Efficient Support for Selective MapReduce Queries

(extended abstract of the MSc dissertation)

Manuel da Silva Santos Gomes Ferreira
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***Abstract*—The work described in this thesis proposes and evaluates ShortMap, a system that relies on a combination of techniques aimed at efficiently supporting selective MapReduce jobs that are only concerned with a subset of the entire dataset. We combine the use of an appropriate data layout with data indexing tools to improve the data access speed and significantly shorten the Map phase of the jobs. An extensive experimental evaluation of ShortMap shows that, by avoiding reading irrelevant blocks, it can provide speedups up to 80 times when compared to the basic Hadoop implementation. Further, our system also outperforms other MapReduce implementations that use variants of the techniques we have embedded in ShortMap. ShortMap is open source and available for download.**

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 such 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.

The MapReduce paradigm[1], and its associated middleware, have become a fundamental tool to parallelize complex computations over large amounts of data. In this context, MapReduce implementations, such as Hadoop[2] have become "de facto" standard middleware frameworks that strongly simplify big data processing.

Originally, MapReduce was designed for jobs such as web indexing, that need to process the entire dataset [3]. However, as the range of applications of MapReduce grows, it is frequently used to execute queries that are only concerned with a small fraction of the entire dataset [4], [5]. For instance, communications service providers maintain datasets about their customers that they can use as an additional source of revenue by selling analysis of data to third parties for market research[6].

Unfortunately, current MapReduce implementations are not well tailored to support this type of operation. According to the MapReduce model, it is the role of the Map task to select the appropriate entries of the dataset, that are relevant to the computation being performed, and pass those entries to reducers. To perform this selection, the Map task may be forced to read the entire dataset, even if only a small fraction is relevant for the query being performed. This can consume a significant fraction of the entire MapReduce job. Figure 1 illustrates the cost of reading the input data, in comparison with the cost of the complete Map phase, for two types of jobs and two types of selective queries when using the default Hadoop implementation over a real Twitter dataset. The "wordcount" job is a common example of a Hadoop workload, and consists of counting the frequency of words in the text; the "sentiment" job consists of calculating the sentiment of twitter users, a realistic workload which mimics the big data processing required to extract scientific results from Twitter datasets[7], [8], [9], [10]. As it can be seen, when querying for all but the most common item, the Map phase can take from 60% to 99% of the total querying time. In fact, previous research has already identified several map-heavy workloads which justify Map task optimization as an interesting research area [4], [11], [5], [12], [13], [14], [15].

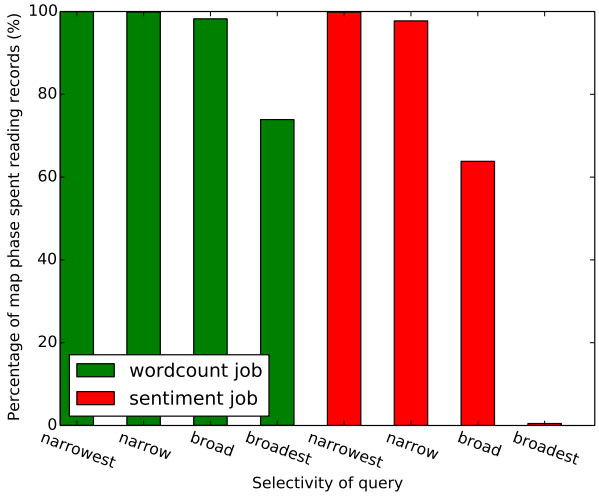

Figure 1. Cost of reading data, relative to the duration of Map tasks, depending on job type and query selectivity. The "broadest" query filters records by the most common value in the dataset, while the "narrowest" query filters by the most uncommon one.

A common characteristic of many big data datasets, that motivates our work, is that the distribution of the frequencies of values for any attribute typically follows a highly skewed distribution such as a Zipfian distribution [16]. This is illustrated in Figure 2, that presents the distribution of frequencies of the "user location" attribute in a sample of

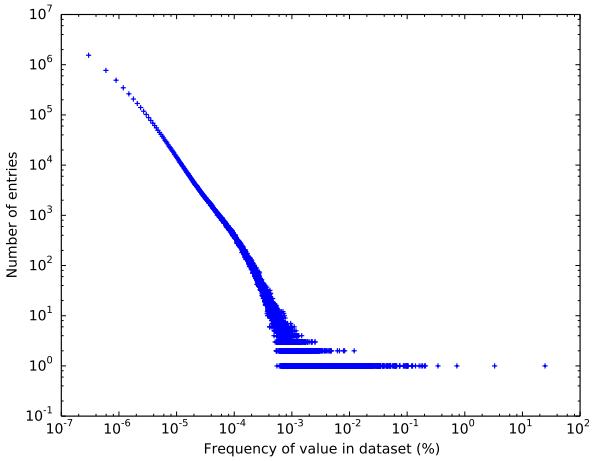

Figure 2. Distribution of the "user location" values in a realistic Twitter dataset.

the real Twitter dataset used to evaluate ShortMap. This kind of distributions is particularly amenable for being indexed since, even though a small percentage of the values are very frequent in the dataset, most are uncommon. This suggest that, with the appropriate use of indexing mechanisms, high gains may be achieved by avoiding reading all data when using selection queries by all but the most common values.

