2015 IEEE 7th International Conference on Cloud Computing Technology and Science

SmartFetch: Efficient Support for Selective Queries

Manuel Ferreira*, João Paiva*, Manuel Bravo*†, Luís Rodrigues*
*INESC-ID Lisboa, Instituto Superior Técnico, Universidade de Lisboa
†Université Catholique de Louvain, Belgium

***Abstract*—The paper proposes SmartFetch, a storage strategy that relies on a combination of techniques aimed at efficiently supporting selective jobs that are only concerned with a subset of the entire dataset in systems such as Hadoop and Spark. We combine the use of an appropriate data-layout with data indexing tools to improve the data access speed and significantly shorten total job execution time. An extensive experimental evaluation of SmartFetch shows that, by avoiding reading irrelevant blocks, it can provide significant speedups when compared to the basic Hadoop and Spark implementations. Further, our system also outperforms other implementations that use several variants of the techniques we have embedded in SmartFetch.**

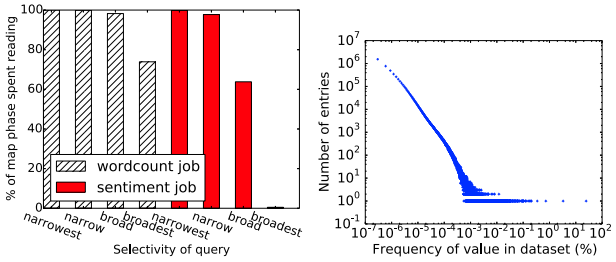
***Keywords*—selective queries, data processing, Hadoop, Spark**

I. Introduction

Today, there is an increasing need to analyse vast 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 computers, to produce results in reasonable time.

The MapReduce paradigm [1], and its variants, have become a fundamental tool to parallelize complex computations over large amounts of data. In this context, MapReduce implementations, such as Hadoop [2], and some of its variants, such as Spark [3] or Flink [4], [5], have become "de facto" standard middleware frameworks which significantly simplify big data processing. Originally, these tools were designed for jobs concerned with the entire dataset, such as web indexing or machine learning. However, as the range of big data applications grows, these platforms are frequently used to execute queries that are only concerned with a small fraction of the entire dataset [6], [7]. 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 [8]. In such applications, data analytics may be performed on demand, for the specific entries that are of interest to each customer.

Unfortunately, current MapReduce implementations, and its variants, are not well tailored to support this type of operation. For instance, in 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 may consume a significant fraction of the entire job. Figure 1(a) shows 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 Twitter dataset.



(a) Cost of reading data. The "broadest"/"narrowest" query filters records by the most common/"narrowest" value in the dataset, respectively.

(b) Distribution of the "user location" values in the Twitter dataset.

Fig. 1: Motivation.

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 knowledge from Twitter datasets [9], [10], [11], [12]. 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 [6], [13], [7], [14], [15], [16], [17]. Systems such as Spark [3] make heavy use of in-memory processing to optimize long iterative pipelines. Still, as it will be seen, Spark can benefit from the proposed techniques every time data needs to be read from disk, such as in the initial phase or when RDDs are materialized.

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 [18]. This is illustrated in Figure 1(b), which presents the distribution of frequencies of the "user location" attribute in a sample of the Twitter dataset used to evaluate SmartFetch. This kind of distribution 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 suggests that, with the appropriate use of indexing, 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 study how to extract data from disk in order to reduce the total job execution time. We propose SmartFetch, a storage strategy that relies on a combination of techniques aimed to efficiently support selective queries/jobs that are only concerned with a subset of the entire dataset.

978-1-4673-9560-1/15 $31.00 © 2015 IEEE
DOI 10.1109/CloudCom.2015.83

We combine the use of an appropriate data-layout with data indexing to significantly shorten the time it takes to fetch data from disk. An extensive experimental evaluation of SmartFetch shows that, by avoiding reading irrelevant data, it can provide speedups up to 200 times when compared to the basic Hadoop implementations. We have also integrated our techniques in Spark and have found that, despite the Spark optimizations, less impressive but still relevant gains can be achieved by SmartFetch. SmartFetch combines these mechanims in a novel way that outperforms previous systems [6], [19].

