Replication and Data Placement in Distributed Key-Value Stores

Manuel Gomes Ferreira
manuel.g.ferreira@ist.utl.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. MapReduce is a programming model widely used to process large amounts of data. Its its known for good scalability, ease of use, and fault-tolerance. However, it still exhibits some performance limitations when accessing the data to be processed, being far from reaching the performance of a well-configured parallel Database Management Systems (DBMSs). In this report we survey the most relevant optimisations that have been proposed for MapReduce, including indexing, data layouts, and co-location. We propose a novel system that, combining these three techniques, aims at boosting the execution time of MapReduce jobs by providing more efficient access to input data. Finally, we briefly describe how we plan to conduct the experimental evaluation of our system using a Twitter dataset.

1 Introduction

Today, there is an increasing need to analyse very large datasets, a task that requires specialised storage and processing infrastructures. The problems associated with the management of very large datasets have been coined "big data". Big data requires the use of massively parallel software, running on hundreds of servers, in order to produce useful results in reasonable time.

MapReduce is a programming model aimed at processing big data in a parallel and distributed manner. Known for its scalability, ease of use and fault-tolerance, MapReduce has been widely used by different domain applications such as search engines, social networks, or processing of scientific data. The MapReduce framework shields programmers from the problems of writing a distributed program, such as managing the fault-tolerance or the explicitly programming of the communication among processes. Users only have to define their programs mainly by means of two simple functions called *Map* and *Reduce*. The run-time automatically assigns Map and Reduce tasks to nodes, manages the communication among those nodes, monitors the execution of the tasks, and adapts to the size of the cluster, making the application easily scalable.

MapReduce is very flexible, and does not enforce any particular data model or format. However, being completely oblivious to the data format, MapReduce is generally forced to scan the entire input, even if only fractions of it are relevant

for a particular job. Due to these limitations, MapReduce is far from reaching the performance of well-configured parallel Database Management Systems (DBMSs) [20], which on the other hand, are much more difficult to tune and to configure and requires the data to be fit into a rigid and well-define schema.

In order to improve the performance of MapReduce, several research projects have experimented to augment the basic MapReduce with additional mechanisms that can optimize data access. An example of one such mechanism is the use of indexes, a technique that has proven to be useful in the context of DBMSs, to reduce the scope of a search. In this report we survey the most relevant optimisations that have been proposed for MapReduce and suggest ways to extend those efforts.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 we provide a brief introduction to MapReduce and in Section 4 we discuss the aspects that affect its performance. Section 5 surveys related work on improving the performance of Mapreduce. Section 6 describes the proposed architecture to be implemented and Section 7 describes how we plan to evaluate our results. Finally, Section 8 presents the schedule of future work and Section 9 concludes the report.

2 Goals

This work addresses the problem of optimizing the replication and data placement in distributed storage systems in order to improve the execution time of MapReduce jobs. More precisely, we aim at:

> *Goals:* Designing and implementing a novel system that boosts the execution time of MapReduce jobs by providing efficient data accesses.

As it will become clear further ahead, our system will combine different techniques that have been proposed in the literature, such as indexes, data layout and co-location, in order to capture the best of each one.

We plan to perform an experimental evaluation in order to measure the effectiveness of our system when performing MapReduce jobs.

The project will produce the following expected results.

> *Expected results:* The work will produce i) a specification of the data placement algorithms; ii) an implementation for Apache Hadoop, iii) an extensive experimental evaluation using a workload based on a realistic traces obtained from Twitter[2].

3 MapReduce Basics

MapReduce[6] is a framework for supporting the parallel processing of large amounts of unstructured data. MapReduce includes both a model to structure the computation and a runtime to parallelize the computation on a cluster of workstations in a fault-tolerant manner.

The computation model supported by MapReduce is quite generic. Data processing must be composed into two functions that are executed sequentially, namely: a *Map* and a *Reduce* function. Basically, the Map function is responsible for processing parts of the input and generating intermediate results which are then collected and combined on the Reduce function to generate the final output. This allows a broad range of computations to be modelled as MapReduce jobs, which is one of the reasons why it is a widely used framework, applied on diverse types of unstructured data such as documents, industry logs, and other non-relational data. In fact, MapReduce was originally proposed by Google but has been adopted by many different applications that have to manage large volumes of data, such as Facebook [5] or scientific processing of the outputs produced by particles accelerators.

A runtime supporting the MapReduce model in a cluster of workstations automatically assigns map and reduce jobs to hosts, monitors the execution of those tasks, and relaunches tasks if failures occur (for instance, if a host fails). This relieves the programmer from being concerned with the control plane during the execution of the application. The runtime can adapt the job scheduling to the number of servers existing in the cluster, making MapReduce jobs easily scalable.

3.1 Mappers and Reducers

A program running on MapReduce is called a *job*, and it is composed by a set of *tasks*. Some of these tasks apply the Map function to the input (the map tasks), while others apply the Reduce function to the intermediate results (the reduce tasks). So, a task can be viewed as an indivisible computation run in a node of the cluster, which operates over a portion of data. Nodes that compute tasks in the map phase are called *mappers* whereas the ones performing tasks in the reduce phase are named *reducers*.

When a job is submitted, the map tasks are created and each one is associated a portion of the input data. The Map function takes as input a key/value pair and, after some computation, it generates an intermediate key/value pair. The intermediate key/value pairs generated by these tasks are then assigned to reducers. This means that, during the execution, the intermediate results generated by mappers must be sent to the reducers (typically, via the network), such that a reducer will get all intermediate key/value pairs that share the same intermediate key.

The reduce tasks are then responsible for applying the Reduce function to the intermediate values of all keys assigned to it. This function will combine all the intermediate values in order to generate the final result of a job.

The nodes that execute map or reduce tasks are named *workers*, since they are the nodes that are responsible for doing the computational work. Each worker may serve as mapper, reducer, or as both if it performs map and reduce tasks. In addition to workers, there is one special node called *master* whose responsibility is to perform the task scheduling, i.e. the assignment of tasks to workers, and

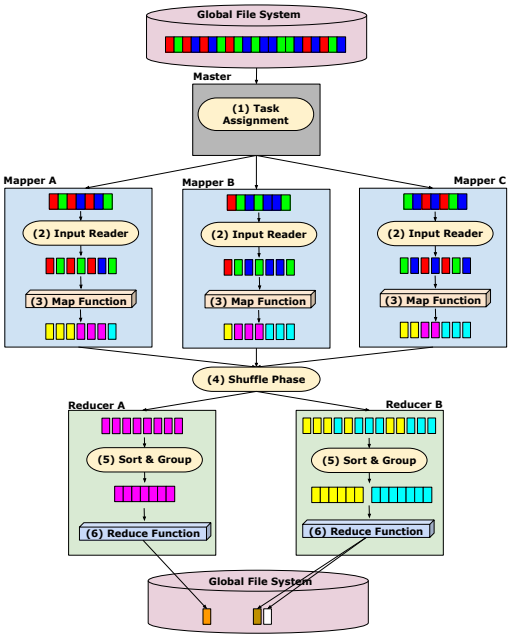

Fig. 1: MapReduce Steps

also to ensure the job completion (by monitoring the workers and re-assigning tasks to nodes if needed).

3.2 MapReduce Steps

Upon job submission, the input data is automatically partitioned by the MapReduce library into pieces, called splits. Each map task processes a single split, thus $\mathcal{M}$ splits entails $\mathcal{M}$ map tasks.

The map phase consists in having mappers read the corresponding splits and generate intermediate key/value pairs. Typically, the mapper uses a helper *input reader* to read the raw input data. The goal of the reader is to convert the input to a key/value pairs amenable to be processed by the Map function. This allows the MapReduce framework to support a variety of jobs that are not tied to any specific input format.

After the map phase, a partition function is applied over the intermediate key space to divide it into $\mathcal{R}$ partitions, each one to be processed in a reduce task (e.g., $hash(key)$ **mod** $\mathcal{R}$). Both $\mathcal{R}$ and the partitioning function can be configured by the user.