In this paper we show how the time of the Map phase can be substantially reduced. Specifically, we propose and evaluate ShortMap, a system that relies on a combination of techniques aimed at efficiently supporting selective MapReduce queries that are only concerned with a subset of the entire dataset. We combine the use of an appropriate data-layout with the use of data indexing tools to significantly shorten the Map phase of the jobs. An extensive experimental evaluation of ShortMap shows that, by avoiding reading irrelevant data, it can provide speedups up to 80 times when compared to the basic Hadoop implementation. Further, our system also outperforms other MapReduce implementations that use variants of the techniques we have embedded in ShortMap.

The rest of the paper is structured as follows. For self-containment, Section I provides a short introduction to MapReduce and associated middleware. Sections II and III describe the architecture and the algorithms used by ShortMap, while Section IV captures some relevant implementation details. Section V provides a extensive experimental evaluation ShortMap. A comparison of ShortMap with related work is given in Section VI and Section VII concludes the paper.

I. MapReduce Background

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). Any node in the system may act as a mapper,

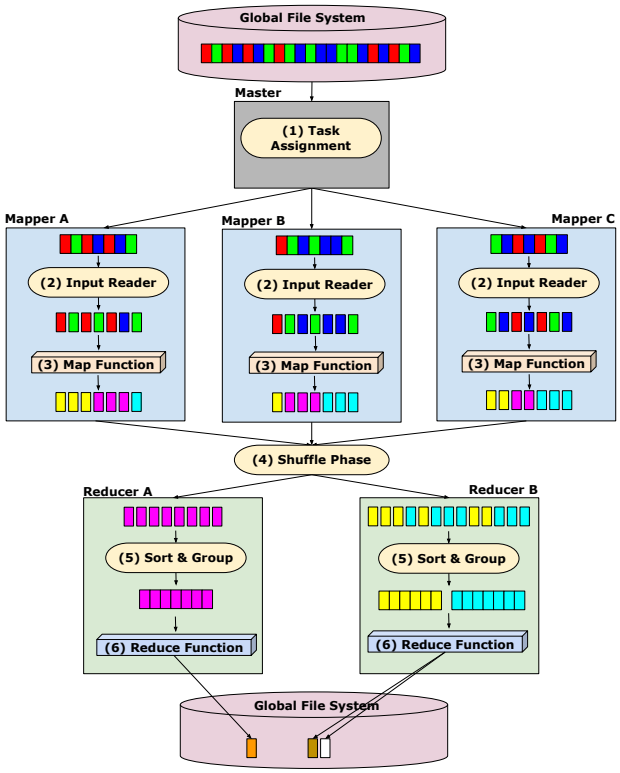

Figure 3. MapReduce Steps.

a reducer or both, and the whole process is orchestrated by a (logically) centralised master.

Upon job submission, the input data is automatically partitioned into pieces, called splits. Each Map task processes a single split applying the Map function to the data contained in it. An input reader converts the data into a stream of key/value pairs which correspond to records to be processed by the Map function. The Map function then takes as input a key/value pair and, after some computation, generates an intermediate key/value pair, which is then assigned to the reducer nodes.

Each reducer will get all intermediate key/value pairs that share the same intermediate key and apply 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.

Figure 3 illustrates the sequence of steps executed during the execution of a MapReduce job, which are enumerated below:

1) **Task assignment**: The input data is divided into splits and, for each one, the master creates a Map task and assigns it to a worker.
2) **Input Reader**: A Map task starts with a function to extract the key/value pairs from the raw data inside the split.
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) **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 processing nodes, the master passes to reducers the information about the mappers's location, so that each reducer may be able to remotely read its input.
5) **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.
6) **Reduce function**: In the Reduce tasks, each reducer passes an intermediate key 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 reliability.

Apache Hadoop [2] is the most popular open-source implementation of the MapReduce framework. 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 nodes; and the Hadoop MapReduce Framework, a processing layer responsible for running MapReduce jobs.

- *Storage Layer:* Hadoop DFS (HDFS) is a distributed file system that uses three different entities: one NameNode, one SecondaryNameNode and one or more DataNodes. The 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. In order to improve availability, Hadoop replicates each block by a default, but configurable, factor of 3.
- *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 uses the following task scheduling method: Workers are initially assigned with tasks that can be executed with local data. However, should a worker finish all tasks associated with the data it stores locally, it may be fed tasks which entail obtaining data from other nodes.

II. Main ShortMap Mechanisms

We now present ShortMap, a system that embodies a complementary and coherent set of techniques that are aimed at improving the Map phase of jobs that only manipulate a fraction of the dataset. ShortMap combines the following mechanisms:

- *Data-layout:* ShortMap organizes the dataset in a way that promotes locality. We achieve this by storing table contents by columns instead of by rows.
- *Data Grouping:* ShortMap groups similar data at each node, to improve the effectiveness of the indexing mechanism without compromising load-balancing.
- *Indexing:* ShortMap creates local indexes, from the data that is stored at each node. Indexes are created and maintained for the most relevant attributes.

We discuss the rationale, and the details, of each of these mechanisms in the following subsections.

A. Data-layout

In a MapReduce system, the input data can be modelled as a set of tables, where each table is composed of multiple columns, or attributes. One of these attributes, typically the first, is named the key and identifies each record, or row, of the table. Tables are typically very large and must be stored in multiple data blocks. There are mainly two approaches to map the table content into blocks: row-oriented or column-oriented.

In the row-oriented data layout all attributes of one record are stored sequentially, and multiple records are placed contiguously into disk. Row-based systems are designed to efficiently process queries where many attributes of a record need to be processed at the same time, since an entire record can be retrieved with a single access to disk.

