from the applications that exploit this property. For this purpose, we extend HDFS to enable colocating related files at the file system level. Our extensions require minimal changes to HDFS: We introduce a new file property to identify related data files and modify the data placement policy of HDFS to colocate *all* copies of those related files. 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 colocation by simply specifying related files. Use cases include colocating log files with reference files for joins, colocat-

[1]The author is currently an Assistant Professor in Computer Science Department, WPI (meltabakh@cs.wpi.edu).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 37th International Conference on Very Large Data Bases, August 29th - September 3rd 2011, Seattle, Washington.
Proceedings of the VLDB Endowment, Vol. 4, No. 9
Copyright 2011 VLDB Endowment 2150-8097/11/06... $ 10.00.

ing partitions for grouping and aggregation, colocating index files with their data files, colocating columns of a table—each stored in a separate file—, etc. CoHadoop's flexibility makes it attractive to high-level languages on MapReduce, such as Pig [15], Hive [8], and Jaql [9]. We expect these systems to automatically find the best way to exploit CoHadoop's capabilities to optimize query processing based on the observed workload.

We conducted extensive experiments to evaluate the performance of CoHadoop. In our experiments, we investigated the performance gain due to copartitioning and colocation separately, and showed that it is not enough to just copartition the data; we also need to colocate related partitions for maximum performance gain. We observed that CoHadoop provides significantly better performance than plain Hadoop solutions; no modifications to the scheduler are required and Hadoop's fault tolerance and data distribution characteristics are maintained. We also compared our approach with the "TrojanJoin" of Hadoop++ [6] and show that in addition to CoHadoop's flexibility and incremental features in colocating the data, it performs better than Hadoop++.

The contributions of this paper can be summarized as follows:

1. We propose a flexible, dynamic, and light-weight approach to colocating related data files, which is implemented directly in HDFS. Our solution is not tied to a particular use case, and can handle many different usage patterns, covering most earlier work.

2. We identify two use cases in log processing, i.e., join and sessionization, where copartitioning related files and colocating them speeds up query processing significantly. We also present efficient algorithms that exploit colocated data and align well with incremental and continuous data ingestion.

3. We study the fault tolerance, data distribution, and data loss properties of CoHadoop both empirically and analytically using a stochastic model.

4. We report detailed experimental results on the performance of CoHadoop for join and sessionization queries under different settings.

The remainder of this paper is organized as follows: In Section 2, we provide background information about HDFS. Next, we describe colocation in HDFS in Section 3. In Section 4, we present join and sessionization algorithms that exploit data colocation property. Section 5 discusses our experimental results. Finally, we review related work in Section 6 and conclude in Section 7.

2. HDFS BACKGROUND

HDFS [18] is a distributed file system that provides high throughput access to data. Files are split into *blocks* of fixed size and stored on *datanodes*. The block size is configurable and defaults to 64MB. Files can be written only once, i.e., updates of existing files are not allowed. The HDFS *namenode* keeps track of the directory structure of the file system. It also maintains a list of active datanodes as well as their data blocks in a dynamic data structure called `BlockMap`. Whenever a datanode starts up, it registers itself at the namenode with the list of blocks in its storage; these blocks are added to the namenode's `BlockMap`. Whenever the namenode detects failure of a datanode, the blocks of the failed node are removed from the `BlockMap`. Datanodes can both send blocks to clients upon request, but also store new blocks sent by the client. This process is coordinated by the namenode, which directs clients to the correct datanodes.

HDFS can be configured to replicate files for fast recovery in the case of failures. The default replication factor is three, which means that a block is stored on three separate datanodes. HDFS uses a simple data placement policy to select the datanodes that store the blocks and replicas of a file. The default policy of HDFS places the first copy of a newly created block on the local datanode at which the block is created (provided that there is enough space). This is called *write affinity*. HDFS then tries to select a datanode within the same rack for the second copy, and a datanode in a different rack for the third copy. As we will explain in Section 3, we have modified this data placement policy to support colocation.

Hadoop uses so-called `InputFormats` to define how files are split and consumed by the map tasks. Several `InputFormats` are provided with Hadoop. Input formats that operate on files are based on an abstract type called `FileInputFormat`. When starting a Hadoop job, the `FileInputFormat` is provided with a path containing the files to process. It then divides these files into one or more *splits*, which constitute the unit of work for a single map task in a MapReduce program. By default, the various `FileInputFormat` implementations break a file into 64 MB chunks (the default block size of HDFS). The Hadoop scheduler attempts its best to schedule map tasks on nodes that have a local copy of their splits. `InputFormats` provide an extensibility point that users can exploit to control the distribution of data to map tasks by assembling the customized splits. In Section 4, we show how `InputFormats` can help to implement efficient join and aggregation algorithms.

3. DATA COLOCATION

The objective of HDFS' current data placement policy [18] is to achieve load balancing by distributing the data evenly across the datanodes, independently of the intended use of the 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. CoHadoop, a lightweight extension to Hadoop, enables applications to easily define and exploit such customized strategies. It has been designed to be easy to use by applications, while at the same time retaining the good load balancing and fault tolerance properties of HDFS. In this section, we describe and analyze CoHadoop's approach to data colocation; algorithms and applications that exploit colocation are discussed in Section 4.

Research in parallel databases [20] as well as recent work on Hadoop [6, 1] have shown that careful data organization enables efficient algorithms for query processing. For example, a common technique that database systems employ is (1) to partition the data on some join or grouping attribute depending on the usage pattern, and (2) to colocate corresponding partitions on the same node. Transferring this idea to Hadoop is not straightforward: Although partitioning is easy to achieve in Hadoop [10], colocation is not [6, 1]. This is because Hadoop provides no principled way for applications to control where the data are stored.[1] Without colocation, data shuffling costs and network overhead reduces the effectiveness of partitioning. To overcome this and related problems, CoHadoop provides a generic mechanism that allows applications to control data placement at the file-system level, if needed. This mechanism can be used, for instance, to colocate corresponding partitions and their replicas on the same set of datanodes.

[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 is cumbersome and error prone as it requires interaction with (or even replacement of) Hadoop's scheduler.

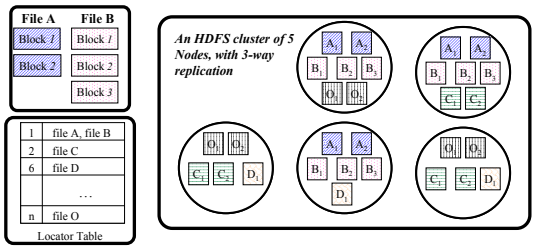


Figure 1: Example file colocation in CoHadoop.