Figure 1 illustrates the sequence of steps executed during the execution of a a MapReduce job, which are enumerated below:

1. **Task assignment**: The input data is divided into splits and, for each split, the master creates a map task and assigns it to a worker.
2. **Input Reader**: Each map task executes a function to extract the key/value pairs from the raw data inside the splits.
3. **Map function**: Each key/value pair is fed into the user-defined Map function that can generate zero, one or more intermediate key/value pairs. Since these pairs represent temporary results, the intermediate data is stored in the local disks of the mappers.
4. **Combiner phase (optional)**: If the user-defined Reduce function is commutative and associative, and there is a significant repetition of intermediate keys produced by each mapper, an optional user-defined Combiner function can be applied to the outputs of the mapper[6]. The goal is to group, according to the intermediate key, all the intermediate values that were generated by the map tasks of each mapper. Typically, the same Reduce function code is used for the Combiner function. If this Combiner function is applied, each mapper outputs only one intermediate value per intermediate key.
5. **Shuffle phase**: The intermediate key/value pairs are assigned to reducers by means of a partition function in a manner such that all intermediate key/value pairs with the same intermediate key will be processed by the same reduce task and hence by the same reducer. Since these intermediate key/value pairs are spread arbitrarily across the cluster, the master passes to reducers the information about the mappers's location, so that each reducer may be able to remotely read its input.
6. **Sort & Group**: When the remote reads are finished, each reducer sorts the intermediate data in order to group all its input pairs by their intermediate keys.
7. **Reduce function**: Each reducer passes the assigned intermediate keys, and the corresponding set of intermediate values, to the user-defined Reduce function. The output of the reducers is stored in the global file system for durability.

During this entire sequence of steps, the master periodically pings the workers in order to provide programmer-transparent fault-tolerance. If no response is obtained within a given interval, the worker is considered as failed and the map or reduce tasks running on that node are restarted. If the master fails, the whole MapReduce computation is aborted[6].

3.3 Data Locality in MapReduce

MapReduce makes no assumptions on how the computation layer is related with the storage layer, as they may be completely independent. However, in practice, most MapReduce implementations rely in a distributed file system (Google File System (GFS) [12] for Google's MapReduce [6], or Hadoop Distributed File System (HDFS) [22] with Hadoop MapReduce [1]). Furthermore, often, the nodes responsible for the computational work may be the same nodes that store the input data.

In this common configuration, it is possible to save network bandwidth if the master takes into account which data each node is storing when scheduling the map tasks. First, it attempts to schedule a map task on a node that stores the data to be processed by that map task. If this is not possible, then it will attempt to schedule the task on the closest node (network-wise) to the one storing the data.

3.4 A MapReduce Implementation: Hadoop

Apache Hadoop [1] is the most popular open-source implementation of the MapReduce framework. It is written in Java and has received contributions from large companies such as Facebook and others[5]. Similarly to Google's MapReduce, Hadoop also employs two different layers: the HDFS, a distributed storage layer responsible for persistently storing the data among the cluster nodes; and the Hadoop MapReduce Framework, a processing layer responsible for running MapReduce jobs on these same nodes.

Storage Layer Hadoop DFS (HDFS) is a distributed file system designed to store immutable files, i.e., once the data is written into the HDFS is not modifiable but can be read multiple-times.

The HFDS implementation uses three different entities: one NameNode, one SecondaryNameNode and one or more DataNodes. A NameNode is responsible for storing the metadata of all files in the distributed file system. In order to recover the metadata files in case of a NameNode failure, the SecondaryNameNode keeps a copy of the latest checkpoint of the filesystem metadata.

Each file in HDFS is divided into several fixed-size blocks (typically configured with 64MB each), such that each block is stored on any of the DataNodes. Hadoop replicates each block (by default, it creates 3 replicas) and places them strategically in order to improve availability: two replicas on DataNodes on the same rack and the third one on a different rack, to prevent from loss of data, should an entire rack fail.

It is worth noting the difference between input splits and HDFS blocks. While an HDFS block is an indivisible part of a file that is stored in each node, an input split is logical data that is processed by a map task. There is no requirement for splits to be tied to blocks, and not even files. Consider the case where a file is split by lines, such that each map task processes one line (Hadoop supports the usage of arbitrary split functions). It may happen that a line overflows from an HDFS block, i.e., the split boundaries do not coincide with the HDFS block boundaries. In this case, the mapper processing that line must retrieve (possibly from some other node) the next HDFS block to obtain the final part of the line.

Processing Layer The entities involved in the processing layer are one master, named the JobTracker, and one or more workers, named the TaskTrackers. The role of the JobTracker is to coordinate all the jobs running on the system and

to assign tasks to run on the TaskTrackers which periodically report to the JobTracker the progress of their running tasks.

Hadoop's default scheduler makes each job use the whole cluster and takes the jobs' priorities into account when scheduling them. This means that higher priority jobs will run first. However, Hadoop also supports other schedulers, including shared cluster schedulers that allow running jobs from multiple users at the same time.

In terms of task scheduling, Hadoop does not build an *a priori* plan to establish which tasks are going to run on which nodes; instead Hadoop decides on which nodes to deploy each task in runtime[17]. As soon as a worker finishes a task it is assigned the next one. This means that should a worker finish all tasks associated with the data it stores locally, it may be fed tasks which entail obtaining data from other workers.

There are ten different places in the query-execution pipeline of Hadoop where User Defined Functions (UDFs) may be injected[7]. A user of the framework can define how the input data is split and how a split is parsed into key/value pairs, for example. This aspect makes Hadoop an easily customizable MapReduce framework.

3.5 Use Cases

We now provide three different examples of applications of MapReduce, each of which has different requirements in terms of data management.

Word Counting The word counting problem consists in counting the number of occurrences of the different words that appear in a text file. Several real world problems, like log analysis or text mining, can be viewed as instances of the word counting problem.

When MapReduce is used to solve this problem, one first starts by splitting the input text file and then assigning the resulting splits to the mappers. For each occurrence of *word*, the Map function emits the intermediate key/value pair in the form of $\langle word, 1 \rangle$. At the end of this phase, each mapper outputs as many intermediate key/value pairs as the number of occurrences of all the words it has processed, and those pairs are then transferred from the mappers to the reducers in the shuffle phase.

The assignment of keys to reducers is typically performed with the help of a hash function. Each reducer will get the pairs corresponding to all words assigned to it by the hash function. The Reduce function then groups all pairs by word, and outputs pairs in the format $\langle word, \mathrm{sum} \rangle$, where *sum* corresponds to the sum of the values associated with *word* in the intermediate results.

This use case is one which perfectly fits the usage of the Combiner function. In order to reduce the amount of data transferred in the shuffle phase, the mappers can do part of the reduce operation: each one sums the occurrences for each word that has identified. When using the Combiner step, the number

of intermediate key/value pairs output by each mapper would be equal to the number of different words it has processed.

Overall, the MapReduce model has a good fit to the word counting problem, since in this case both Map and Reduce functions do useful work in a balanced manner.

Selection The selection problem consists in returning only the set of records that meet some condition to produce the result. It is analogous to the SELECT operation in traditional DBMSs.

The first step to process such jobs, as normally happens in MapReduce, is to split the input dataset among the mappers that will extract the records from splits and pass them to the Map function. Typically, in this sort of problem, the Map function is responsible for doing the filtering, i.e. perform the test condition on each record and passing to the reducers only those that satisfy it. The Reduce function is then applied over the selected records.

In general, this type of problem is characterised by a longer and heavier mapping step, given that mappers need to analyse all the input, a task that is bounded by the I/O time required to read all the data from disk[20, 14].

Joining The problem of joining consists in creating a new dataset resultant from combining two datasets using values common to both datasets. It is analogous to the JOIN operation in DBMSs. HITS[16] and Page Rank[19] are examples of applications that are required to perform joins.

The most natural way to join two datasets according to some attributes using MapReduce is through a reduce-side join, where the actual join happens on the reduce phase of the framework. Using this approach, the work flow of a reduce-side join using MapReduce proceeds as follows: The two datasets are divided into splits and assigned to the mappers. After extracted, each record is passed to the Map function that is responsible for emitting its join attribute and the record as a key/value pair. The records are then shuffled among the reducers such that all the records from both datasets having the same join attribute go to the same reducer where are grouped together and fed to the Reduce function.