On the other hand, it has been shown[17], that a column-oriented layout is particularly well suited for selection-based jobs that access a small number of columns, since columns that are not relevant for the intended output are not loaded from disk and filtered through by the job, reducing the execution time of a MapReduce job.

Unfortunately, since each column in a dataset may represent data with different lengths, a naive partition of the dataset in columns may cause column blocks to become unaligned, complicating the process of building whole records from partitioned datasets. In ShortMap, we avoid this drawback by first partitioning the dataset horizontally, by creating *row groups*, and only then each row group is vertically partitioned by columns, each one being stored in a different file (Figure 4). In this way an attribute file is only read when a given query refers to the corresponding attribute, skipping data belonging to the other attributes that are irrelevant for the query. Since by default HDFS places blocks in a random way across all data nodes, ShortMap also includes a row-group aware Block Placement Policy, to make sure that blocks corresponding to the same row group are placed in the same node. This allows our system to use a columnar layout without having to fetch data from other nodes when a full record is required by the job and must be reconstructed from several columns.

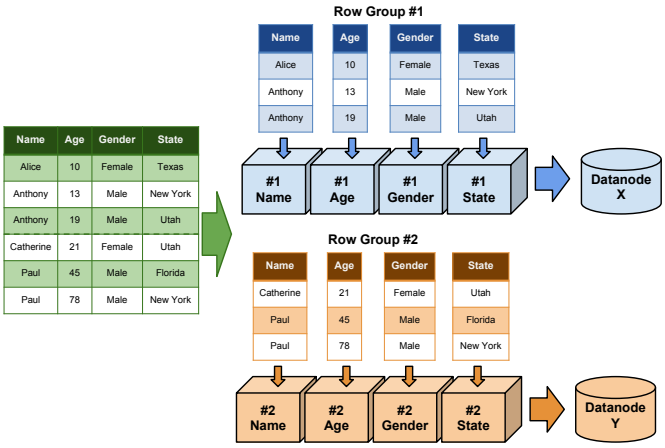

Figure 4. Row groups in ShortMap.

B. Grouping

The grouping component of ShortMap is responsible for rewriting the input data into a format which favours the usage of column-oriented indexes. In fact, columnar indexes may be quite ineffective without the use of some kind of grouping: On very large datasets, the frequency of most values is significantly larger than the number of row groups the data is partitioned into, which causes any value to be an index "hit" for most row groups.

To make sure that any value in the dataset will be present in the smallest number of blocks as possible, ShortMap reads the whole dataset, identifies records which share the same value for the indexed attributes, and outputs them in a new order such that records with the same value will be contiguous in the dataset. This re-ordering is done *locally*, for each node in the system. The use of local grouping has two significant advantages with regard to an alternative global sorting: not only it makes grouping inexpensive, but also preserves the good load balancing achieved by HDFS's initial distribution of data.

C. Indexing

ShortMap uses indexing to avoid loading the whole dataset into main memory during the Map phase. This is accomplished by using the index as an indicator if a given block is relevant for the job being processed.

ShortMap indexes are fully decentralised in the sense that each node builds its own index based on the data it stores. This design decision simplifies the creation of indexes, and makes for quick local index lookups. In terms of structure, the indexes are actually inverted indexes, where each entry maps an attribute value to its location in the dataset. In fact, there are two possible representations for this pair, each with different tradeoffs. The first is to map the attribute value to a row group identifier. This identifier represents the first row group where the value occurs. Since data in each node in ShortMap is grouped by attribute value, the system is guaranteed to scan through all records which have a given attribute if it starts scanning at the row group pointed by the index and stops as it reaches a different value. The second possible representation is to use a pair of $\langle$ *RowGroupID, [column1-offset, column2-offset, . . . , columnN-offset]* $\rangle$ attribute value. The first element of the pair points to the first row group which contains the attribute value. The second element is a list with length equal to the number of columns in the dataset, such that each entry corresponds the first offset containing the value within the block of the corresponding column. Even though the second representation has the potential to create larger indexes (since it must contain an entry for each column in the dataset), it also has the potential to achieve better performance as it allows ShortMap to directly retrieve a record from a specific offset in a block (without having to filter through all records which may appear before the target in the block).

In order to save memory during query time, the indexes are stored on persistent storage along with the data. The indexes are stored partitioned, such that a small set of pairs is kept on each file. Thus, when a task is scheduled at a node, only the index pairs corresponding to the attribute value queried for by the Map task are brought to memory from disk, to determine if the blocks associated with the split of the Map task should be read. In order to achieve efficient index querying, ShortMap keeps an in-memory cache of the most recent attribute values queried for using a least recently used policy. This design simplifies the management of indexes, since it allows a node to keep the index pairs in memory during the whole job, without requiring information on job completion (which in MapReduce is only available at the master, the JobTracker).

III. Other ShortMap Aspects

In the previous section we have described the main mechanism that we have incorporated in ShortMap. In this section we discuss additional issues, such as the use of replication, the storage of indexes, the validation of blocks before transfer, and the pre-processing step required by ShortMap.

A. Replication

Hadoop supports replication, allowing each block to be replicated in (a configurable number of) R nodes, such that when a node fails the data may be recovered from other nodes in the system. This mechanism is also leveraged when scheduling tasks, since when choosing a node where to spawn a task, the scheduler gives priorities to nodes which already store the data. ShortMap also allows each block to be replicated in R different nodes with similar advantages. However, since each node groups the data it stores, replication in ShortMap must be made on a per-node basis instead on a per-block basis. Furthermore, similarly to other state of the art systems [5], [12], ShortMap also allows each replica to group the data according to a different attribute, in order to allow for optimised queries for more than one attribute.