The rest of the paper is structured as follows. Sections II and III describe the architecture and the algorithms used by SmartFetch, while Section IV captures relevant implementation details. Section V provides an extensive experimental evaluation of SmartFetch. Section VI compares SmartFetch with related work and Section VII concludes the paper.

II. Main SmartFetch Mechanisms

We now present SmartFetch, a storage strategy that embodies a complementary and coherent set of techniques that aim at reducing the time spent fetching data from disk for jobs that only manipulate a fraction of the dataset. SmartFetch combines the following mechanisms: *i) Data-layout:* SmartFetch organizes the dataset in a way that promotes locality. We achieve this by storing table contents by columns instead of by rows. *ii) Data Grouping:* SmartFetch groups similar data at each node, to improve the effectiveness of the indexing mechanism without compromising load-balancing. *iii) Indexing:* SmartFetch creates local indexes, from the data that is stored at each node. Indexes are created and maintained for the most relevant attributes. Although some of these mechanisms have been explored before in different ways, SmartFetch combines them in a novel manner.

A. Data-layout

Input data sets are composed by records. Each record regularly consists of multiple columns, or attributes. One of these attributes, is named the key and identifies each record. The number of records in a data set is typically very large and must be partitioned, into data blocks, across nodes (machines). There are mainly two approaches to map the data set into partitions: row- and 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 [19], 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, the total execution time of the job is substantially reduced. Furthermore, this layout achieves a higher compression ratio since the data domain is the same in each column, hence each column tends to exhibit low information entropy [20]. The better the data is compressed, the shorter the I/O time (reading data from disk is I/O bound; thus, it can be balanced with CPU).

Unfortunately, since each column in a dataset may represent data with different lengths, a naive partitioning in columns may cause column data blocks to become unaligned, complicating the process of building whole records from partitioned datasets. In SmartFetch, we avoid this drawback by first partitioning the dataset horizontally, by creating *row groups* (similarly to [21], [22]), and only then each row group is vertically partitioned by columns (Figure 2). Thus, an attribute is only read when a given query refers to the corresponding attribute, skipping data belonging to other attributes that are irrelevant for the query.

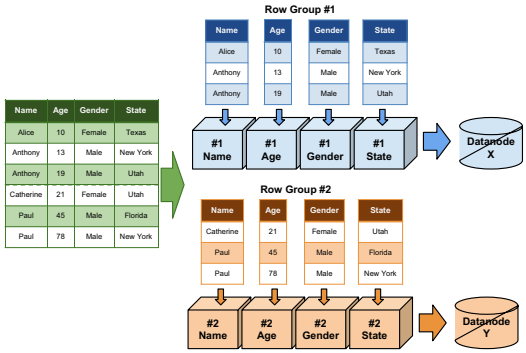


Fig. 2: Row groups in SmartFetch.

Since we use HDFS as our backend and HDFS places data blocks in a random way across all nodes, SmartFetch also includes a row-group aware Block Placement Policy, to make sure that data 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 a job and must be reconstructed from several columns.

B. Grouping

The grouping component of SmartFetch 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, SmartFetch identifies records which share the same value for the indexed attributes, and groups them in a new order such that records with the same value will be contiguous in the dataset. This reordering is done *locally*, for each node in the system. The use of local grouping has two significant advantages with regard to the alternative global sorting approach that has been used in previous systems: not only it makes grouping inexpensive, but also preserves the good load balancing achieved by HDFS's initial distribution of data.

C. Indexing

SmartFetch uses indexing to avoid loading the whole dataset into main memory. This is accomplished by using the

index as an indicator if a given block of data is relevant for the job being processed.