To achieve colocation, CoHadoop extends HDFS with a new file-level property called a *locator*, and modifies Hadoop's data placement policy so that it makes use of this locator property. In our implementation, each locator is represented by an integer value, but other data types may also be used. 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 to the same locator. Files with the same locator are placed on the same set of datanodes, whereas files with no locator are placed via Hadoop's default strategy. Note that colocation involves all data blocks, including replicas. Figure 1 shows an example of colocating two files, `A` and `B`, via a common locator. All of `A`'s two HDFS blocks and `B`'s three blocks are stored on the same set of datanodes.

To manage the locator information and keep track of colocated files, we introduce a new data structure, the *locator table*, into the namenode of HDFS. The locator table stores a mapping of locators to the list of files that share this locator. Figure 1 shows the locator table corresponding to the example cluster. 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, we store the locator of each file in its `INode`, a disk-resident data structure that keeps track of all file properties.

Data placement is modified in the following way: Whenever a new file f with locator l is created, we query the locator table to check whether or not it contains an entry for l. If not, we add a new entry (l, f) to the locator table and use the default data placement policy of HDFS to choose the datanodes to store the replicas. Otherwise, locator l is known and we obtain the list of *all* files with locator l from the locator table. For each file, we query the `BlockMap` data structure of the namenode to obtain a list of its blocks and their storage locations (set of datanodes). To store r replicas of the new file, we need to choose r datanodes from this list. We select the r datanodes that store the highest number of blocks with locator l and have enough space for the new file. If less than r of these nodes have sufficient space, we select more nodes based on HDFS' default data placement policy. Thus data placement in CoHadoop is *best effort*: The creation of new files always succeeds (given that there is sufficient space in the cluster) and data are not moved around to enforce strict colocation. We chose this approach because it retains HDFS' fault tolerance properties (see below) and does not incur reorganization costs. However, CoHadoop guarantees perfect colocation when there is sufficient space on each node of the cluster and no failures occur.

As indicated above, colocation has some effect on the data distribution over the cluster. In fact, the data distribution depends on several factors, including the number of files assigned to each locator, the order of file creation, file sizes, and the disk capacity of the datanodes. For example, if too many files are assigned the same locator so that the current set of r datanodes runs out of space, CoHadoop will 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 colocated: CoHadoop sacrifices colocation for availability. Note that for this reason the creation order of files in CoHadoop may also affect the data distribution since the first file that is ingested determines the set of nodes to store subsequent files with the same locator. In general, an overuse of a single or few locators leads to skew in the data distribution. We recommend (and expect) that applications use locators sensibly so that a balanced data distribution can be achieved.

In Appendix 9.1, we analyze the data distribution and fault tolerance properties of CoHadoop and compare it with Hadoop via a simple probabilistic model. Regarding data loss due to failing datanodes, we show that the probability of losing data is significantly lower in CoHadoop than in Hadoop. Given that data loss occurs, however, CoHadoop loses more data. The probability and amount of data loss balance out: The expected amount of lost data is the same for both approaches. Regarding the data distribution over the cluster, our model indicates that CoHadoop only slightly increases the variation of the load when locators are used sensibly. Our experiments (Section 5.3) suggest that this increase does not negatively affect the query performance or fault tolerance.

4. EXPLOITING DATA COLOCATION

The proposed data colocation mechanism can be exploited by many different applications. In this section, we examplarily discuss how to use colocation to improve the efficiency of join and sessionization queries in the context of log processing.

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. Log data is usually collected at multiple application servers at different times and is large in volume. 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. We describe two common queries for log processing and demonstrate how they are usually evaluated in plain Hadoop.

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

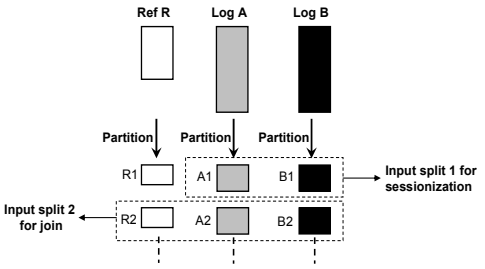


Figure 2: Copartition of reference data R, log datasets A and B

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 CoHadoop

As described in Section 4.1, 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 2 shows an example with two log files of transactions (A and B) and a reference file of account information (R). Instead of directly copying the files to HDFS, we can introduce a preprocessing step at load time in which we partition both log files and the reference file by account id; see Figure 2. Notice that unlike Hadoop++ [6], which requires all the input files to be loaded and partitioned by a single MapReduce job, CoHadoop 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 (e.g., A1 and B1), 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 colocating the corresponding partitions, i.e., storing them on the same set of datanodes. Figure 3 demonstrates how such colocation avoids network overhead in the context of the sessionization query. In the following sections, we discuss in more detail how to exploit CoHadoop's locators to partition and colocate data, and how to exploit colocation for sessionization and join queries.

4.2.1 Partitioning and Colocating Data

We implement data partitioning 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 colocate corresponding partitions (same key range) from all files, we create a new locator for each key range and assign this locator to all corresponding partitions. Given the desired number of partitions, this is done by creating and passing a list of locators

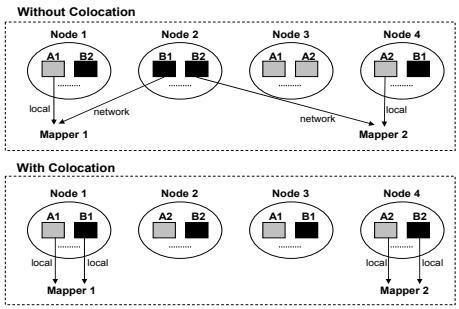


Figure 3: Sessionization without colocation vs. with colocation.

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 colocated. When additional log files are ingested, we simply run the just-described job on the new data, using the same partitioning function and list of locators. Note that our approach ensures that the records in each individual partition are sorted by the partitioning key (since this key is used to assign records to reducers).

One issue to consider while partitioning the data is the size of each partition. On the 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. We investigated the effect of partition size in our experiments (see Section 5.2) and found that small partition sizes work best with colocation, even though a larger number of files is produced.

Another issue that may arise is skew in the partition sizes due to a skewed distribution of the partitioning key. The algorithms and techniques proposed in this paper can handle mild skew (see Section 9.2.3). Handling highly-skewed data requires more sophisticated algorithms and techniques [5], which are orthogonal to colocation and beyond the scope of this paper.