Like in the selection problem, joining requires a full scan of the entire datasets (in this case, to perform the partition, i.e. to divide them according to the join attribute). Moreover, the shuffle phase of the job entails the transfer of a large amount of data over the network, which represents a significant overhead and can introduce significant delays in execution time.

4 MapReduce Performance

MapReduce may not provide the desired performance in some use cases, such as the selection and joining scenarios presented in the previous section. This fact has motivated several recent works, which aim at improving MapReduce performance for different usage patterns. In this section we describe the most

relevant of these techniques, such as indexing, data layouts, and co-location, which aim at improving data accesses during query processing.

4.1 Indexing

In a plain MapReduce framework, the execution of a job requires that a full scan of the dataset is performed (i.e., by reading all the records one by one) even if only a small subset of them is going to be selected. Furthermore, all splits generate map tasks even if some of them do not contain relevant data and will not create any intermediate data. Also, in the map phase, all blocks have to be completely fetched from disk, which implies that all attributes of all records are brought to main memory without taking into account which ones the job is interested in. This problem can be avoided if the input data is indexed before the mapping step takes place.

Shortly, an index is built for one attribute and consists in a list of different values of that attribute contained in the indexed data, as well as the positions where these values can be found in the input file. Consequently, an index can significantly reduce the amount of data that must be read from disk, leading to a better performance of jobs, such as the ones implementing selection, where only a subset of the records is actually needed to produce the desired output.

Some systems create an unique index based in a single attribute[7], while others consider multiple attributes which leads to building multiple indexes[21, 8, 18].

The following indexing techniques can be applied:

- **Record-level indexing**: generally, this approach requires having the index inside each split, which is then used at query time to select the relevant records from the split, instead of reading the entire split record by record [7].
- **Split-level indexing**: it has been observed that the execution time of a query depends on the number of waves of map tasks that are executed, which in turn depends on the number of processed splits[10]. Furthermore, the time to process a split is dominated by the I/O time for reading its data plus the overhead of launching the map task to process it. So, in the end, the time taken to process a small number of records in a split can be roughly the same as the time required to process all records of that split.
  The idea of split-level indexes is to keep, for each attribute value, information about the splits that contain at least one record with that attribute value, such that splits with no relevant information are not processed. When processing a query, the master uses this index in order to get the minimal set of splits needed for that query, so map tasks are generated only for these splits.
- **Block-level indexing**: In cases where splits are comprised of multiple blocks, one can create block-level indexes. Since blocks contain multiple records, these indexes have the potential to bring meaningful time savings, according to the observations of Eltabakh et al. (2013). Block-level indexes map attribute values to blocks that contain at least one record with that

attribute value. Through these indexes, when processing a split, the relevant blocks are loaded, (possibly decompressed), and processed; whereas the blocks known not to match the selection criteria are skipped.

4.2 Data Layout

As mentioned in Section 3.4, the storage unit of HDFS consists in a fixed-size (typically 64MB) block of data. In fact, the way the data is organized *within* this block of data can greatly affect the response times of the system. We call this internal organization of blocks the *data layout*. We now introduce four different data layouts that may be applied on MapReduce:

– **Row-oriented**: As Figure 2 shows, when using this layout all fields of one record are stored sequentially in the file, and multiple records are placed contiguously in the disk blocks. This layout provides fast data loading since, in most cases, the data to be stored is supplied row-by-row, i.e. one entire record at a time. Row-based systems are designed to efficiently process queries where many columns of a record need to be processed at the same time, since an entire row can be retrieved with a single access to disk. On the other hand, it does not perform so well when supporting queries that only need to look at a small subset of columns. Furthermore, since each row typically contains different attribute from different domains, a row often has high *information entropy*, which makes this data layout not well suited for compression[3].

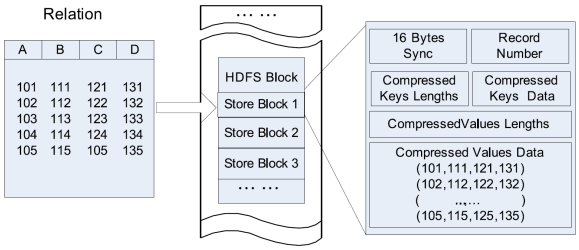

Fig. 2: Row-oriented data layout [13]

– **Column-oriented**: When using this layout, datasets are vertically partitioned by column, each column is stored separately and accessed only when the corresponding attribute is needed (see Figure 3). This layout is well suited for read operations that accessed a small number of columns, since columns that are not relevant for the intended output are not read. Furthermore, it can achieve a high compression ratio since the data domain is the same in each column, hence each column tends to exhibit low *information entropy*[3]. The better the data is compressed, the shorter the I/O time spent in reading it from disk. In this data format, each column is stored in (one or more) HDFS block, which means that it may be impossible to guarantee that all columns will be stored on the same node. As a result, record reconstruction may require network transfers to fetch the needed columns from different storage nodes.

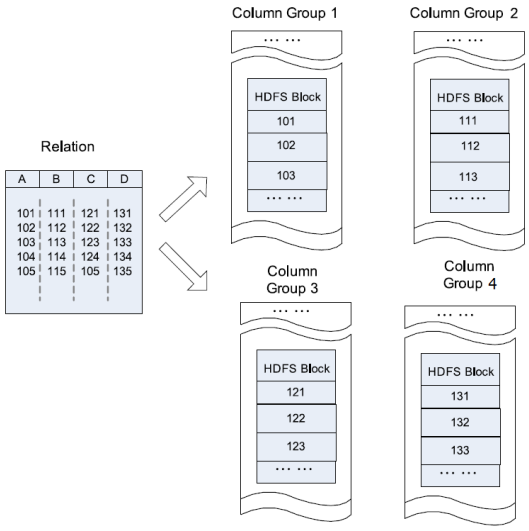

Fig. 3: Column-oriented data layout

– **Column groups**: This layout consists in organizing all columns of a dataset in different groups of columns. Different column groups may have overlapping columns (see Figure 4). The data inside each column group is not tied to any specific data layout: it may be stored in a row or column-oriented way. An advantage of column groups is that it can avoid the overhead of record reconstruction if a query requires an existing combination of columns. However, column overlapping in different column groups may lead to redundant space utilization. Although supporting a better compression than a row-oriented layout, the compression ratio achieved with column-groups is not as good as with a pure column-oriented layout since there is a mix of data types.

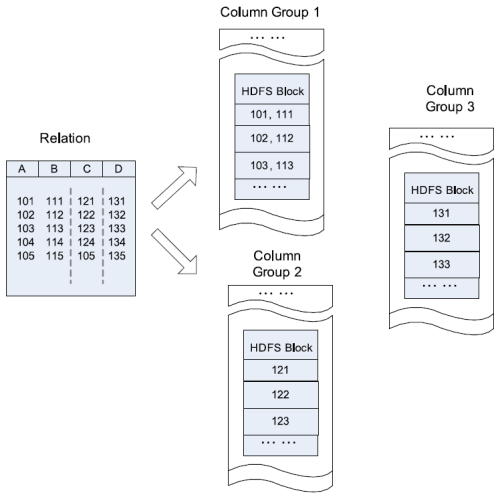

Fig. 4: Column groups data layout [13]

– **PAX**[4]: This hybrid layout combines the advantages of both row-oriented and column-oriented layouts as show in Figure 5. All fields of a record are on the same block as the row-oriented layout, providing a low cost of record reconstruction. However, within each disk page containing the block, PAX uses mini-pages to group all values belonging to each column. This enables to read only the mini-pages that contain the columns needed for the query at hand.