SmartFetch 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 SmartFetch 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 of length equal to the number of columns in the dataset, such that each entry corresponds to 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 SmartFetch 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 persisted to the disk along with the data. These indexes are partitioned in such a way that when a task is scheduled at a node, only the indexes corresponding to the queried attribute value are brought to memory from disk. Additionally, partitioning the indexes in this way enables the task to determine which blocks of its split (partition of the data that the task is responsible for) should be read, if any. In order to achieve efficient index querying, SmartFetch keeps an in-memory cache of indexes of the most recent queried attribute values using a least recently used policy. This design simplifies the management of indexes, since it allows a node to keep the index in memory during the whole job, without requiring information on job completion (which in MapReduce is only available at the JobTracker).

III. Other SmartFetch Aspects

In the previous section we have described the main mechanisms incorporated in SmartFetch. 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 SmartFetch.

Replication: HDFS 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. SmartFetch also allows each data block to be replicated in R different nodes with similar advantages. However, since each node groups the data it stores, replication in SmartFetch must be made on a per-node basis instead on a per-block basis. Furthermore, similarly to other state of the art systems [7], [14], SmartFetch also allows each replica to group the data according to a different attributes. This allows to optimize queries for more than one attribute.

Index Storage: SmartFetch's indexes are stored in disk and loaded as needed during querying. In order to avoid loading unnecessary parts of the index, SmartFetch partitions and saves it into different files, which act as buckets. The partitioning is done based on a prefix of the hashed value of the key (i.e. the value of the attribute).

Pre-Processing: SmartFetch 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 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 mapper as well as reducer. The job goes through all lines in all input files (i.e. all records stored by the node). During the Map phase, mappers output the pair $\langle (ID, value), List\langle field\rangle\rangle$, 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 intermediate key of each pair also contains the value of the attribute by 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 manually selecting which attribute, or attributes, is most relevant for indexing.

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 multiple 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 other state of the art column-oriented storage systems [19], [23], [21]. Throughout this process, SmartFetch captures the starting positions of each different value, such that it can populate the index block.

IV. SmartFetch Implementation

SmartFetch has been implemented as a set of extensions to the Hadoop and Spark frameworks and the prototype is available for download[1]. SmartFetch extensions consist of roughly 6600 lines of code. In this section, we go through the most important tweaks done on Hadoop to support SmartFetch functionality. This includes defining a new `Block Placement Policy` which is aware of row-groups, a new `Input Format` that instantiates a new `Record Reader`, and an optimization that avoids unnecessarily tranferring data across nodes. We opted to run Spark over HDFS in order to leverage the integration work, described in the following paragraphs, when applying SmartFetch to Spark.

[1] https://github.com/manelf91/hadoop-common

First, we need to ensure that files belonging to the same row group are placed on the same node, in order to allow for local record reconstruction when multiple attributes are queried. Since by default, HDFS places blocks randomly across all nodes, SmartFetch 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.

Second, upon job submission, the input is divided into splits, according to the `Input Format` associated with the job being performed. The generated splits are then processed by the map tasks. For SmartFetch, we defined an `Input Format` that is row-group aware, in order to create only one split per row-group instead of one per HDFS block. In addition, the `Input Format` also instantiates the `Record Reader`, which is responsible for generating the key/value pairs from the raw input data and sending them, one by one, to the map function. For SmartFetch, we implemented a `Record Reader` with two particular capabilities. Firstly, the `Record Reader` extracts from the job configuration the attributes referred by the query and passes the values corresponding to those attributes to the map function. Secondly, 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 `Record Reader` obtains the offset of the first row to start reading from that point until reach either the end of the row-group or an entry that no longer satisfies the query.

Finally, the MapReduce model supports the remote retrieval of data blocks. In particular, when a node has finished processing all tasks related with its data, the JobTracker will assign it tasks of nodes which have fallen behind. When processing selective queries, this may cause a block with no relevant information for the query 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 research by others[13]). To avoid this problem, 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 it.