4.2.2 Sessionization Query

We implement the sessionization query 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. We designed a special input format called `MergeInputFormat`, which merges the s partitions into a single input split as shown in Figure 2. 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 (see Section 4.2.1): 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

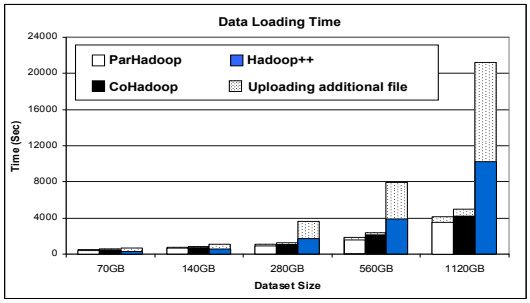


Figure 4: Data loading time.

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; we obtain $R_1, R_2, \ldots, R_s$. As shown in Figure 2, 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. To implement the hash-based algorithm, we make use of another input format called `SequentialInputFormat`. The `SequentialInputFormat` first reads all the records from the reference partition,[2] followed by the records from each of the log partitions in sequential order. 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.

5. EXPERIMENTS

We empirically evaluated the performance of CoHadoop by studying (1) the effect of various configuration parameters on data colocation—such as partition size and replication factor—, and (2) the impact of data colocation on the load time, data distribution, and fault tolerance. We compared CoHadoop with plain Hadoop (version 0.20.2) under two different data layouts: (a) Data are neither partitioned nor colocated (referred to as RawHadoop), and (b) Data are partitioned but not colocated (referred to as ParHadoop). We also compared CoHadoop with Hadoop++ [6], the state-of-the-art solution for pre-processing and partitioning the data in Hadoop for joins.

Cluster Setup. The experiments were evaluated on a 41-node IBM SystemX iDataPlex dx340. Each server consisted of two quad-core Intel Xeon E5540 64-bit 2.8GHz processors, 32GB RAM, 4 SATA disks, and interconnected using 1GB Ethernet. Each server ran Ubuntu Linux (kernel version 2.6.32-24), IBM Java 1.6, Hadoop 0.20.2. Hadoop's master processes (MapReduce jobtracker and HDFS namenode) were installed on one server and the remaining 40 servers were used as workers. Each worker was configured to run up to 6 map and 2 reduce tasks concurrently. The following configuration parameters were overridden in order to boost performance: sort buffer size was set to 512MB, JVM's were reused, speculative execution was turned off, and a maximum of 6GB JVM heap space was used per task. The replication factor was set to 3 unless stated otherwise. All experiments were repeated 3 or more times. We report the average of those measurements; the measurements were within $\pm 5\%$ of the mean.

Datasets. We generated synthetic transactional data that simulate the log-processing scenario discussed in Section 4.1. We generated two types of data: `Accounts` (reference data) and `Transactions` (log data). Each account has zero or more transactions. Each `Accounts` record is 50 bytes in size and contains an `AccountId` field (8 bytes) and other fields. Each `Transactions` record is 500 bytes in size and contains an `AccountId` (8 bytes) and a `Timestamp` (8 bytes) and some other data fields. The number of transactions for each `AccountId` follows a *log-normal distribution* with $\mu = \lg 10$ and $\sigma = 1$, which is a representative distribution for our client's data. The `Timestamp` values are uniformly selected from a specified time range. We assume that the transaction data are collected from multiple servers, which is typical in Hadoop and log processing applications. In the experiments, we consider *seven* transaction datasets within the scope of the queries. We keep the size of the `Accounts` dataset constant at 10GB while varying the size of the transaction datasets as reported in the figures. In CoHadoop and ParHadoop, datasets are hash-partitioned into files on the `AccountId` field. The number of files is controlled such that the resulting file size is within $\pm 5\%$ of the HDFS block size. Note that the way we generate the colocated partitions results in a quite balanced distribution. This is because (1) each locator is associated with the same number of partitions, (2) all partitions are roughly similar in size, and (3) all partitions for the same transaction dataset (one partition for each locator) are generated in parallel at the same time, and hence locators are well distributed over the cluster. In Appendix 9.2.3, we study the performance using a more skewed distribution.

Queries. We consider two types of queries as highlighted in Section 4.1: (1) A join queries between the `Accounts` and `Transactions` datasets based on the `AccountId` field, and (2) sessionization queries over the `Transactions` datasets where transactions belonging to the same account are grouped together and then divided into sessions based on a 30 minutes time window. In RawHadoop, these queries are implemented as a MapReduce job. In CoHadoop and ParHadoop, the datasets are already pre-partitioned so that we use map-only jobs.

5.1 Data Loading

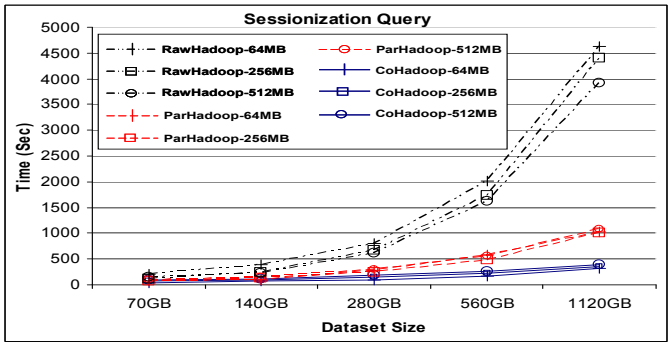
In our first experiment, we compared the time needed to pre-process and load a dataset by CoHadoop, ParHadoop, and Hadoop++. We also measured the time needed to pre-process and load an additional dataset to the dataset (incremental upload). To load a dataset, each of the three systems performed a full MapReduce job, in which (1) ParHadoop partitions the data, (2) CoHadoop partitions and colocates the data, and (3) Hadoop++ co-partitions the data and creates the Trojan indexes.[3] As shown in Figure 4, CoHadoop and ParHadoop have similar performance. CoHadoop is slightly slower due to increased network utilization: ParHadoop always writes the first copy of a partition locally, while CoHadoop may write all copies to remote nodes (if indicated so in the locator table). Note that the pre-processing and loading cost encountered by CoHadoop is very small compared to the savings at query time; e.g., the pre-processing cost is redeemed by a single sessionization query (Figure 5a). In contrast, Hadoop++ incurs a significant runtime overhead; it takes more than twice as much time as ParHadoop or CoHadoop.