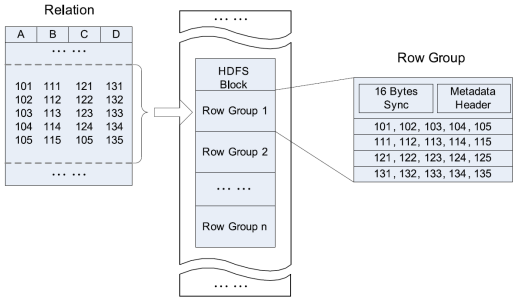

Fig. 5: PAX data layout [13]

4.3 Co-location

The data placement policy of Hadoop was designed for applications that access one single file[7]. Furthermore, HDFS distributes the input data among the storage nodes in an oblivious way, i.e., it does not look to the content of the file to decide how to store it.

However, we have seen that some jobs, such as the ones that perform joins, may need to re-partition the datasets in the map phase, and then migrate large quantities of data to perform the reduce step. This can be mitigated if data is stored such that related data is kept in the same node.

Co-location is a technique that attempts to ensure that related data is stored together on the same nodes. As a result, applications that require the combined processing of this related data can execute without needing to migrate data during the execution of the job. In particular, if data is co-located in an appropriate manner, namely if records with the same keys are stored on the same node, the mapping step may yield immediately the output of the join, avoiding the shuffle and the reduce phases.

5 Relevant Systems

In this section, we present some of the most relevant works that use the aforementioned techniques to improve MapReduce.

5.1 Hadoop++

Hadoop++[7] is a system designed to improve Hadoop performance. The goal of its authors is to achieve with a performance comparable to that of a

parallel database management system (DBMS). The authors propose two main techniques to improve the performance of Hadoop, named the Trojan Index and the Trojan Join. The goal of the former is to integrate record-level indexing capability into Hadoop, such that only the records that are relevant for a given job are accessed (this is most suitable for selection-based jobs), whereas the goal of the latter is to support data co-location, enabling joining-based jobs to be performed locally.

These capabilities are added to the Hadoop framework without modifying the base code. Hadoop++ adds new fields to the internal layout of splits and it injects appropriate UDFs in the Hadoop execution plan. Since data is stored in a row-fashion way, no data compression is used.

Trojan Index A Trojan Index, as shown in Figure 6, is a record-level index over a single attribute that is injected into each split. A Trojan Index is composed by an index data (SData T) plus additional information, such as an header (H) containing the indexed data size, the index size, the higher and the lower indexed attribute values, the number of records, and a footer (F) that identifies the split boundary.

To execute a given job, the footer (F) is used to retrieve all the indexed splits over which the map tasks are going to be created. When processing a split, the header (H) is used to compare the lower and the higher attribute values existing in that split with the low and the high selection keys of the job. If the two ranges overlap i.e. if the split contains relevant records for that job, the Trojan Index provides the offset of the first relevant record. The next records inside that split are then sequentially read until finding one with an attribute value higher than the high key of the job.

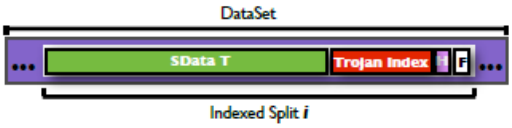

Fig. 6: Indexed Data Layout [7]

The creation of a Trojan Index is done at load time through a MapReduce job with proper UDFs: The map phase takes one record and emits a pair $\langle \text{splitID}+\mathcal{A}, \text{record} \rangle$ as the intermediate key-value pairs. The key is a concatenation of the attribute, $\mathcal{A}$, over which we are building the index, and the identifier of the split, splitID, whereas the value record contains all the attributes of that record. For each reducer to process roughly the same number of splits, the hash UDF re-partitions the records among them according to the splitID. In the reduce phase, so each reducer receives the records sorted by the attribute $\mathcal{A}$, another proper UDF is used. In order to recreate the original splits as loaded

into the system, a UDF aggregates the records based on the splitID. It is worth noting that now those splits are internally sorted. Finally, each reducer applies an index builder function over each split which builds the Trojan Index. So this new indexed data could be used in further queries, the reducers' output is stored on the distributed file system.

Trojan Join A Trojan Join consists in having the data co-partitioned and co-located in load time, i.e. the records of both relations with the same join attributes are grouped in the same split and hence placed on the same node allowing a join to be done in a map-phase only.

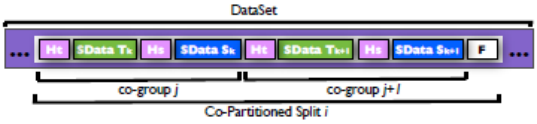

Fig. 7: Co-partitioned Data Layout [7]

The layout of Trojan Join, as shown in Figure 7, is composed by a footer (F) which identifies the split boundary, data from the two relations T S (depicted green and blue) and a header for each relation, (Ht) and (Hs), indicating the size of each partition inside the split.

As a pre-processing step, Trojan Join employs a MapReduce job in a simple manner to pre-partition and co-locate the data sets: for each data input split of each relation, the mappers apply a partitioning function on each record based on the join attribute. After this re-partitioning, all the records with the same join key (co-group) are naturally in the same reducer. The reducers' output are new data splits containing co-groups from the two relations that share the same join key and additional information that along with proper UDFs may perform further join queries locally. As the Trojan Index, these new data splits are stored on the distributed file system.

To process a join, UDFs that properly manage this new data layout are used. Similarly to regular Hadoop, a map task is created for each split. The difference is that the reduce phase is avoided since all the records of the two relations with the same join key are already on the same node and so, the cross product is processed in the map-side rather than at the reduce-side, hence eliminating the large amount of data transferring.

Discussion The Trojan Index is an index that allows to sequentially read the relevant records within each split. By storing data in rows, no record reconstruction is needed. With a record-level index, Trojan Index does not avoid passing through all the splits. Also, as a row-store, all the attributes have to be read from disk. On the other hand, Trojan Join avoids the reduce phase on join executions since the data is already pre-partitioned. However, these improvements

are based on the assumption that both the schema and the MapReduce jobs may be known in advance. Also, both solutions suffer from being static: the Trojan Index does not take into account the current workload to somehow change the attribute index, and with Trojan Join as new data files are ingested into the system, data needs to be re-organized to incorporate the new records.

5.2 LIAH

LIAH (Lazy Indexing and Adaptivity in Hadoop)[21] is a system that augments Hadoop with indexing capability to improve the performance of selection jobs. It makes use of a lazy indexing approach, to avoid incurring the overhead of creating the index at load time. Furthermore, LIAH can adapt dynamically to the workload, avoiding the need to know beforehand the access patterns to the data, as opposed to systems such as Hadoop++[7].

In LIAH, the indexes are created on a record-level and different attributes may be indexed leading to the existence of multiple indexes. LIAH takes advantage of the fact that data in MapReduce is replicated, to create different indexes for each replica of a block.

The index creation requires minimal changes in the MapReduce pipeline in order to build indexes as byproduct of job executions, as explained below:

First, in order to adapt to the workload, LIAH interprets the attribute accessed on a selection predicate as a hint for what may be an index that would speed up future jobs: For each block, if there is a suitable index for that attribute, the map task is created on the node where the data block with that index resides. Otherwise, a map task is scheduled on one of the nodes storing that block without an associated index. That map task will then fetch its block from disk to main memory, as normally, and feeds each record to the Map function. The most important feature of LIAH is that, after an unindexed block has been processed by the Map function, it is fed to the components that are responsible for creating the missed index on that block. In the end, the block becomes also indexed with no additional I/O operations. The NameNode is then informed about this newly indexed block so that future jobs may take advantage of it.

Second, an entire dataset is not indexed in a single step. Instead, LIAH creates the indexes in an incremental manner across multiple job executions, in order to avoid delaying the performance of the first jobs. For this, it uses an *offer rate*, ρ, meaning that it creates the index on one block out of ρ blocks in each MapReduce job.

In what concerns the data layout in LIAH, each row is first parsed according to the user-specified schema, followed by a conversion to PAX representation that avoids unnecessary column reads. Even though PAX allows for a good column-wise data compression[4], compression techniques were not addressed in this work.