V. Evaluation

In this section, we experimentally answer four research questions: (i) which indexes work best? (ii) what are the benefits of a columnar-oriented data-layout versus a row-oriented data-layout? (iii) what are the benefits of indexing and grouping techniques among columnar-oriented stores? and (iv) what is the impact of different grouping strategies?

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. All values presented are the average of at least 3 executions. Several approaches used in the comparison require some form of pre-processing. Since this pre-processing stage is easily amortized across multiple executions, we have not include it in the comparison.

We used a sample of the Twitter dataset [24], collected between May and September 2012. For most experiments, we have used a sample of this dataset composed of 325,333,833 tweets, that correspond to 988GB of raw data, which when compressed with gzip, have a total of 161GB. For the final experiments, where we measure the speedup provided by incorporating SmartFetch in Hadoop and Spark, we have used a larger sample from the same dataset, containing 610,955,987 tweets corresponding to approximately 2 TB of raw data. 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).

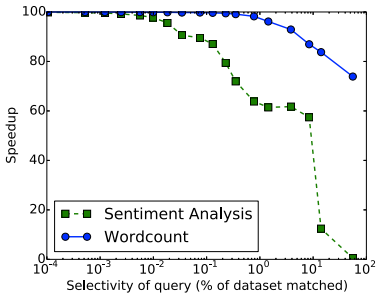


Fig. 3: Cost of reading data, relative to the duration of Map tasks, depending on job type and query selectivity.

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 in, 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 [9], [10], [11], [12], and it has significantly higher CPU processing requirements than the wordcount job.

To better illustrate the difference between both types of jobs, Figure 3 shows the percentage of the Map phase time that SmartFetch spends reading records from disk, when executing the sentiment analysis and the wordcount 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.

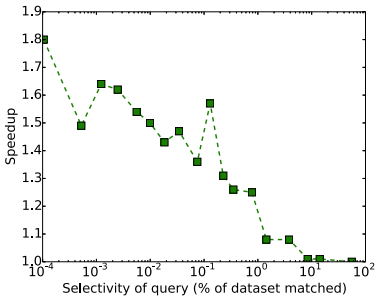


Fig. 4: Read operation speedup of using a list of offsets over using a simple row group identifier for implementing SmartFetch's indexes.

A. Which Indexes Work Best?

In Section II-C we described two possible representations for SmartFetch's indexes: mapping each attribute value to 1) a row-group identifier or 2) a row-group identifier and a list of offsets. These two options differ in performance and space usage. On the one hand, using only a row-group identifier means that the size of the index will not depend on the number of attributes in the system but more data must be read at query time. On the other hand, using a list of offsets may allow for performance improvements since the system can seek directly to the relevant records. To answer this question, we implemented both mechanisms, executed queries of various degrees of selectivity on SmartFetch and recorded the time required to read data from disk.

As Figure 4 shows, using a list of offsets yields considerable improvements when reading data, especially for queries with high selectivity. This can be explained by the fact that for most scenarios with high selectivity, the option of using only a row-group identifier involves always reading a whole block. On the other hand, using an offset allows the system to skip directly to the few relevant records within the block. Although the compression of the dataset limits the possibility of using the "seek" operation of file systems, and thus reducing the efficiency of the list of offsets representation, the experiment shows that modest improvements are achieved.

Regarding the space usage, we have found that for our dataset, the index built using a list of offsets can be take to 16 times more space. However, this comparison is dependant on the number of dataset attributes, as well as on the size and diversity of the indexed attribute. Since the space used by both options takes less than 1% of the size of the dataset, we argue that the space usage loss is shadowed by the performance improvement results. Thus, we decided to use the list of offset index for the remainder of this evaluation.