To study the performance of incremental upload, we ingest an additional dataset to the already loaded datasets.[4] CoHadoop and ParHadoop need to only pre-process this new dataset (by performing a MapReduce job on that dataset), whereas Hadoop++ needs to rerun the co-partitioning and indexing process over the entire datasets, including the new dataset. Figure 4 shows that CoHadoop

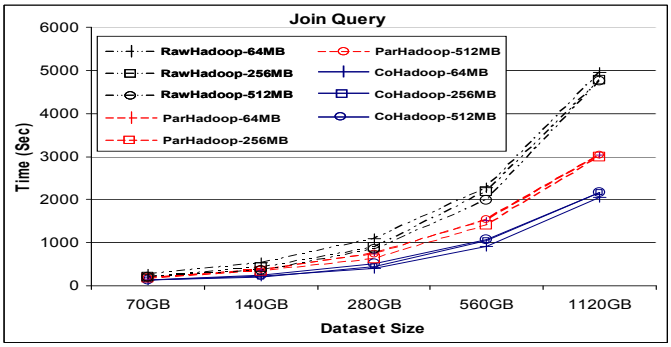
[2]Note that reference partitions are in general much smaller than the corresponding log partitions and fit in memory.

[3]The time taken to convert the data from its natural format to Hadoop++'s format is excluded from the figures.

[4]The size of the new dataset equals the size of each of the seven datasets, i.e., $\approx$15% of the `Transactions` data size.



(a) Sessionization query



(b) Join query

Figure 5: Performance of sessionization and join queries.

and ParHadoop can quickly ingest the additional dataset, whereas Hadoop++ incurs a cost higher than that of creating the original partitions.

5.2 Partition Sizes

We studied the effect of the partition size for different dataset sizes on the query response time. Figure 5 shows the results from the join and sessionization queries under HDFS block sizes of 64MB, 256MB, and 512MB.[5] The figure shows that data partitioning can significantly improve the performance, and that the best performance is achieved when the corresponding partitions are additionally colocated. For example, in the sessionization query (Figure 5a), ParHadoop saves up to 75% of RawHadoop's response time. This is because ParHadoop executes map-only jobs, and hence avoids the data shuffling/sorting phase as well as the reduce phase. For the same query, CoHadoop saves up to 60% of ParHadoop's response time, i.e., up to 93% of RawHadoop's response time. This is because reading the data locally is much faster than reading data over the network, especially when the data is large and the network is saturated.

In Figure 6, we keep the dataset size constant at 1120GB and zoom in to report the query response time under the three partition sizes. The figure shows that RawHadoop's preformance improves by 5% (10%) when the HDFS block size increases from 64MB to 256MB (512MB). In contrast, ParHadoop and CoHadoop slow down slightly; 5% (20%) for these block sizes. The reason for this difference in behavior is that RawHadoop launches seven times more mappers than ParHadoop and CoHadoop (recall that each map task in ParHadoop or CoHadoop processes seven corresponding partitions). Larger block sizes lead to fewer number of splits, which in turn reduces RawHadoop's overhead of starting and scheduling map tasks. With respect to ParHadoop and CoHadoop, the increased block size leads to a smaller number of partitions (since we set block size = partition size); each mapper thus processes much more data (up to 3.5GB), which has a negative impact on performance. We conclude that the smaller the partition sizes, the better the performance of CoHadoop.

The results of the join query (Figures 5b and 6b) exhibit a similar trend. For fairness, we made sure that the map-only tasks in ParHadoop are scheduled on datanodes containing at least one of the transaction files. Therefore, only the smaller account file and the remaining transaction files are shipped to the map tasks. The relative savings due to colocation are less than in the case of sessionization; e.g., ParHadoop and CoHadoop save up to 40% and 60%, respectively, of RawHadoop's response time. The main reason for the reduced saving is that the join query's output is around two orders of magnitude larger than that of the sessionization query. Thus a significant amount of time is spent in writing the output to HDFS (same for each approach).

[5]We use terms *block size* and *partition size* interchangeably since the partition size is kept within $\pm 5\%$ of the HDFS block size.

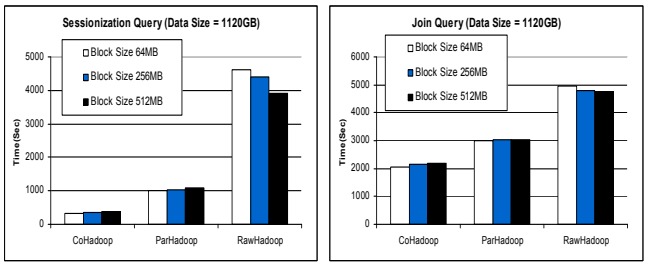
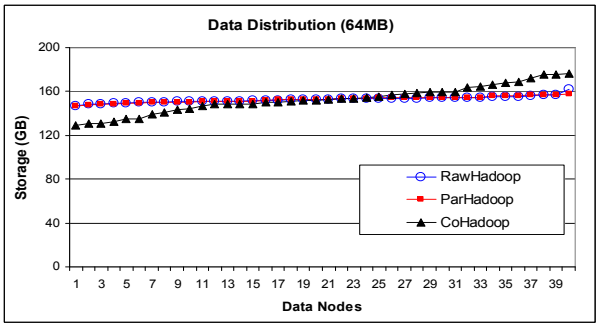


Figure 6: Varying block/partition size for 1120GB dataset.



(a) Data distribution over the cluster for block size 64MB.

	RawHadoop	ParHadoop	CoHadoop
Block Size = 64MB	1.7%	1.7%	8.2%
Block Size = 256MB	3.2%	3.1%	8.7%
Block Size = 512MB	4.8%	3.7%	12.9%

(b) Coefficient of variation percentage under different block sizes.

Figure 7: Data distribution over the cluster.