Discussion In contrast to previous works [8, 7] that build static indexes, and require to know the profile of the MapReduce jobs in advance, LIAH can adapt

to changes in workloads. Also, by piggybacking indexing on the I/O operations of map tasks, the index creation of LIAH imposes little overhead. The PAX representation used by LIAH, allows to read only the required attributes. However, as the indexes are within blocks, blocks always need to be read, even if they contain no data required by the job. As opposed to previous works, LIAH requires changes on Hadoop, in particular, on the processing pipeline of Hadoop. Therefore, the code of LIAH needs to be kept up-to-date along with Hadoop.

5.3 E3

The E3 (Eagle-Eyed Elephant) framework[10] has been designed for boosting the efficiency of selection-based jobs in a Hadoop environment. This system uses split-level indexing techniques, whose goal is to access only the relevant data splits for the given queries.

In order for the E3 framework to eliminate as much as possible the need to process irrelevant splits, it builds indexes over all attributes in the dataset. These multiple indexes are then complemented with novel techniques involving domain segmentation, materialized views, and adaptive caching:

- **Range indexing for numerical attributes:** A range index consists in multiple ranges for each numerical attribute in each data split. These ranges are generated by a one-dimension domain segmentation technique that given a set of values $\{v_1,...,v_n\}$ and a maximum number of ranges, k, it returns at most k ranges such that they are "as tightly as possible" containing $\{v_1,...,v_n\}$. (see Figure 8). This index allows to eliminate splits containing no ranges hit by any equality or range predicates.

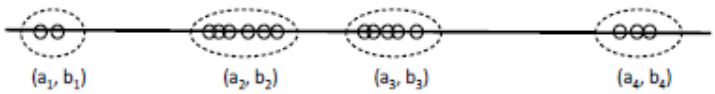

Fig. 8: Creating range index [10]

- **Inverted indexes for string attributes:** An inverted index is built for each string attribute. In this case, an inverted index consists in a mapping between a string atom (in this work, an attribute value is called an atom) and the splits containing an occurrence of that atom. This inverted index is physically represented by a fixed-length bitmap with so many bits as the number of splits in the dataset. The bitmap of the atom v has the ith bit set to 1 if the ith data split contains v. This index is appropriated for equality predicates over string attributes.
- **Materialized views for nasty atoms:** A nasty atom v is an atom that appears only a few times in each split but does it so in all of the $\mathcal{N}$ splits of a file. If inverted indexes were used, all $\mathcal{N}$ splits would be accessed. A materialized view is used to overcome this issue: All records containing v are stored in the materialized view. Then, in order to answer queries referring v, only the $\mathcal{M}$ splits composing the materialized view are accessed instead of the $\mathcal{N}$ splits of the original data, where $\mathcal{M} \ll \mathcal{N}$.

– **Adaptive caching for nasty atom pairs:** A nasty atom pair v,w consists in a pair of atoms that individually appear in most of the splits but as a pair, i.e. the atoms jointly, appear in very few records. To handle this, the query workload is monitored and these pairs are cached along with the splits that contain them. Adaptive caching differ from materialized view since it caches atoms that individually might be spread over almost all the data splits and hence not appear in the materialized view.

All these techniques are used at query time by an algorithm that runs inside an UDF InputFormat and that takes as input a conjunctive predicate in the form of $\mathcal{P} = p_1 \wedge p_2 \wedge p_3 \wedge ... \wedge p_n$, applies the appropriate techniques described above for each individual predicate, and returns a list of the minimal set of splits that must be processed to evaluate the predicate $\mathcal{P}$. Only after this, the JobTracker creates the map tasks on the splits selected by the algorithm.

In order to build the indexes and the materialized view, the E3 framework requires two passes over the data: First, a map-reduce job is performed to build the inverted indexes and to identify the subset of nasty atoms to be included in the materialized view according to a greedy algorithm. Second, the materialized views is built in a map-only job containing every record with at least one of the selected nasty atoms.

The design of E3 is not tied to any specific data layout: it uses Jaql for translating the storage format into JSON views of the data what allows it to operate over a vast storage formats. This JSON views consist on arrays of records that are collections of attributes.

Discussion E3 takes into account all attributes of a dataset to build effective indexes and to prevent the processing of irrelevant splits, thus avoiding unnecessary I/O and the cost of launching unnecessary map tasks. In addition, no record reconstruction is need since records are kept entirely which on the other hand implies that all the attributes have to be read from disk. Furthermore, it does not require any prior knowledge about the query workload. However, this is achieved at the expenses of a relatively costly pre-processing phase, with two passes over the entire dataset. In addition, storing the materialized view, the inverted index, and the range indexes, may lead to a significant storage overhead. Through a UDF that extracts the needed splits and map-reduce jobs to construct the various indexes and the materialized view, we may conclude that E3 was implemented without changing the core of the Hadoop.

5.4 Full-Text Indexing

Lin et al.[18] propose an index aimed at optimizing selection-based jobs on text fields to access only the blocks containing text that match a certain query pattern. This work shows how such index can interact with block compression in order to improve the I/O costs.

This index consists in a single full-text inverted index that works on block-level: given a piece of text the job is interested in, it returns the data blocks

containing records where that piece of text occurs. However, in order to obtain compact indexes, the text field in each record is first pre-processed to retain only one occurrence of each term. Consequently, this index can only be used to perform coarse-grained boolean queries.

It is worth noting how the data layout is addressed before we get into how the index works, since the latter depends on the former. The records are serialized by either Protocol Buffers or Thrift serialization frameworks. Data blocks composed by the encoded records in a row-oriented way are compressed using LZO compression. Files in LZO compression format are divided into smaller blocks (typically 256KB) that are automatically handled by an appropriate UDF (InputFormat, more exactly) to be normally processed by map tasks of Hadoop. Consequently, a split will comprise multiple LZO-compressed blocks.

The inverted index is used, when processing a split, to retrieve the blocks matching the selection criteria. At the end, only the relevant blocks within it are read from disk, decompressed and scanned. To complete the selection, in each relevant block, the encoded records are extracted, decoded and the same selection criteria is applied on each one.

The index creation procedure must be explicitly set up by an administrator who defines which serialization framework to use, how the encoded fields are extracted, among other directives.

Discussion This work addresses an inverted full-index that allows to avoid unnecessary disk I/O readings for blocks containing no interested data for a query at hand. Specifically, the overhead of extract, decode and scan the records inside an irrelevant block is intrinsically avoided as well. Unfortunately, the index creation is not a transparent process as data is ingested. Instead, it must be explicitly set up by an administrator.

In the Hadoop framework, each mapper unions all the blocks that comprises its split and extracts the records from it. However, in this implementation, it was necessary to modify the processing pipeline of Hadoop in order to a mapper skips ahead the irrelevant blocks when processing a split.

5.5 Trojan Data Layouts

The work of Jindal et al. (2011) proposes the usage of a new data layout for improving selection-based jobs. This layout consists in organizing the data inside each HDFS block in column groups, allowing to access only the relevant columns to answer a query at hand. This data layout, named Trojan Layout, is motivated by the observation that MapReduce wastes a significant amount of time reading unnecessary column data, due to the records being stored in a row-oriented way.

Externally, Trojan Layout follows the PAX philosophy in the sense that it keeps the same data in each HDFS block as the row-oriented layout. However, it exploits the fact that each block in HDFS is typically replicated three times for fault-tolerance proposes: each replica is internally organized in different column

groups, each one aimed to improve the performance of a different type of queries. As we will see later, this system was designed be aware of the schema and queries access patterns. Despite data being stored in a column-fashion way, enabling to achieve a good compression ratio, compression techniques were not addressed by the authors.

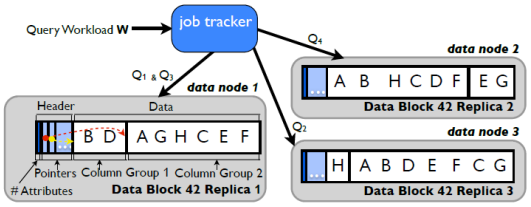

Fig. 9: Per-replica Trojan Layout [15]

As Figure 9 depicts, the Trojan Layout can be divided in two main components: an header and a set of column groups. The header contains the number of attributes stored in a data block and a several attribute pointers, each one pointing to the column group that contains that attribute. Each column group is internally organized in a row-oriented way.