Independently of the index representation, and discarding global indexes, one could decide to build either a per-block index (similarly to Hadoop++ [6]), or a per-node index, as done by SmartFetch. Hadoop++ appends one index to each block after a pre-processing phase where the block is internally sorted by the values of a chosen attribute. Conversely, SmartFetch creates a single index at each node, and the index maps the values to their corresponding row-group and list of offsets. Table I presents the space used by SmartFetch's two types of index (with and without the list of offsets) and those of a per-block index, depending on the attribute used and on the row group size. The two attributes are representative of two types of distributions which are common in our dataset: "User Language" represents a low cardinality attribute, whereas "User Location" represents a high cardinality attribute.

The results show that for any attribute and row-group size, the best index configuration in terms of space is the per-node index which omits the lists of offsets. This result is due to a) this index having strictly less entries than the block indexes (which may have entries repeated between them); and b) this representation not being dependant on the number of columns in the dataset. The results also show that the index used in all results of this section are comparable to those of a per-block index for a high cardinality attribute, and considerably better for the low cardinality attribute. It is also important to notice that unlike the per-block indexes, the size of the per-node indexes is independent of the size of row-groups.

B. Which Layout Works Best?

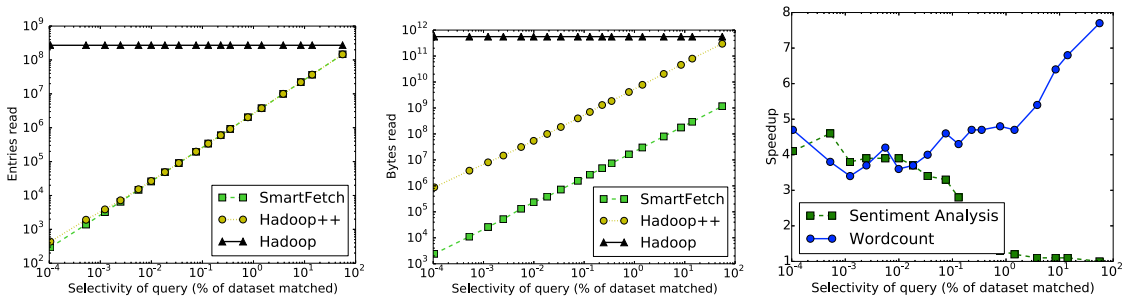
One of the concerns of SmartFetch is the data-layout. To achieve better efficiency when reading data, we use a columnar storage. In order to evaluate the effect of this design decision, in this section we compare SmartFetch with strategies which make use of row-oriented layout for the storage and perform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

A prominent and comparable solution to our system is Hadoop++ [6]. 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 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 [6].

To better illustrate the design differences among Hadoop with SmartFetch, Hadoop++, and unmodified Hadoop, Figures 5(a) and 5(b) 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 SmartFetch (since its indexes allow it to bypass non-relevant blocks). Similarly to SmartFetch, the number of bytes read is dependent on how common the value queried for is. In terms

TABLE I: Space usage of Per-Node Index and Per-Block Index.

Attribute	User Language			User Location		
Row Group Size (MB)	16	32	64	16	32	64
Per-Node Row Group Index (KB)	4.04×10^{-1}	4.04×10^{-1}	4.04×10^{-1}	5.99×10^{5}	5.99×10^{5}	5.99×10^{5}
Per-Node Offset List Index (KB)	6.47×10^{0}	6.47×10^{0}	6.47×10^{0}	3.83×10^{6}	3.83×10^{6}	3.83×10^{6}
Per-Block Index (KB)	3.61×10^{2}	7.00×10^{2}	1.35×10^{3}	2.87×10^{6}	2.70×10^{6}	2.29×10^{6}



(a) Entries read.

(b) Bytes read.

(c) Speedup of SmartFetch over the Hadoop++.

Fig. 5: Comparison between Hadoop, Hadoop++ and SmartFetch

of read data, the main difference between both systems is that since Hadoop++ uses row storage, it must read all columns of records, whereas SmartFetch needs only to read the blocks corresponding to the relevant columns.