5.3 Data Distribution and Fault Tolerance

Next, we investigated the impact of colocation on data distribution and fault tolerance. While RawHadoop and ParHadoop try to evenly distribute the data on the cluster, CoHadoop additionally tries to colocate related files whenever possible. In Figure 7a, we show the data distribution over the cluster for block size of 64MB. The x-axis represents the cluster nodes sorted in an increasing order of used disk space, which in turn is shown in gigabytes (GB) on the y-axis. As can be seen, colocation only slightly disturbs the distribution of the data over the cluster nodes: The data are still

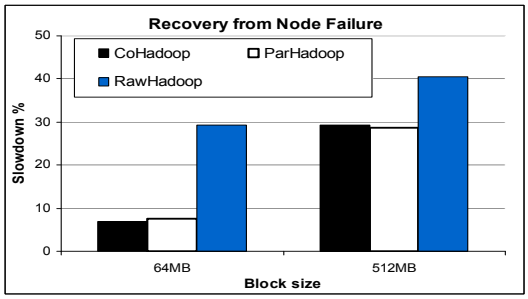


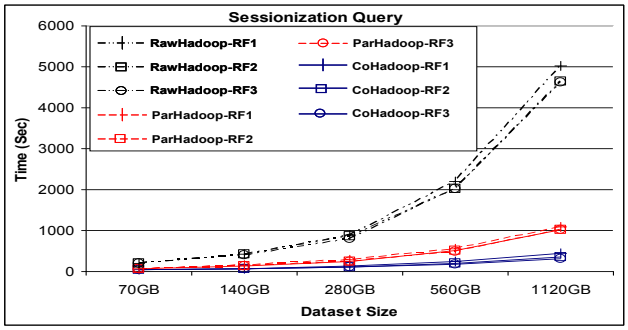
Figure 8: Fault Tolerance.

spread out and not concentrated at a few of nodes. Under larger block sizes, e.g., 256MB and 512MB, the three systems show similar trends, but the variation of the data distribution increases. This is shown in Figure 7b, where we report the coefficient of variation (the standard deviation divided by the mean) of the disk space for different block sizes. We observe that CoHadoop has 3–5 times higher variation than the other systems. The increase in variation is related to the number of colocated files (here 7); see Appendix 9.1.2 for analytical results and Appendix 9.2.2 for additional experiments. Note that the variation in all approaches increases as the block sizes increases: Larger block sizes and the resulting smaller number of data blocks make load balancing more coarse-grained.

We compared RawHadoop, ParHadoop, and CoHadoop to examine the effect of colocation on failure recovery. We performed a node failure experiment similar to the ones done in [6, 1]: We first set Hadoop's expiry interval to 60 seconds so that a node is considered down if we do not receive a heartbeart from it within a 60s window. Then, after 50% of the job's work is done, we randomly select a node to kill and measure how much this slows down processing. In Figure 8, we report the results for the sessionization query and HDFS block sizes of 64MB and 512MB. The dataset size is 1120GB. The slowdown percentage (y-axis) is given by $|t' - t|/t$ (as in [6]), where t is the query execution time without failure and t' is the execution time with failure. As a baseline, we also include the slowdown percentage of RawHadoop in Figure 8. Not surprisingly, RawHadoop has the highest slowdown percentage. This is because RawHadoop performs a full MapReduce job instead of a map-only job, and hence a node failure causes all map tasks that are executed on the failed node to be re-executed, the output from these mappers to be re-shuffled, and the reducers on the failed node to be (re-)executed as a second wave of reducers. Faster recovery due to the map-only jobs is an advantage of CoHadoop and ParHadoop, further strengthening the importance of co-partitioning. As the figure shows, the slowdown percentage is higher for a block size of 512MB because the amount of intermediate results lost due to the failure is higher. The figure also shows that CoHadoop and ParHadoop have almost the same slowdown percentage, which indicates that data colocation does not affect the fault tolerance in Hadoop.

5.4 Replication Factor

As mentioned previously, Hadoop creates multiple replicas of the data for both recovery purpose as well as for flexible scheduling of jobs. Hadoop's scheduler does a best-effort scheduling to assign map tasks to nodes that contain all (or portions of) the input data; these tasks are called *data-local* map tasks. As the replication factor increases, the percentage of data-local map tasks is expected to increase as well. In this section, we study the effect of the replication factor on query response time and investigate whether or not CoHadoop is more sensitive to the replication factor than the other systems. In Figure 9, we present results for the session-



(a) Performance under different replication factors.

	Average speedup %	
	Replication factor 2	Replication factor 3
CoHadoop	11.6%	19.73%
ParHadoop	5.8%	9.88%
RawHadoop	4.5%	6.88%

(b) Average speedup percent w.r.t Replication Factor 1.

Figure 9: Sessionization query under diff. replication factors.

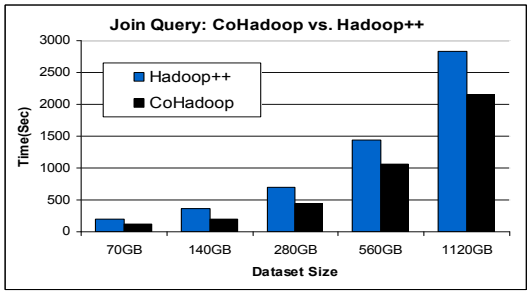


Figure 10: CoHadoop vs. Hadoop++.

ization query with an HDFS block size of 64MB and replication factors between 1 and 3. Figure 9a shows the absolute response times while Figure 9b shows average speed-up (over the five dataset sizes) w.r.t. replication factor 1. The speed-up percentage is defined as $(t_1 - t_r)/t_1$, where t_r is the time taken under replication factor r. As expected, an increased number of replica lead to faster processing times. As the replication factor increases from 1 to 2 to 3, the average fraction of data-local map tasks increases from 78% to 90% to 97%, respectively. Interestingly, CoHadoop has the highest speed-up and thus benefits most from replication. To see why, observe that in contrast to ParHadoop and RawHadoop, which read most of the input partitions remotely, a data-local map tasks in CoHadoop reads all its input partitions locally.

5.5 Comparison with Hadoop++

We compared CoHadoop with Hadoop++ [6], which is the closest to our work in that it also exploits data pre-partitioning and colocation. We focus only on the join query since it is the one considered in [6]. The HDFS block size is set to 256MB. We did minimal changes to Hadoop++ that include: (1) merging the seven transaction datasets into one because Hadoop++ colocates only two datasets (a reference and a transaction dataset), and (2) writing a converter program that converts our binary format to Hadoop++ binary format. The results in Figure 10 illustrate that CoHadoop outperforms Hadoop++ by around 20% to 55%. In addition to its better performance, CoHadoop is more suitable for applications that continuously ingest new data because of its incremental nature in colocating new files.