At query time, the attributes needed to be read are identified. Then, the best Trojan Layout is picked according to those attributes. Finally, the map tasks are scheduled to the nodes containing the data block replicas containing the selected Trojan Layout. At the end, through a properly designed UDF, only the relevant column group is brought to main memory.

	A	B	C	D	E	F	G	H
Q1	1	0	1	0	1	1	1	1
Q2	1	1	1	1	1	1	1	0
Q3	1	0	1	0	1	1	1	1
Q3	0	0	0	0	1	0	1	0

Table 1: Example of access pattern. A given cell has a value 1 if a query accesses that attribute, otherwise it has value 0

The creation of this layout requires some pre-processing in order to identify the best query groups and reformat the data blocks to fit those query groups. Users have first to provide the query workload, the schema of their datasets, and the replication factor to a design wizard called Trojan Layout Wizard (TLW). Then, TLW groups the queries based on a query grouping algorithm that takes into account their access pattern. Finally, the column groups are created in order to speed up the generated query groups. For instance, in Table 1, attributes A,

C, E, F, G and H, are co-accessed in queries *Q1* and *Q3*. Thus, these attributes are grouped to speed up queries *Q1* and *Q3* as Figure 9 shows.

Discussion In this work, data is organized according to the incoming workload. Therefore, for a given query, the most suitable column group is chosen, avoiding to read unnecessary column data. This is achieved without changing the Hadoop processing pipeline but requires a Trojan Layout aware HDFS. On the other hand, record reconstruction of conditional queries may not exhibit good performance: if a query needs to access a column group only when a given record meets some condition, that column group has to be brought from disk and reading its data value by value until reaching the correspondent values of the record that satisfied the condition. This approach does not provide any mechanism to avoid passing through irrelevant HDFS blocks which would speed up the execution time of selective jobs. The major disadvantage of this system is that it assumes that the query workload can be known in advance and remains immutable along time leading to static column groups, an assumption that may prove too strong for some use cases.

5.6 RCFile

The RCFile (Record Columnar File)[13] is a system that introduces a novel data placement structure implemented on top of HDFS designed to improve selection-based jobs. The data layout introduced in RCFile follows the logic of the PAX format to improve the performance of access to compressed data by allowing to access only the necessary columns for a given query.

The data layout in RCFile follows the PAX idea of "first horizontally-partition, then vertically-partition", combining the advantages of both row-store and column-store: As a row-store, it guarantees that data of the same row is located in the same node, requiring a low cost of record reconstruction; As a column-store, it allows to apply column-wise data compression and avoid unnecessary columns reads.

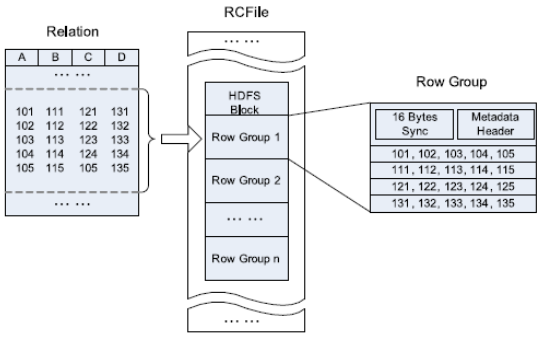

Fig. 10: Data Layout of RCFile [13]

As depicted in Figure 10, a HDFS block is first partitioned into row groups. Each row group contains a sync marker, a metadata header and table data organized in columns. The sync marker is used to separate continuous row groups whereas the metadata header stores information about that row group such as how many records are in that row group and how many bytes are in each column. The data layout of the table data in each row group is stored in a column-oriented way: All the values of the same column are stored contiguously. Such columns are independently compressed, achieving a good compression ratio.

Data reads for answering a query at hand proceeds as follows: a map task that is assigned a given HDFS block sequentially reads each row group inside it. To process a row group, through its header, only the columns that are going to be useful are loaded and decompressed.

To convert the raw data into this data format when loading a dataset, the only operation RCFile has to perform is re-organize the data inside each row-group.

Discussion RCFile proposes a new data placement structure that exploits column-wise data compression and skips unnecessary column reads and also avoids the network costs of record reconstruction as a row-store. RCFile is implemented through UDFs requiring no changes on the processing pipeline of Hadoop. Although a good record reconstruction cost is achieved by avoiding to do it through the network, in RCFile, as Trojan Data Layouts, whenever a record reconstruction is needed, it has to fetch the required columns and going through all their data until the values of the record that triggered the reconstruction have been read. Also, even the HDFS blocks containing no useful data have to be scanned. Its largest disadvantage is that since extra metadata such as the header and sync marker need to be rewritten for each row-group, it leads to additional space overhead.

5.7 CIF Storage Format

CIF[11] is a column-oriented storage format for HDFS that supports co-location through a customized placement policy, data compression, and optimized readings. By gathering this multiple components, CIF provides efficient access to the relevant column data, improving the Hadoop responsiveness when executing selection-based jobs.

The idea of CIF is that a dataset is first horizontally partitioned into splits, as shown in Figure 11, and each data split is stored in a separate sub-directory. Each column of a given split is individually stored as a single file within the sub-directory associated to that split. On each sub-directory there is also an additional file describing this schema. A new HDFS block placement policy were implemented to guarantee that column files associated to the same split are co-located on the same node to avoid a high record reconstruction cost (see Figure 12).

To process a split when a job is submitted, only the column files associated to that split that are needed to produce the result are scanned and the records

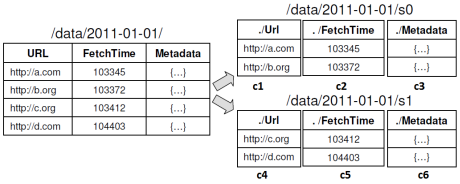

Fig. 11: Data Layout of CIF [11]

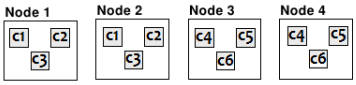

Fig. 12: Co-location with CIF [11]

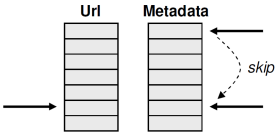

Fig. 13: Skip list [11]

are reconstructed only with those attributes. When scanning a column file, it uses a lazy record reconstruction technique that applies both a skip list and a lazy decompression scheme. The skip list resides within each column file and enables to skip a set of column values that are known not be relevant. Consider, for example, a query that returns the *Metadata* values of the records whose *Url* satisfy some given pattern. All the *Url* values are read one by one and, whenever a given *Url* value matches the pattern, the skip list enables to jump to the *Metadata* value that corresponds to the same record and that belongs to the file corresponding to *Metadata* column. As a column file may occupy more than one HDFS block, the lazy decompression scheme decompresses a HDFS block only when it is actually going to be read.

When loading data, the application has to pay the price of organizing it in the column-oriented fashion as previously described.

Discussion CIF introduces a new column-oriented storage format for Hadoop that reads only the columns whose attributes are referred in a job configuration. Moreover, it combines a lazy record construction that allows to eliminate disk I/O and mechanisms to co-locate column data that enables reconstructing records without network transfers. Even tough not being as good as a row-store, CIF employs a skip list to efficiently skip irrelevant column values when reconstruction records which implies accessing to different files. In addition, when reading a column, there is no way of avoiding to pass through the irrelevant HDFS blocks that comprise that column. These techniques have been added without modifications to the core of the Hadoop.

5.8 CoHadoop

CoHadoop[9] is an extension of Hadoop that employs a co-location mechanism allowing applications to control where their data is stored. This enables to

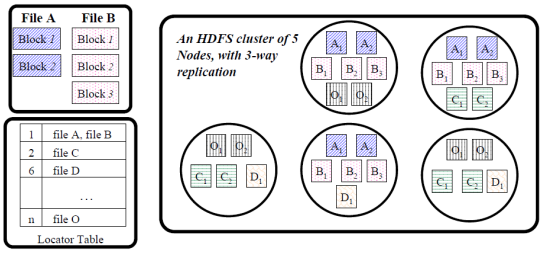

Fig. 14: Example of co-location with CoHadoop [9]

co-locate related data on the same set of nodes improving the performance of joining-based jobs since remote I/O accesses are avoided.