B. Index Storage

As mentioned previously, ShortMap's indexes are stored on disk and loaded as needed during querying. In order to be able to load portions of the index, index pairs are saved into different files (which act as buckets) depending on a fixed portion the hash value of the key (i.e. the value of the attribute).

C. Block Transfer Pre-Validation

Regarding index usage, notice that ShortMap does not make use of a centralised index but MapReduce's model supports nodes retrieving data blocks from other nodes in the system. In particular, when a node has finished processing all tasks related with its data, the JobTracker will assign it tasks related with data of nodes which have fallen behind in processing. When processing selective queries, this may cause a block with no relevant information to the worker to be transferred over the network. This is inefficient, specially because we have found the network to be one of the key bottlenecks in the execution of a MapReduce job (this observation concurs with other research in the area [11]).

To avoid this problem, in our implementation we modified Hadoop such that the node that hosts the data makes a local check, by loading the corresponding indexes to determine if the block is relevant or not, before transferring the block (avoiding the transference if it includes no relevant content to the requesting worker). The results presented later in the paper use a version of Hadoop with this optimisation. Note, however, that ShortMap is not tied to this implementation decision.

D. Pre-Processing

ShortMap requires the execution of a data pre-processing stage to build the indexes and re-format data blocks by splitting the rows into columns and grouping similar data. Since this pre-processing stage does not involve any exchange of information among nodes, it can be performed efficiently, and can actually leverage on MapReduce itself.

In more detail, after the data is loaded to HDFS, we run a pre-processing MapReduce job configured such that each node in the system acts as a mapper as well as reducer. The job goes through all lines in all input files (i.e. all records stored by the node), and in the Map phase outputs a pair $< (ID, value), List < field >>$ associating the record itself, formatted as a list of values for fields, with an intermediate key. The intermediate key makes sure that the record will be processed by the reducer co-located with the mapper, since it contains the identifier of the node storing the data; in order to allow the reducer to perform the grouping, the Reduce key also contains the value of the record for the attribute for which the data will be grouped. Since the attribute to be used for data grouping must be defined at the stage of pre-processing, the user is responsible for selecting which attribute is most relevant for indexing. In our prototype, this attribute is included in a configuration file, but in a working system it should be included as a parameter of the *copyFromLocal* command of HDFS.

At the Reduce phase, each reducer receives several sets of records, grouped by attribute value. Thus, the reducer's task is to output each field of the record to a different HDFS block. Notice that since each column contains different types of data, each output block may grow at a different rate. To guarantee that all blocks will fit in an HDFS block and that the several blocks corresponding to a row group are aligned with each other, as soon as any column's content reaches the size of an HDFS block, a new row group is created. This mechanism follows a design similar to other state of the art column-oriented storage systems [17], [18], [19]. Throughout this process, ShortMap captures the starting positions of each different value, such that it can populate the block index.

IV. ShortMap Implementation

ShortMap has been implemented as a set of extensions to the Hadoop framework and the prototype is available for download[1].

ShortMap extensions consist of roughly 6600 lines of code. In this section, we go through the most important tweaks done on Hadoop to support ShortMap functionality.

Regarding the loading of data into HDFS, we need to ensure that files belonging to the same row group were placed on the same node, in order to allow for local record reconstruction in cases where more than one attribute is queried. Since by default, HDFS places blocks in a random way across all data nodes, ShortMap includes a `Row-Group Aware Block Placement Policy` to make sure that blocks corresponding to the same row group are placed in the same node. No modifications on the core of Hadoop were needed to implement this feature, since custom Block Placement Policies are easily pluggable.

Before processing starts, the input data is divided into splits upon job submission, according to the `InputFormat` associated with the job being performed. The generated splits are then processed by the Map tasks. Since ShortMap is based on file manipulation, the most appropriate `InputFormat` is the `FileInputFormat` of Hadoop, that is the base class for all file-based `InputFormats`. However, the `FileInputFormat` goes through all the input files and creates one split per HDFS block. For ShortMap, we defined an`InputFormat` that is row-group aware, in order to create only one split per row-group instead of one per HDFS block.

In addition, the `FileInputFormat` also instantiates the `RecordReader`, which is responsible for generating the key/value pairs from the raw input split data and for sending them, one by one, to the Map function. For ShortMap, we implemented a `RecordReader` with two particular capabilities. Firstly, it performs a local index lookup to check if any row-groups assigned to the split contain a relevant entry. In the positive case, the `RecordReader` obtains the offset of the first row to start reading from that point until

[1]The prototype can be downloaded from:https://github.com/shortmap/shortmap

reaching either the end of the row-group or an entry that no longer satisfies the query. Since we allow for nodes to avoid transferring non-relevant blocks when other nodes are processing the data they store, we have also added a new routine that is called by the processing node so the owner of a given row-group performs the index lookup and sends it the relevant portion of the index the row group is relevant; or a negative answer otherwise. Secondly, the `RecordReader` extracts from the job configuration the attributes referred by the query and passes the values corresponding to those attributes to the Map function.

It is interesting to note that, although these changes can achieve significant improvements of the MapReduce performance (over the standard Hadoop implementation), the majority of them require no modifications on the core of Hadoop but, instead, can be implemented using the extensibility-hooks provided by Hadoop.

V. Evaluation

In this section, we show how each part of ShortMap contributes to its overall result by comparing with other state of the art solutions. Finally, we present results for how our system compares with an unmodified Hadoop version.