Figure 5(c) shows how these decisions reflect in terms of performance of the system. This figure presents the speedup provided by using SmartFetch in Hadoop over Hadoop++ while querying for different values. As expected from the analysis of read data, for all executions, SmartFetch exhibits speedups over Hadoop++. Similarly to what has been observed when comparing Hadoop with SmartFetch vs unmodified 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 most selective jobs, SmartFetch shows up to 5 times speedups since it has to read less data.

C. How to Mix Indexing and Grouping?

Similarly to SmartFetch, other systems have also adopted a columnar storage. In this section, we study the impact of the indexing and data grouping techniques by comparing our system with CIF [19], which also uses a columnar-oriented data-layout. Since CIF's code is not publicly available, again we have implemented a prototype of CIF using the codebase of SmartFetch, by disabling the indexing and data grouping components of our system.

Figures 6(a) and 6(b) compare SmartFetch with Hadoop and CIF in terms of entries and bytes read. Since CIF lacks any kind of indexing, the number of bytes and entries read does not vary as a function of how common the queried value is, similarly to unmodified Hadoop. However, the amount of data read by CIF is smaller than that of Hadoop, since it can read only the columns relevant for the query at hand.

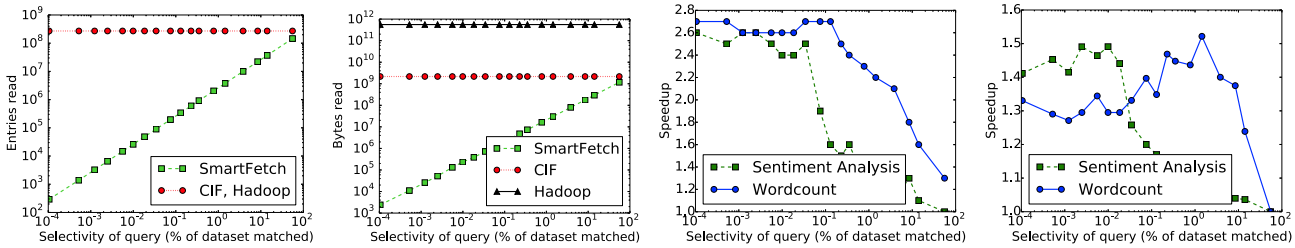
Since SmartFetch uses indexing and data grouping to reduce the amount of data read, it outperforms CIF for the scenarios where the indexes are more effective. Figure 6(c) depicts the speedup achieved by SmartFetch over CIF, for both of the considered jobs. For less selective queries, the speedup is more reduced, while when querying for more uncommon values, the indexes allow SmartFetch to provide a speedup of up to 2.6 over CIF. For the job with a higher processing cost the average speedup is smaller (about 1.9), while for the lighter job a higher average speedup is achieved (around 2.4).

D. What is the Impact of Different Grouping Strategies?

Several state of the art systems opt by using column layouts along with per-block indexes [14], [7]. 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 SmartFetch), and indexes each row group individually (unlike SmartFetch, which groups data and indexes data on a per-node basis). The main goal of the study presented in this section is to evaluate the effect of the grouping and the per-node index component of SmartFetch.

Figure 6(d) presents the speedup provided by SmartFetch over an indexed column-oriented store. As was observed before, since the less selective queries of the sentiment analysis job are heavily CPU-bound, SmartFetch does not present a significant advantage over other solutions. Still, for this job SmartFetch 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 SmartFetch which only loads a partial index on the first task associated with the job. Secondly, and more importantly, since data is not grouped, several blocks may match the query value. This would require the system to open several blocks and seek to the offset of the value, whereas SmartFetch 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 SmartFetch over an indexed column-oriented store actually increases when the selectivity of the query decreases. These results are strongly tied with the number of blocks matched by the query. For SmartFetch, the number of blocks matched is proportional to the selectivity of the query. However, on a system that does not rely on grouping, the number of blocks matched 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 SmartFetch remains very small; whereas for the system using per-block indexes, all blocks match the queries. This also explains why towards the most common attributes, the speedup of SmartFetch drops to values close to 1: in this case, in both systems, a large number of blocks match and the cost



(a) Entries read by Hadoop, CIF and SmartFetch.