CoHadoop operates over raw-data and achieves co-location by extension HDFS with a new file-level property called *locator* that are set with a value in the files that are intended to be co-located. Data placement is modified so the files sharing the same locator value are placed on the same set of nodes. Files with no locator are placed using Hadoop's default strategy. In the NameNode of HDFS there is a *locator table* that does the mapping of locators to the list of files that share this locator (see Figure 14). Upon arrival of a new file with a locator value, this table is used to place the new file on nodes that store files related to the new one.

In order to running a map-only join, a pre-processing MapReduce job to partition the data is required: The map phase is responsible for partition each dataset according to the join attribute. In the reduce phase, one locator value is assigned to each join attribute range. This ensures that all the records containing the same join attribute are assigned to the same locator value when written to HDFS. When additional data is ingested, this mechanism is run using the same join attribute along with the locator table.

A join query can then be executed locally and is performed using a hash-based map-side join algorithm: Each input split consists of all partitions within the same key range of each dataset. CoHadoop uses an UDF called `SequentialInputFormat` in order to first read all the records from one partition followed by all the records of the other partition. The reading of the first partition will built an hash table using the joining attribute values of the records within that partition. Then, it probes this hash table using the joining attribute values from the other dataset to produce the join output.

Discussion CoHadoop decouples partitioning and co-location so that different applications can use different methods to define related files. This enables CoHadoop to co-locate related files defined by other means than partitioning, such as column files in a columnar storage format. For this, CoHadoop requires applications to give hints about the relation between data, which in the case of the joining, was the partitioning function the responsible for creating the relation

between records. As previously stated, CoHadoop requires to change the HDFS and the data placement policy.

5.9 Comparative Analysis

Most of the related work is aimed at avoiding the need to perform a full scan of the input data in the cases where only a fraction of the input is relevant for the desired output. The main approach that has been pursued to achieve this goal consists in building indexes that allow to identify where the needed input is located.

Indexing does not change the data layout. Therefore, even if it allows to retrieve only the records with relevant input, it still requires the application to read undesired columns in the target records. By changing the data layout and storing the input "by-column", it becomes possible to read only the columns that are relevant for the computation. Furthermore, since data in a column belongs to the same domain, this layout has the potential for offering better compression efficiency.

An interesting approach that, to the best of our knowledge, has not been explored would be to combine indexes with a column-oriented data layout: indexes do the filtering on the data to be scanned and a column-wise layout allows to achieve good data compression and to read only the needed column data, reducing even more the I/O costs.

Data co-location has also proved to be an useful technique, not only to efficiently support joins but also to optimize the storage of data using a column-oriented layout (by storing columns of the same set of records on the same nodes).

One important aspect of most of the systems we have surveyed is that they require prior knowledge of the schema definition and of the query workload. Without a predefined schema, the indexing, data layout, and co-location optimization mechanisms cannot be applied. CoHadoop does not require the schema explicitly to optimize the co-location but requires user to tag data with domain-specific information.

In addition, with the exception of LIAH (which works online and incrementally), these optimization techniques require the execution of a pre-processing phase, where indexes are built, the data layout is changed, or the data is moved to ensure proper co-location. In Hadoop++, for instance, the input data is first transferred through the network and rewritten again into the HDFS which leads to a significant loading cost. Still, according to [20], most applications are willing to pay a one-time loading cost to reorganize the data if the dataset are going to be analysed multiple times. It is also worth noting that the adaptive approach of LIAH could be integrated into other approaches such Trojan Data Layouts, to interpret the access patterns and adapt the existing column groups to better serve the workload of future and similar queries.

Despite of Hadoop having a strict processing pipeline with a structure hard to modify, UDFs allows to inject arbitrary code into this framework turning it into a more flexible distributed runtime. In fact, these UDFs may suffice for

Table 2: Systems

		Hadoop++ [7]		LIAH [21]	E3 [10]	Full-Text Indexing [18]	Trojan Data Layouts [15]	RCFile [13]	CIF [11]	CoHadoop [9]
		Trojan Index	Trojan Join							
Approach		Access only the relevant records	Process join operations locally	Access only the relevant records	Access only the relevant splits	Access only the relevant blocks	Access only the relevant columns	Access only the relevant columns	Access only the relevant columns	Store related data on the same node
Use Cases		Selection	Joining	Selection	Selection	Selection	Selection	Selection	Selection	Joining
Techniques										
Indexing	**Yes/No**	Yes	No	Yes	Yes	Yes	No	No	No	No
	Index-level	Record	-	Record	Split	Block	-	-	-	-
	Multiple/Single	Single	-	Multiple	Multiple	Single	-	-	-	-
Data Layout	**Data Organization**	Row	Row	PAX	Row	Row	Column groups	PAX	Column	Raw data
	Data Compression	No	No	No	No	Yes	No	Yes	Yes	No
Co-location		-	Based on join attribute	-	-	-	-	-	Based on split-directory	Based on locator property
Overall										
Changes Hadoop		No	No	Yes	No	Yes	Yes	No	No	Yes
Previous Knowledge		Schema and Workload	Schema and Workload	Schema	Schema	Schema	Schema and workload	Schema	Schema	Hints about related files
Pre-Processing		Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Static/Online		Static	Static	Online	Online (cache)	N/A	Static	N/A	N/A	Online
Record Reconstruction		-	-	Within HDFS block	-	N/A	Within HDFS block	Within HDFS block	From different files but with a skip list	-
Overheads		Reads irrelevant splits and attributes	-	Reads irrelevant HDFS blocks	Storage; Reads irrelevant attributes	Extensive manual configuration	Reads irrelevant HDFS blocks	Storage; Reads irrelevant HDFS blocks	Reads irrelevant HDFS blocks	-

some systems we presented such Hadoop++ or CIF, but not for others that needed to tune other aspects of Hadoop. In the case of LIAH, the changes on the framework were needed to index the data online according to the current workload and without any extra I/O operation. CoHadoop and the Trojan Data Layouts on the other hand had to add extra metadata information to the HDFS.

6 Proposed Approach

Known for its simplicity and ease deployment, the MapReduce framework exhibits some limitations that preclude it from reaching the performance of parallel DBMS. Parallel DBMS's are designed for highly structured relational databases which enables to have indexes optimized for accessing the data. MapReduce on the other hand has the flexibility of processing either unstructured or relational data, making this framework widely adopted for many sorts of applications. This flexibility however comes at the cost of performance.

The efficiency of the input data processing phase has been addressed by many research efforts by means of indexing, data layout, or co-location approaches. The first two techniques aim at reducing disk I/O costs whereas the latter aims at reducing the network usage.

In this project, we aim at building a system using Hadoop to integrate the three techniques enumerated above, to combine their advantages:

- Firstly, we propose to store the input in a column-wise fashion: dividing it into row groups, each one then vertically partitioned into either individually or groups of columns that are stored in separate HDFS blocks. By storing the data in a column-wise fashion we expect to achieve good compression rates and allow read operations to be performed exclusively over the column blocks corresponding to the requested attribute, resulting in reduced I/O costs when loading data from disk.
- Secondly, inspired by the CIF approach, we plan to co-locate the column blocks belonging to the same row group on the same set of nodes, ensuring a low cost of record reconstruction.
- Finally, we will build inverted indexes that will enable to select all column blocks that have, at least, one attribute value that satisfies the job predicate.

It is worth noting the difference between this architecture and the ones from previously presented systems that also resort to indexing: instead of indexing entire records, we aim at indexing columnar data. In order to speed up all the selective queries no matter which attribute is referred, we intend to keep as many inverted indexes as attributes. The resulting system is expected to perform well mainly for selection-based jobs. However, queries that apply a filtering following by a join may also benefit from our architecture.

We expect to preserve the fault tolerant property of MapReduce, since data will still be stored in data blocks, we only modify MapReduce to store column data instead of row data.