All experiments have been performed using a cluster of 20 virtual machines (deployed on 10 physical machines) running Xen, equipped with two 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 processors and 40 GB of RAM, running Ubuntu Linux 2.6.32-33-server and interconnected via a private Gigabit Ethernet.

For all experiments presented in the current section, we used a sample of the Twitter dataset[20], collected between May and September 2012. This dataset is comprised of 325,333,833 tweets, that correspond to 988GB of raw data, which when compressed with gzip, have a total of 161GB. The tweets are stored in JSON format, each containing 23 attributes (such as an identifier, creation date, hashtags, the text message itself, as well as an embedded JSON object with 38 more attributes about the owner of the tweet such as her language, location, identifier, etc). Due to non-disclosure restrictions imposed by Twitter, we are unable to make the dataset public, but a similar dataset can be obtained by querying the Twitter's API.

All values presented are the average of at least 3 executions, and for our configuration, an unmodified version of Hadoop takes between 1 and 3 hours to process a query.

Our workloads capture scenarios where a provider might offer its infra-structure for their clients to perform data analysis based on a portion of the dataset, such as the demographic they are interested on, a specific location, language or associated hashtag. We use two different types of selective MapReduce jobs; Each type of workload allows to analyse different aspects of the system, according to the amount of processing required by the corresponding Map function. The first workload consists in applying a selective query to the dataset, to retrieve only tweets using a specific language, and then apply a simple word count. This analysis works as a baseline for comparison with a second, more complex and realistic analysis, which consists on calculating the sentiment of users from the corresponding tweets' text. This job mimics the big data processing required to extract scientific results from Twitter datasets [7], [8], [9], [10], and it has significantly higher CPU processing requirements than the wordcount job.

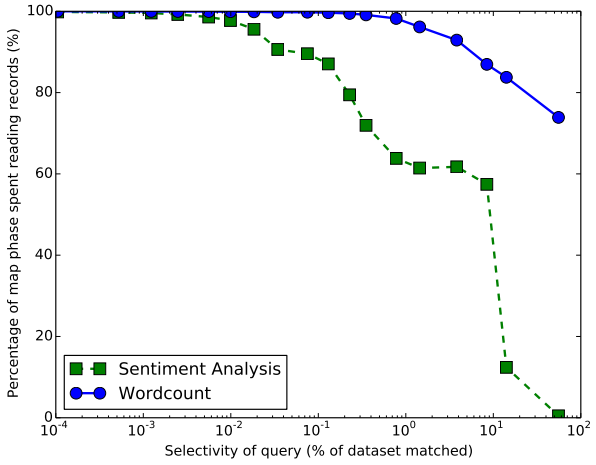

Figure 5. Cost of reading data, relative to the duration of Map tasks, depending on job type and query selectivity.

To better illustrate the difference between both types of jobs, Figure 5 shows the percentage of the Map phase time that ShortMap spends reading records from disk, when executing the sentiment analysis and the word-counting job. These values, similarly to others in this section, are presented as a function of the selectivity of the query, i.e., to what percentage of the dataset does the value queried by correspond. Since the word-counting job needs less data processing on the Map function, it spends a larger fraction (at least 73%) of the Map phase time reading the necessary records. Lighter jobs are therefore often bounded by the time required to scan the input data. Conversely, sentiment analysis requires more processing on the Map function. This is particularly prominent when processing the text of the users whose language is the most common in the dataset. In this case, since there is a large number of text messages to process, most of the Map phase time is spent processing records and not reading them from disk.

A. ShortMap against Indexed Row-Oriented Stores

One of the main concerns of ShortMap is Data-Layout. To achieve better efficiency when reading data, we use columnar storage. In order to evaluate the effect of this design decision, in this section we compare ShortMap with systems which make use of row-oriented layout for the storage and perform lookups using indexes.

A prominent and comparable solution to our system is Hadoop++,[4]. Hadoop++ not only uses a row-oriented layout, but also creates indexes for the items contained in each block. Unlike our solution, Hadoop++ sorts each block by itself, and creates a block-local index, which is

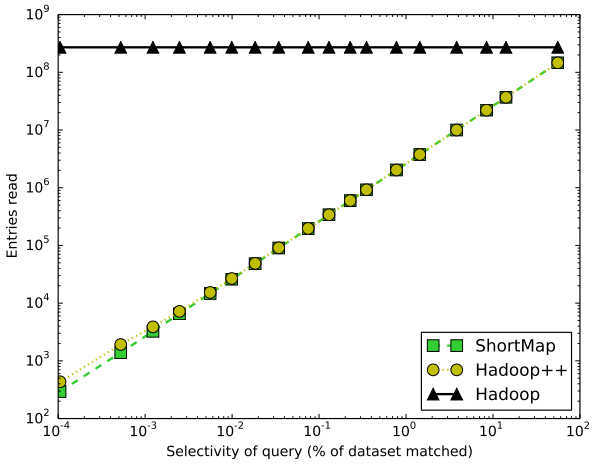

Figure 6. Entries read by Hadoop, Hadoop++ and ShortMap.

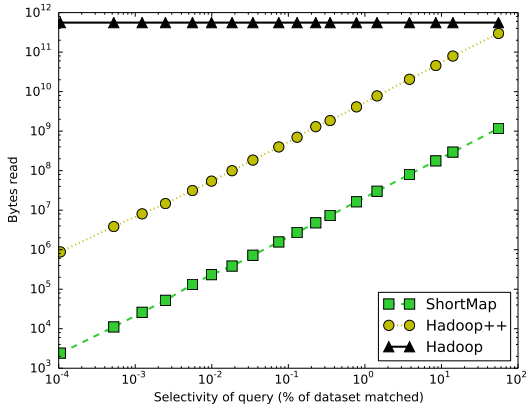

Figure 7. Bytes read by Hadoop, Hadoop++ and ShortMap.