6. RELATED WORK

Recent benchmarks have identified a performance gap between Hadoop and parallel databases [14, 16, 10]. There has been considerable interest [1, 6, 10] in enriching Hadoop with techniques from parallel databases, while retaining Hadoop's flexibility. Our work in CoHadoop continues this line of research. Jiang et al. [10] conduct an intensive benchmark of various parts of Hadoop's processing pipeline. They found that (among others) indexing and map-side "partition joins" can greatly improve Hadoop's performance. In contrast to our work, they do not colocate partitioned data fragments. HadoopDB [1] and Hadoop++ [6] are closest to our work in spirit because they also try to colocate data. In contrast to our work, however, this is done by changing the *physical layout*: HadoopDB replaces HDFS by full-fledged relational databases, whereas Hadoop++ injects indexes and copartitioned data directly into raw data files. HadoopDB breaks the programming model and simplicity of MapReduce; it can be viewed as "another parallel database" [6]. Hadoop++ is less intrusive: colocated data (such as indexes and copartitions for joins) are stored as "Trojans" within HDFS files and splits; no changes to Hadoop itself are required. In contrast to CoHadoop, however, colocation in Hadoop++ is *static* and done at *load time*: any change of desired indexes, copartitioning, or even arrival of new data forces Hadoop++ to reorganize the entire dataset. Moreover, their colocation is geared toward joins and hence they can only colocate two files, whereas CoHadoop is pretty flexible in terms of the queries it can support, and number of files it can colocate.

Cheetah [3] and Hive [19] are two data warehousing solutions on Hadoop, and borrow many ideas from parallel databases. But, neither supports colocation 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 colocated as well. Their solution intermixes partitioning and colocation at the file system level, whereas CoHadoop decouples them so that different applications can use different methods to define related files. In this respect, CoHadoop can colocate related files defined by other means than partitioning, such as column files in a columnar storage format.

CoHadoop is heavily inspired by the more advanced partitioning features of parallel database systems [20], such as IBM DB2, TeraData, Aster Data nCluster, Vertica, Infobright, and Greenplum. In these systems, tables are copartitioned, and the query optimizer exploits this fact to generate efficient query plans [7]. CoHadoop adapts these ideas to the MapReduce infrastructure, while retaining Hadoop's dynamicity and flexibility. To achieve this, our approach differs from parallel databases in that we separate partitioning and colocation. Partitioning is controlled either directly by applications or by higher-level query languages such as Jaql [9], Pig [12], or Hive [19], which allow usage of non-relational data and a wide variety of partitioning rules. CoHadoop performs colocation at the file-system level and in a best-effort manner: When space constraints or failures prevent colocation, CoHadoop prioritizes high availability and fault tolerance.

The latest version of HDFS (0.21, released on August 23rd, 2010) provides a new API to override the default block placement policy, enabling applications to control placement of replicas by providing a custom Java class. Our data placement policy can be incorporated into HDFS using this API, which decreases the cost of code maintenance as new releases of HDFS becomes available. (The locator property associated with files is still needed, though.)

7. CONCLUSION

We presented CoHadoop, a lightweight solution for colocating related files in HDFS. Our approach to colocation is simple yet flexible; it can be exploited in different ways by different applications. We identified two use cases—join and sessionization—in the context of log processing and described map-only algorithms that exploit colocated partitions. We studied the performance of CoHadoop under different settings and compared it with both plain Hadoop solutions and map-only algorithms that work on partitioned data without colocation. Our experiments indicate that copartitioning and colocation *together* provide the best performance. Both theoretical analysis and experiments suggest that CoHadoop maintains the fault tolerance characteristics of Hadoop to a large extent.

Acknowledgement: We would like to thank Prof. Jens Dittrich and his team for providing us with the Hadoop++ source code.

8. REFERENCES

[1] A. Abouzeid and et al. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *VLDB*, 2009.

[2] S. Blanas and et al. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, pages 975–986, 2010.

[3] S. Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. In *VLDB'10*, 2010.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[5] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *VLDB'92*, 1992.

[6] J. Dittrich et al. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). In *VLDB*, 2010.

[7] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. *SIGMOD Rec.*, 19(2):102–111, 1990.

[8] *Hive*. `http://hadoop.apache.org/hive`.

[9] *Jaql*. `http://code.google.com/p/jaql`.

[10] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *PVLDB*, 3(1):472–483, 2010.

[11] H. Liu and D. Orban. Gridbatch: Cloud computing for large-scale data-intensive batch applications. In *CCGRID'08*, 2008.

[12] C. Olston et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[13] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX*, pages 267–273, 2008.

[14] A. Pavlo and et al. A comparison of approaches to large-scale data analysis. In *SIGMOD*, 2009.

[15] *Pig*. `http://hadoop.apache.org/pig`.

[16] M. Stonebraker and et al. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[17] The Apache Software Foundation. Hadoop. `http://hadoop.apache.org`.

[18] The Apache Software Foundation. HDFS architecture guide. `http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html`.

[19] A. Thusoo et al. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[20] P. Valduriez. Parallel database management. *Encyclopedia of Database Systems*, 2009.

Symbol	Description
n	Total number of nodes
k	Number of failed nodes
m	Number of datasets to colocate
s	Number of partitions per dataset
r	Replication factor
D_i	Dataset i, $i \in \{1, \ldots, m\}$
D_{ij}	Partition j of data set 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, $X = \sum_{i,j} I_{R_{ij}=0}$
C_l	I/O load of node l without failures
C_l^{fail}	I/O load of node l after 1 failure

Table 1: Summary of notation

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

Table 2: Probability of data loss and expected number of partitions lost given that a loss occurs (in parentheses)

9. APPENDIX

9.1 Analysis of CoHadoop

We derive a simple probabilistic model of block placement to compare the fault tolerance and data distribution characteristics of HDFS with and without colocation. The model notations are summarized in Table 1. Suppose that the cluster consists of n nodes and that we store m datasets D_i of s partitions D_{ij} each, i.e., $D_i = \{ D_{i1}, \ldots, D_{is} \}$ for $1 \leq i \leq m$. Each partition is stored in a separate file. For each j, we colocate partitions D_{1j} through D_{mj} for efficient processing, i.e., we model the setup of Section 4. 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. We assume 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 (no colocation) or from a single node (with colocation). Finally, we assume 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. These assumptions appear reasonable in practice: We expect users of colocation (such as Jaql) to ensure that partitions are balanced and reasonably small (e.g., ≤512MB).

9.1.1 Data Loss

We analyze the probability and amount of a data loss when $k \geq r$ nodes of the cluster fail.[6] Without loss of generality, we assume that the first k nodes fail.