(b) Bytes read by Hadoop, CIF and SmartFetch.

(c) Speedup of SmartFetch vs CIF.

(d) Speedup of SmartFetch vs an indexed column-oriented store.

Fig. 6: Speedup of SmartFetch vs column-oriented stores.

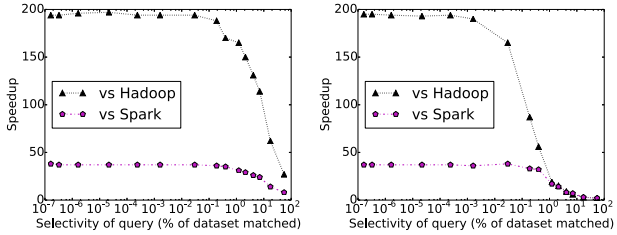
of reading the data shadows that of seeking inside the blocks. Furthermore, also notice that for the rarest item, the speedup of SmartFetch is also more reduced than for the following queries. This is due to this value being present only on a single block for both systems, yielding a similar performance (with a slight edge to SmartFetch due to reading only a single index).

E. Speedup provided by SmartFetch in Hadoop and Spark

Finally, for completeness, we evaluate the speedup provided by SmartFetch when integrated with Hadoop and Spark[3]. For this evaluation, we used a considerably larger dataset containing 610,955,987 tweets corresponding to approximately 2 TB of raw data. When running Spark, data was also stored using the Hadoop DFS. Figure 7 shows the speedup that SmartFetch achieves over unmodified versions of Hadoop and Spark for the wordcount and sentiment analysis jobs, respectively. SmartFetch achieves speedups from 8 (for the most common value) up to 38 times (for the rarest items), for the wordcount job and between 2 and 37 times for the sentiment analysis job. The speedup is lower when performing jobs with higher data processing rates. These speedups are achieved because, like to Hadoop, Spark must read the whole dataset from disk at least one time, in order to perform a selective query. This shows that even though Spark can considerably outperform Hadoop for most workloads, it can still benefit from using SmartFetch in selective scenarios.

VI. Related Work

Several state of the art systems take advantage of different data layouts. The three main approaches in the literature are row-oriented layouts [6], [25], [26], column-oriented layouts [23], [19], [22] and the PAX format [14], [21], [27]. Row-oriented layouts are not well suited for supporting selective queries since they have a higher overhead when retrieving partial records (as shown in Section V-B). Column-oriented layouts, similarly to PAX, support partial reads of the dataset. 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. SmartFetch 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. Modifying SmartFetch 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. A system that combines in certain way row- and column-oriented approaches is Llama [28] in a new file format called *CFile*. The idea is to partition the data by columns also with the goal of optimizing selective queries and at the same time, avoiding the penalty of recomposing rows. Unfortunately, in their approach, this penalty may still be incurred if the set of attributes requested by a query does not match any vertical group. SmartFetch tackles this problem in a better way, by defining a `Row-Group Aware Block Placement Policy` as explained in Section IV. A similar work is Cheetah [27]. The authors propose to partition the data into cells, containing n rows each of them. Inside the cell, the rows are partitioned per column and then compressed with gzip. Since the cell is the unit of storage, parts of a row are never scattered across nodes, allowing Cheetah to avoid expensive network access. However, in contrast to SmartFetch, Cheetah do not use neither grouping nor indexing techniques.



(a) Word Count.

(b) Sentiment Analysis.

Fig. 7: Speedup of SmartFetch vs Spark and Hadoop.

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++ [6] proposes a Trojan Index, which is created on a per-block basis and appended to the block. Thus, reading the index before loading the block from disk permits Hadoop+ to reduce the number of records retrieved from disk. HAIL [7] and LIAH [14] 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 limits the performance of the system (as shown in Section V-D). Similarly to our system, the work by Lin et al. [26]

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-D, indexing tends to be an ineffective technique when not combined with block rewriting.