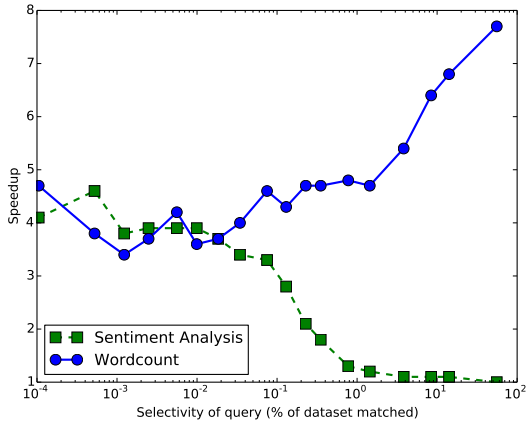

Figure 8. Speedup of ShortMap over the Hadoop++, when running the sentiment analysis and the word counting job.

appended to the block. In Hadoop++, this index is loaded at query time, and used as a hint to know which parts of the block should be read to answer the query at hand. Since the code for Hadoop++ is not publicly available, we have also implemented a prototype of this system following the specification provided in [4].

To better illustrate the design differences among ShortMap, Hadoop++, and Hadoop, Figures 6 and 7 show the number of bytes and entries read by each solution. Hadoop++ reads much less data and entries than Hadoop, and about the same number of entries as ShortMap (since its indexes allow it to bypass non-relevant blocks). Similarly to ShortMap, the number of bytes read is dependent on how common the value queried for is. In terms of read data, the main difference between both systems is that since Hadoop++ uses row storage, it must read all columns of records, whereas ShortMap needs only to read the blocks corresponding to the relevant columns.

Figure 8 shows how these decisions reflect in terms of performance of the system. This figure presents the speedup of ShortMap over Hadoop++ while querying for different values. As expected from the analysis of read data, for all executions, ShortMap exhibits speedups over Hadoop++. The performance of both systems tends to be similar when querying for more common items and performing a complex computation (i.e. sentiment analysis). On the other hand, for the more selective jobs ShortMap shows up to 5 times speedups, due to reading less data as was shown previously.

Interestingly, when for the wordcount job, an unexpected effect comes into play and ShortMap's speedup actually increases as we look at less selective queries. This result is due to the fact that Hadoop++, for the most common attributes, needs to read all the blocks from the dataset. In fact, even not all the records from the blocks are relevant for this query (only 50% of the records are relevant), all the blocks contain relevant records since they are uniformly distributed among blocks. This way, for less selective queries, the performance of Hadoop++ reaches the one of Hadoop, while ShortMap still avoids to read the remainder columns. Overall, for the results presented, ShortMap achieves an average speedup of 4.4 over Hadoop++ for the word-count job, and of 2.4 for the sentiment analysis.

B. *ShortMap against Indexed Column-Oriented Stores.*

Several state of the art systems opt by using column layouts along with per-block indexes [12], [5]. In this section, we study how our system compares with a variant of our implementation of Hadoop++ which instead of storing data per rows, stores it in columnar row groups (similarly to ShortMap), and indexes each row group individually (unlike ShortMap, which groups data and indexes data on a per-node basis). Since the code for neither LIAH [12] nor HAIL [5] is not publicly available, this prototype represents a simplified version of such systems. The main goal of the study presented in this section is to evaluate the effect of the grouping and per-node index component of ShortMap.

Figure **??** presents the speedup of ShortMap over an indexed column-oriented store. As was observed before, since the less selective queries of the sentiment analysis

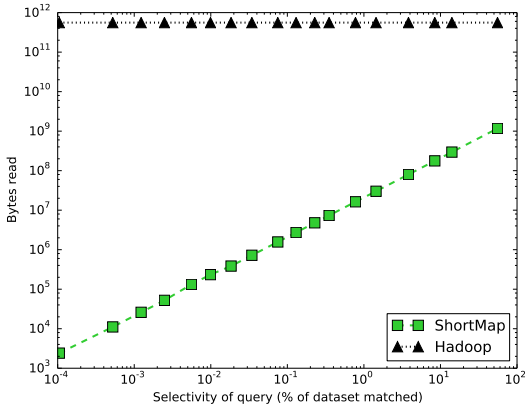

Figure 9. Comparison between the bytes read by Hadoop and ShortMap.

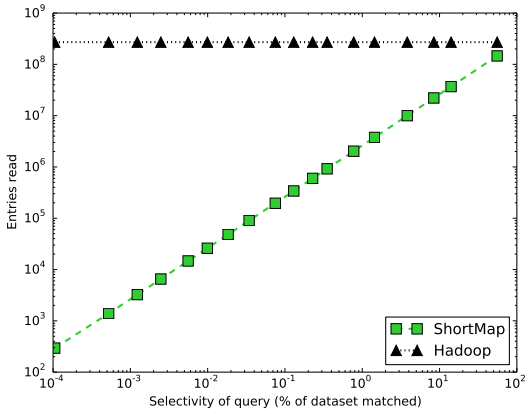

Figure 10. Comparison between the number of records read by Hadoop and ShortMap.

job are heavily CPU-bound, ShortMap does not present a significant advantage over other solutions. Still, for this job ShortMap can reach up to 50% better performance, and achieves an average of 25% speedup. Two facts contribute to these results. Firstly, using per-block indexes requires the system to load several indexes from disk while processing the job, in contrast with ShortMap which only loads a partial index on the first task associated with the job. Secondly, and more importantly, since the data is not grouped, several blocks may be a match for the query value, which requires the system to open several blocks, to seek to the offset of the value, whereas ShortMap most likely will only load a single block.