The challenges of our proposal are mainly associated with the size of the row groups and the indexes: as referred in [11, 13], tuning a row group size becomes critical. A large row group size can have better compression efficiency but lower read performance. Concerning the indexes, we think it would be advantageous to have indexes that made possible to decompress only the relevant part of a data block. For this, each entry of an index would have to keep the attribute value, the data block and an offset for that value on the associated block. To avoid a significant storage overhead due to this additional information, we could consider to index only a subset of the blocks (for instance, the most accessed ones). Another relevant aspect is that, similarly to CIF, we have to implement a new block-placement policy for HDFS in order to co-locate associated column data blocks. However, this does not require any modification to Hadoop, since from version 0.21.0 the Hadoop system allows changing its placement policy without modifying Hadoop or HDFS [11]. Regarding the creation of indexes, we can resort to mechanisms similar to those of Hadoop++[7] (but for columnar data instead of row), to implement indexes without modifying the core of Hadoop. Taking this into account, we expect to build our described architecture keeping the HDFS and the Hadoop processing framework unchanged.

7 Evaluation

We intend to evaluate the performance of our prototype and compare it against Hadoop and a simpler version of our system that corresponds to a prototype of CIF. Our main goal is to show that we can speed up analytical MapReduce jobs having selection predicates.

The experiments will be carried out on a cluster with 10 nodes. The nodes are equipped with an Intel Xeon CPU with 8 cores, 40GB main memory, and 1TB disks. The operating system is Ubuntu Linux. In our experiments we will use the Hadoop (2.2.0).

The dataset consists of 325 million entries (*tweets*). Each tweet has multiple attributes such as its *ID*, its creation date, the text itself, information about its owner, a list of *Url's* and users that were mentioned on it, the location it was sent, among others.

In terms of query set, we identify two types of selective predicates that allow us to adequately compare our system, Hadoop and CIF: 1) queries scanning all the tweets of the dataset (example: return the owners and the dates from all tweets) and 2) queries requiring only the tweets with attributes that fulfil a specified criterion (example: return the owners and dates from tweets originated at New York).

To measure the effectiveness of our proposal, we propose the following evaluation metrics:

- The time spent in the map phase for being the moment when data is actually scanned. CIF reaches a performance of 60x that of Hadoop when scanning one single attribute from all the records of the dataset. We expect a similar

performance from our system when executing this type of jobs. However, in the second type of selective predicates identified above, CIF requires to fully read all blocks storing data from the columns being tested. With our architecture, the indexing capability allows to skip those blocks with no data of interest, avoiding to fetch, uncompress, and fully read them. In such cases, we expect to outperform CIF.
- The time spending in uploading the input data, i.e. how much time is spent organizing the dataset into column data blocks, co-locating them and creating the indexes. Due to the costs of reorganizing data, we expect that our system will perform worse than Hadoop, but a similar time as CIF in this stage. However, since our system requires a phase of index creation, we expect a higher overhead in the total data ingestion time.
- The storage overhead caused by indexes and additional information that may be required. We expect to not have a considerable storage overhead when compared with uncompressed Hadoop, since the compression of column-oriented data may balance the additional data that is require to implement our architecture.

8 Scheduling of Future Work

Future work is scheduled as follows:

- January 9 - March 29, 2014: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 30 - May 3, 2014: Perform the complete experimental evaluation of the results.
- May 4 - May 23, 2014: Write a paper describing the project.
- May 24 - June 15, 2014: Finish the writing of the dissertation.
- June 15, 2014: Deliver the MSc dissertation.

9 Conclusions

Due to the increasing importance of being able to process big data, MapReduce has become widely adopted, mostly due to its scalability and ease of use. In this work we surveyed systems aimed at overcoming limitations of the basic MapReduce model in what regards the access to data. Examples of techniques that have been incorporated in MapReduce include implementing indexes, optimizing the data layout, or co-locating related data on the same location.

Our goal is to further enhance the performance of MapReduce when executing selective-based jobs by creating a system that combines and tunes the strengths of indexes, data layouts, and co-location minimizing the cost of data accesses.

We have drafted an architecture of the proposed solution and identified the metrics that will be used to evaluate it. More details related with the specification, implementation, and experimental evaluation are left for future work whose schedule has also been presented.

Acknowledgements We are grateful to João Paiva for the fruitful discussions and comments during the preparation of this report. This work was partially supported by project PEPITA PTDC/EEI-SCR/2776/2012.

Bibliography

[1] http://hadoop.apache.org.

[2] https://twitter.com.

[3] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376712. URL http://doi.acm.org/10.1145/1376616.1376712.

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-804-4. URL http://dl.acm.org/citation.cfm?id=645927.672367.

[5] Amitanand S. Aiyer, Mikhail Bautin, Guoqiang Jerry Chen, Pritam Damania, Prakash Khemani, Kannan Muthukkaruppan, Karthik Ranganathan, Nicolas Spiegelberg, Liyin Tang, and Madhuwanti Vaidya. Storage infrastructure behind facebook messages: Using hbase at scale. *IEEE Data Eng. Bull.*, 35(2):4–13, 2012. URL http://dblp.uni-trier.de/db/journals/debu/debu35.html.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

[7] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1-2):515–529, September 2010. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=1920841.1920908.

[8] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only aggressive elephants are fast elephants. *CoRR*, abs/1208.0287, 2012.

[9] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. *Proc. VLDB Endow.*, 4(9):575–585, June 2011. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=2002938.2002943.

[10] Mohamed Y. Eltabakh, Fatma Özcan, Yannis Sismanis, Peter J. Haas, Hamid Pirahesh, and Jan Vondrak. Eagle-eyed elephant: Split-oriented indexing in hadoop. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 89–100, New York, NY,

USA, 2013. ACM. ISBN 978-1-4503-1597-5. doi: 10.1145/2452376.2452388. URL http://doi.acm.org/10.1145/2452376.2452388.

[11] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. Column-oriented storage techniques for mapreduce. *Proc. VLDB Endow.*, 4(7):419–429, April 2011. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=1988776.1988778.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, October 2003. ISSN 0163-5980. doi: 10.1145/1165389.945450. URL http://doi.acm.org/10.1145/1165389.945450.

[13] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1199–1208, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-8959-6. doi: 10.1109/ICDE.2011.5767933. URL http://dx.doi.org/10.1109/ICDE.2011.5767933.

[14] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. *Proc. VLDB Endow.*, 3(1-2):472–483, September 2010. ISSN 2150-8097. URL http://dl.acm.org/citation.cfm?id=1920841.1920903.

[15] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. Trojan data layouts: Right shoes for a running elephant. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 21:1–21:14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038937. URL http://doi.acm.org/10.1145/2038916.2038937.

[16] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, September 1999. ISSN 0004-5411. doi: 10.1145/324133.324140. URL http://doi.acm.org/10.1145/324133.324140.

[17] Kyong-Ha Lee, Yoon-Joon Lee, Hyunsik Choi, Yon Dohn Chung, and Bongki Moon. Parallel data processing with mapreduce: a survey. *SIGMOD Rec.*, 40(4):11–20, January 2012. ISSN 0163-5808. doi: 10.1145/2094114.2094118. URL http://doi.acm.org/10.1145/2094114.2094118.

[18] Jimmy Lin, Dmitriy Ryaboy, and Kevin Weil. Full-text indexing for optimizing selection operations in large-scale data analytics. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 59–66, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0700-0. doi: 10.1145/1996092.1996105. URL http://doi.acm.org/10.1145/1996092.1996105.

[19] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. URL http://ilpubs.stanford.edu:8090/422/. Previous number = SIDL-WP-1999-0120.

[20] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-551-2. doi: 10.1145/1559845.1559865. URL http://doi.acm.org/10.1145/1559845.1559865.

[21] Stefan Richter, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. Towards zero-overhead adaptive indexing in hadoop. *CoRR*, abs/1212.3480, 2012.

[22] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7152-2. doi: 10.1109/MSST.2010.5496972. URL http://dx.doi.org/10.1109/MSST.2010.5496972.