Without colocation. Let's analyze the default block placement of Hadoop first. Set $L_{*ij} = (L_{1ij}, \ldots, L_{nij})$; 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

$$\Pr[\, L_{*ij} = \boldsymbol{l} \,] = \begin{cases} \binom{n}{r}^{-1} & \text{if } \sum_l \boldsymbol{l}_l = r \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

for $\boldsymbol{l} \in \{0,1\}^n$. Partition D_{ij} is lost when all of its r replicas are located on the set of failing nodes, or equivalently, when $\sum_{l=1}^{k} L_{lij} = r$. We obtain the probability of data loss

$$\begin{aligned} \Pr[\, X > 0 \,] &= 1 - \prod_{i,j} \Big(1 - \Pr[\sum_{l=1}^{k} L_{lij} = r\,]\Big) \\ &= 1 - \left[1 - \binom{k}{r} / \binom{n}{r}\right]^{ms}, \end{aligned}$$

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

$$\mathrm{E}[\, X \,] = \sum_{i,j} \Pr[\sum_{l=1}^{k} L_{lij} = r\,] = ms \binom{k}{r} / \binom{n}{r}.$$

With Colocation. The analysis is similar as before, but the losses of partitions $D_{1j}, D_{2j}, \ldots, D_{mj}$ 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:

$$\Pr[\, X' > 0 \,] = 1 - \left[1 - \binom{k}{r} / \binom{n}{r}\right]^{s},$$

where we use the prime symbol to mark variables that involve colocated data. The expected number of files lost remains unchanged, i.e., $\mathrm{E}[\, X' \,] = \mathrm{E}[\, X \,]$.

Interpretation. Table 2 gives an example for $n = 40$, $s = 1000$, $r = 3$. Note that since we keep s constant, the total amount of data stored on the cluster increases with m. As argued above, the probability of losing data with colocation is significantly smaller than losing data without colocation. In the case of CoHadoop, the probability of loss is not affected by the number of datasets (because they are all located at the same sets of nodes). In contrast, the probability of loss in Hadoop increases as more data is stored on the cluster (since data is more spread out). The table also shows the expected loss $\mathrm{E}[\, X \mid X > 0 \,] = \mathrm{E}[\, X \,] / \Pr[\, X > 0 \,]$ given that a data loss occurs. Since the expected loss is the same for Hadoop and CoHadoop (not shown in table), the expected amount of data loss *given that data is lost* is higher when data is colocated. Overall, colocation does not have a negative effect on data loss: multiplying probability and conditional expected loss gives the same expected loss for both approaches (up to rounding errors).

[6] If $k < r$, no data loss occurs.

9.1.2 Data Distribution

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

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

where $R_{ij} = \sum_l L_{lij}$ 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 our measure of load, this happens only when $r = n$.)

Without Colocation. Suppose that there are no failures, i.e., $R_{ij} = r$. From (1), we obtain $\mathrm{E}[L_{lij}] = r/n$ and thus

$$\mathrm{E}[C_l] = \sum_{i,j} \mathrm{E}[L_{l1j}]/r = \frac{ms}{n}.$$

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

$$\mathrm{Var}[L_{lij}] = \mathrm{E}\left[L_{lij}^2\right] - \mathrm{E}[L_{lij}]^2 = p(1-p),$$

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

$$\mathrm{Var}[C_l] = \sum_{i,j} \mathrm{Var}[L_{l1j}/r] = \frac{ms}{r^2}p(1-p).$$

As expected, $\mathrm{Var}[C_l]$ is monotonically decreasing with increasing replication factor; it reaches 0 when $r = n$.

Now suppose that (without loss of generality) node 1 fails. Then, $R_{ij} = r - L_{1ij}$, i.e., the replication factor of the files stored at node 1 decreases. Denote by C_l^{fail} the load of a remaining node $l \neq 1$ after failure. Using the law of total expectation, we have

$$\mathrm{E}[C_l^{\text{fail}}] = \sum_{i,j} \mathrm{E}[L_{lij}/(r - L_{1ij})] = \frac{ms}{n-1}.$$

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

$$\mathrm{Var}\left[\frac{L_{lij}}{r - L_{1ij}}\right] = \frac{n^2(r-1) - n(r^2 - r - 1) - r}{(n-1)^2 n(r-1)r}.$$

We obtain

$$\begin{aligned}\mathrm{Var}[C_l^{\text{fail}}] &= \frac{ms}{r(r-1)}p'\left(1 - p' + \frac{n-r}{n(n-1)(r-1)}\right) \\ &\approx \frac{ms}{r(r-1)}p'(1-p'),\end{aligned}$$

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.

With Colocation. With colocation, all replicas of partitions D_{ij}, $1 \leq i \leq m$, are placed on the same set of nodes. In our notation, this means that $L'_{l1j} = \ldots = L'_{lmj}$ and $R'_{1j} = \ldots = R'_{mj}$. By linearity of expectation, the expected load remains unaffected:

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

m	Colocation?	$\mathrm{CV}[C_l]$	$\mathrm{CV}[C_l^{\text{fail}}]$
1	-	11.1%	11.2%
2	No	7.8%	7.9%
	Yes	11.1%	11.2%
4	No	5.6%	5.6%
	Yes	11.1%	11.2%
8	No	3.9%	4.0%
	Yes	11.1%	11.2%

Table 3: Variation of load without (C_l) and with failures (C_l^{fail})

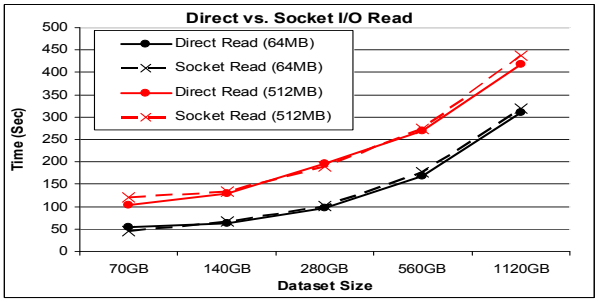


Figure 11: Direct I/O vs. socket I/O read.

The terms for variance do change slightly: m is replaced by m^2 and we obtain

$$\mathrm{Var}[C'_l] = \sum_j \mathrm{Var}\left[mL'_{l1j}/r\right] \approx \frac{m^2 s}{r^2}p(1-p)$$

and

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

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

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

A value significantly less than one indicates 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 colocation is used. In practice, the COV is low without colocation, and so this increase is often negligible. Table 3 shows an example for $n = 40$, $s = 1000$, and $r = 3$. Note that the number of datasets (and thus the total data size) doubles in each row. Without colocation, the COV decreases as the number of datasets increases; the increased number of files makes an uneven distribution less likely. With colocation, the COV remains unaffected by the number of colocated datasets since the location of each colocated set is decided by the first copy of that set.