For the wordcount job, the costs of not using grouping become more pronounced. Unlike the results achieved for the sentiment analysis, the speedup of ShortMap over an indexed column-oriented store actually increase when the selectivity of the query decreases. These results are strongly tied with how many blocks are matched by the query. For ShortMap, the number of blocks matched is proportional with the selectivity of the query. However, on a system that does not rely on grouping, the number of blocks matched

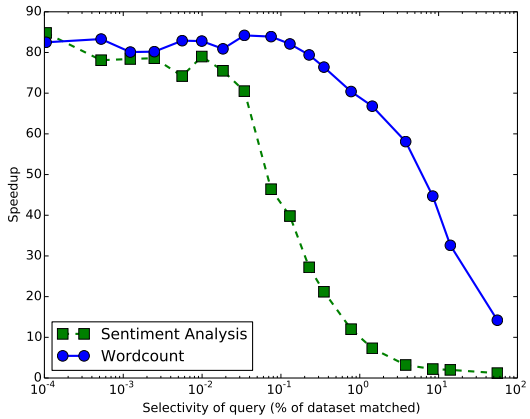

Figure 11. Speedup of ShortMap over the unmodified Hadoop, when running the sentiment analysis and the word-count job.

grows superlinearly with the selectivity of the query since the records containing the matched value are distributed randomly across all blocks. This effect is particularly notorious towards the less selective queries, since the number of blocks matched by the query on ShortMap remains very small whereas for the system using per-block indexes all blocks are a match for the queries. This also explains why towards the most common attributes the speedup of ShortMap drops to values close to 1: for this situation, in both systems, a large number of blocks is matched and the cost of reading the data shadows that of seeking inside the blocks. Furthermore, also notice that for the rarest item, the speedup of ShortMap is also more reduced than for the following queries: this is due to this value being present only on a single block for both systems, yielding a similar performance (with a slight edge to ShortMap due to reading only a single index).

C. ShortMap against Hadoop

In this section, we evaluate the overall performance of ShortMap, by comparing executions of our prototype of ShortMap with the unmodified version of Hadoop. We compare both solutions in terms of the amount of data read from disk, as well as the performance for the two query types for different frequencies of values in the dataset.

Unlike Hadoop, ShortMap does not require a full scan over the entire dataset when performing a selective query. As shown in Figure 9, this results in ShortMap reading significantly less data than Hadoop. Furthermore, ShortMap is required to filter through less entries, which cuts the CPU cost as well, as shown in Figure 10. Notice that even when querying for the most common attribute value, which corresponds to approximately half of the dataset, ShortMap still reads much less data than Hadoop, since it only reads the columns relevant for the query at hand. The impact of this improvement is more notorious in jobs with larger fractions of time spent reading the records, since this is the part of the Map phase time that is shortened.

Figure 11 depicts the speedup of ShortMap over Hadoop, depending on the frequency for which the queried value

can be found in the dataset. The consequences of ShortMap reading considerably less data than Hadoop can clearly be observed in these results, since ShortMap achieves up to almost two orders of magnitude faster queries. As expected from the analysis of read data, the advantages of ShortMap are less significant when querying for more common values, since the gap between the data read by ShortMap and Hadoop closes.

Still, even when querying for the most common attribute, for the word-count job ShortMap reaches a 14.2 times speedup while reading 460 times less data, 1.79 times less entries, and processing the same number of entries. In this scenario, given that many entries need to be processed, the processing costs, albeit small, limit the speedup of ShortMap. This fact is particularly evident when looking at the sentiment analysis: ShortMap is only 1.2 times faster than Hadoop for the most common value since in that situation the performance of the system is mostly limited by the processing cost. Furthermore, notice that due to the higher CPU cost associated with sentiment analysis, the speedup of ShortMap also drops earlier (i.e. for values less common) than for the word-count job. These results lead to the conclusion that even though ShortMap is particularly well suited for selection jobs using uncommon values, the usage of a columnar layout allows it to achieve good speedups even when selecting by the most common values in the dataset.

VI. Related Work

Similarly to ShortMap, several state of the art systems take advantage of different data layouts to bring the performance of Hadoop close to that of parallel database management systems (DBMS). The three main approaches in the literature to handling Data-layout are row-oriented layouts [4], [21], [22], column-oriented layouts [18], [17] and PAX format [12], [19]. Since our system is designed to handle queries for parts of the data, row-oriented layouts are not well suited for storing data since they have a higher overhead when retrieving partial records (as shown in Section V-A). Column-oriented layouts, similarly to PAX, support partial reads of the dataset. The main difference between these two layouts is that while column-oriented layouts write each column of a row group in a different file, PAX writes all columns in a single file and includes a metadata header to allow clients to seek directly to specific columns. ShortMap makes use of columnar layout, mainly because the system is designed to store large compressed data, and placing all columns in a single file increases information entropy, thus reducing compression ratios. Should this not be the case, modifying ShortMap to use PAX instead would involve little more than a trivial change to the indexes, to use several offsets within a single block instead of one offset per column block.