Differently to previous solutions, HadoopDB [29] combines techniques from Hadoop and DBMSs. The authors propose to use the distribution framework from Hadoop to connect single-node DBMSs running on each machine. In consequence, similar results to SmartFetch could theoretically be achieved, by using a column-oriented DBMS, such as C-Store [30]. Nevertheless, it was shown that HadoopDB underperforms with regard to Hadoop++ (and we outperform Hadoop++).

VII. Conclusions

In this paper we have described the design, implementation, and experimental evaluation of SmartFetch, a storage strategy that significantly improves the performance of jobs that are concerned with just a subset of the entire dataset in systems such as MapReduce and Spark. Our experimental results, obtained with a sample of a real dataset, show that SmartFetch can provide speedups up to 40 to 200 times when compared to the basic Spark and Hadoop implementations, respectively. Naturally, better results are obtained for queries that target uncommon values, but our results show that SmartFetch 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.

Several of the ingredients used by SmartFetch have been presented before in different forms. We have provided an extensive experimental comparison with competing implementation that have used those ingredients and have shown that our novel way of combining these techniques outperforms those competing solutions. SmartFetch has been implemented as open source and has been made available for others to experiment and to improve upon.

Acknowledgments: The authors would like to thank A. Francisco for providing the Twitter dataset. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) under projects PEPITA (PTDC/EEI-SCR/2776/2012) and UID/CEC/50021/2013.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI '04*, San Francisco, CA, 2004.

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

[3] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *HotCloud '10*, Boston, MA, 2010, pp. 10–10.

[4] Apache Flink. [Online]. Available: https://flink.apache.org

[5] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec. 2014.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: Locality-aware resource allocation for mapreduce in a cloud," in *SC '11*, 2011.

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

[15] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *CCGRID '10*, Washington, DC, USA, 2010.

[16] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *OSDI '08*, San Diego, California, 2008, pp. 29–42.

[17] 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.

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

[19] A. Floratou, J. Patel, E. Shekita, and S. Tata, "Column-oriented storage techniques for mapreduce," *Proc. VLDB Endow.*, vol. 4, no. 7, pp. 419–429, Apr. 2011.

[20] D. Abadi, S. Madden, and N. Hachem, "Column-stores vs. row-stores: How different are they really?" in *SIGMOD '08*, Vancouver, Canada, 2008, pp. 967–980.

[21] 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 *ICDE '11*, Washington, DC, USA, 2011.

[22] Apache Parquet. [Online]. Available: https://parquet.incubator.apache.org

[23] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich, "Trojan data layouts: Right shoes for a running elephant," in *SOCC '11*, Cascais, Portugal, 2011, pp. 1–14.

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

[25] M. Eltabakh, F. Özcan, Y. Sismanis, P. Haas, H. Pirahesh, and J. Vondrak, "Eagle-eyed elephant: Split-oriented indexing in hadoop," in *EDBT '13*, Genoa, Italy, pp. 89–100.

[26] J. Lin, D. Ryaboy, and K. Weil, "Full-text indexing for optimizing selection operations in large-scale data analytics," in *MapReduce '11*, San Jose, California, USA, pp. 59–66.

[27] S. Chen, "Cheetah: A high performance, custom data warehouse on top of mapreduce," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1459–1468, Sep. 2010.

[28] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu, "Llama: Leveraging columnar storage for scalable join processing in the mapreduce framework," in *SIGMOD '11*. New York, NY, USA: ACM, 2011, pp. 961–972.

[29] A. Abouzied, K. Bajda-Pawlikowski, J. Huang, D. Abadi, and A. Silberschatz, "Hadoopdb in action: Building real world applications," in *SIGMOD '10*. New York, NY, USA: ACM, 2010, pp. 1111–1114.

[30] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented dbms," in *VLDB '05*. VLDB Endowment, 2005, pp. 553–564.