9.2 Performance Discussion

9.2.1 Direct vs. Socket I/O Read

We studied the performance of CoHadoop under two different I/O modes: (1) Direct read, where local data are read through direct disk I/O, and (2) socket read, where local data are read through

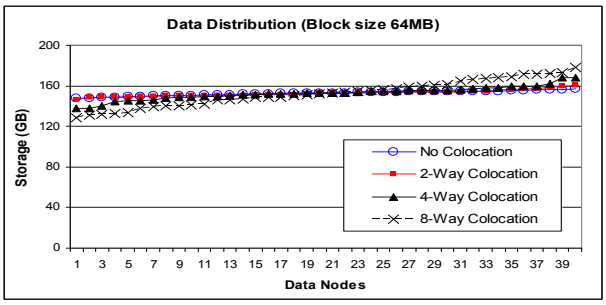


Figure 12: Data distribution over the cluster. Coefficients of variation for 0-, 2-, 4-, and 8-Colocated, are 1.6%, 2.0%, 4.3%, and 8.5%, respectively.

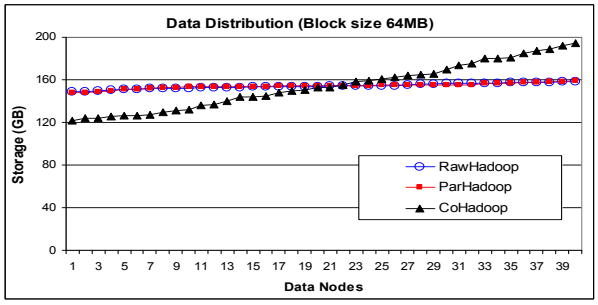


Figure 13: Data distribution over the cluster (mildly-skewed distribution of transactions). Coefficients of variation for RawHadoop, ParHadoop, and CoHadoop are 1.6%, 1.7%, and 14.0%, respectively.

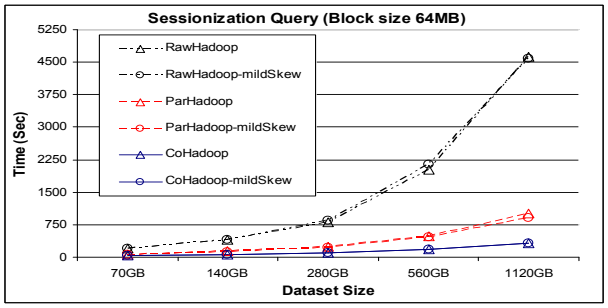


Figure 14: Sessionization query under different distribution parameters.

Hadoop's inter-process communication layer (socket layer) in the same way as the remote data are read. For that purpose, we applied an online JIRA patch available at `https://issues.apache.org/jira/browse/HDFS-347`.

Unlike the micro-benchmark experiment in [10] which runs on a one-node cluster and focuses only on streaming the data through a map function, we use the sessionization query as a more typical workload that involves both CPU and data output costs. In Figure 11, we report the results under 64MB and 512MB HDFS block sizes. The results illustrate that, for the sessionization query, direct read does not outperform socket read by much. As the figure shows, most of the data points have at most 1% to 5% speed-up when using direct disk I/O, where the speed-up is defined as $(t_{\text{socket}} - t_{\text{direct}}/t_{\text{socket}})$.

9.2.2 *Number of Colocated Files*

One of the key factors that affect the performance of CoHadoop and how the data are distributed over the cluster is the number of colocated files. In the experiment section (Section 5), we fixed the number of colocated files to seven (for the sessionization query) and eight (for the join query). In this section, we gradually increase the number of colocated files and measure the effect on the data distribution over the cluster (Figure 12). We use 8 transaction files of size 250GB each, and we assign locators as follows: In the *No Colocation* case, the partitions are not assigned locators, and hence they are stored according to the HDFS's default placement policy. In the *2-Way Colocation* case, we assign the same locator to the corresponding partitions from files i and $i+1$, for $i = 1, 3, 5, 7$. Similarly, in the *4-Way Colocation* case, we assign the same locator to the corresponding partitions from files i, $i+1$, $i+2$, and $i+3$, for $i = 1, 5$. And in the *8-Way Colocation* case, we assign the same locator to the corresponding partitions from all transaction files.

The key observation from the figure is that as we gradually increase the number of colocated files, the variation of the data distribution over the cluster also increases gradually. Additionally, the empirically observed variations are slightly higher than expected from our theoretical analysis (Section 9.1.2). This is because the cluster disks are not initially empty and the data partitions of a given transaction file do not have the same size, as we have assumed in Section 9.1.2.

9.2.3 *Distribution of Transactions*

Throughout the experiment section (Section 5), we used a log-normal distribution for the number of generated transactions per account in each log file. Our choice of parameters ($\mu = \lg 10$ and $\sigma = 1$) typically generates datasets with low skew. In this section, we test the resilience of CoHadoop to mild skew by increasing σ to 2, which in turn increases the variance of the distribution by a factor of roughly 387. We use an HDFS block size of 64MB and a replication factor of 3.

In Figure 13, we plot the data load over the cluster for the mildly-skewed distribution (As before, the x-axis represents the cluster nodes sorted in an increasing order of used storage). Comparing Figure 13 to Figure 7, we observe that CoHadoop encounters more variation in the data distribution over the cluster when more skew is added to the data, whereas RawHadoop and ParHadoop do not change much. This behavior is expected since the latter systems blindly divide the data into equal-sized blocks regardless of the data content, and independently distribute these blocks over the cluster. In contrast, CoHadoop treats all colocated partitions as one unit. Since the size of these partitions varies more in the case of skew, the variation of the used storage over the cluster increases as observed in Figure 13. Despite this increase in storage variation, the query performance remains almost unaffected, as discussed next.

In Figure 14, we compare the performance of the sessionization query under the two different distribution parameters (For the mildly-skewed distribution, the labels are suffixed with '*mildSkew*'). As the figure shows, the effect of the data distribution on query performance is negligible. The reason is that, in the sessionization query, each map task in CoHadoop and ParHadoop will process 7 files that vary in their sizes and it is very unlikely that all of the 7 files processed by a single map task will be large in size. Moreover, the entire sessionization query runs 10 waves of map tasks, therefore even if some map tasks will take more time to complete (because of the skew), the other map tasks will be running on the other empty slots which balances out the overall execution time.