Indexing is a technique commonly used in DBMSs, which was introduced in MapReduce as a mechanism to allow skipping records when reading input during selection queries. Hadoop++ [4] proposes a Trojan Index, which is created on a per-block basis and appended to the block. This index allows MapReduce to read the index before loading the block from disk, thus reducing the number of records retrieved from disk when compared with Hadoop. More recent works such as HAIL [5] and LIAH [12] improve over this mechanism by re-formatting each block in a PAX layout, and creating the indexes in an on-demand way by tracking the queries performed in the system. Unlike our work, these systems create a per-block index, which as was shown in Section **??** leads to different tradeoffs in terms of index size, but also limits the performance of the system since it requires loading an index per block (as shown in Section V-B). Similarly to our system, the work by Lin et al. [22] uses inverted indexes to map attribute values to blocks in the dataset. This approach has the potential to avoid reading blocks of the dataset when performing selection queries. However, as we argue in Sections II-B and V-B, indexing tends to be an ineffective technique when not combined with block rewriting using grouping to avoid a large part of the blocks being a match for the query.

VII. Conclusions

In this paper we have described the design, implementation, and experimental evaluation of ShortMap, a system that significantly improves the performance of MapReduce jobs that are concerned with just a subset of the entire dataset. Our experimental results, obtained with a sample of a real dataset, show that ShortMap can provide speedups up to 80 times the default Hadoop implementation. Naturally, better results are obtained for queries that target uncommon values, but our results show that ShortMap does not incur performance degradation even when querying for the common attributes; actually, it still provides some (although arguably small) benefits in the less favourable cases. We have also extensively compared ShortMap against other state of the art MapReduce implementations that have materialised techniques similar to ours, although in different forms. Our results show that ShortMap also outperforms those competing solutions. ShortMap has been implemented as open source and has been made available for others to experiment and to improve upon. As future work we would like to leverage on the insights on frequencies of attribute values available in the indexes to build a better load balancing for ShortMap. We believe this frequency information could be leveraged to increase the size of the splits used by Hadoop for queries with high selectivity, thus reducing the management costs involved in creating Map tasks.

Acknowledgments

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) via the project PEPITA (PTDC/EEI-SCR/2776/2012) and via the INESC-ID multi-annual funding through the PIDDAC Program fund grant, under project PEst-OE/EEI/LA0021/2013.

Parts of this work have been performed in collaboration with another member of the Distributed Systems Group at INESC-ID, João Paiva.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[2] http://hadoop.apache.org. [Online]. Available: http://hadoop.apache.org

[3] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep. 1999-66, November 1999. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[4] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 515–529, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1920841.1920908

[5] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad, "Only aggressive elephants are fast elephants," *CoRR*, vol. abs/1208.0287, 2012.

[6] J. van der Lande, "Big data analytics: Telecoms operators can make new revenue by selling data, http://www.analysysmason.com/," Apr. 2013. [Online]. Available: http://www.analysysmason.com/

[7] I. Kloumann, C. Danforth, K. Harris, C. Bliss, and P. Dodds, "Positivity of the english language," *PloS one*, vol. 7, no. 1, p. e29484, 2012.

[8] L. Mitchell, M. Frank, K. Harris, P. Dodds, and C. Danforth, "The geography of happiness: Connecting twitter sentiment and expression, demographics, and objective characteristics of place," *PloS one*, vol. 8, no. 5, p. e64417, 2013.

[9] M. Frank, L. Mitchell, P. Dodds, and C. Danforth, "Happiness and the patterns of life: a study of geolocated tweets," *Scientific reports*, vol. 3, 2013.

[10] K. Bertrand, M. Bialik, K. Virdee, A. Gros, and Y. Bar-Yam, "Sentiment in new york city: A high resolution spatial and temporal view," *arXiv preprint arXiv:1308.5010*, 2013.

[11] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: Locality-aware resource allocation for mapreduce in a cloud," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 58:1–58:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063462

[12] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich, "Towards zero-overhead adaptive indexing in hadoop," *CoRR*, vol. abs/1212.3480, 2012.

[13] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 94–103. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2010.112

[14] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855744

[15] M. Lin, L. Zhang, A. Wierman, and J. Tan, "Joint optimization of overlapping phases in mapreduce," *Perform. Eval.*, vol. 70, no. 10, pp. 720–735, Oct. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.peva.2013.08.013

[16] G. Zipf, *The Psychobiology of Language*. Boston, MA: Houghton Mifflin, 1935.

[17] A. Floratou, J. M. Patel, E. Shekita, and S. Tata, "Column-oriented storage techniques for mapreduce," *Proc. VLDB Endow.*, vol. 4, no. 7, pp. 419–429, Apr. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1988776.1988778

[18] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich, "Trojan data layouts: Right shoes for a running elephant," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 21:1–21:14. [Online]. Available: http://doi.acm.org/10.1145/2038916.2038937

[19] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. 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*, ser. ICDE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1199–1208. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2011.5767933

[20] https://dev.twitter.com/docs/platform objects/tweets. [Online]. Available: https://dev.twitter.com/docs/platform-objects/tweets

[21] M. Eltabakh, F. Özcan, Y. Sismanis, P. Haas, H. Pirahesh, and J. Vondrak, "Eagle-eyed elephant: Split-oriented indexing in hadoop," in *Proceedings of the 16th International Conference on Extending Database Technology*, ser. EDBT '13. New York, NY, USA: ACM, 2013, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/2452376.2452388

[22] J. Lin, D. Ryaboy, and K. 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*, ser. MapReduce '11. New York, NY, USA: ACM, 2011, pp. 59–66. [Online]. Available: http://doi.acm.org/10.1145/1996092.1996105