Tuning Small Analytics on Big Data: Data Partitioning and Secondary Indexes in the Hadoop Ecosystem

Abstract

In the recent years the problems of using generic storage (i.e., relational) techniques for very speci c applications have been detected and outlined and, as consequence, some alternatives to Relational DBMSs (e.g., HBase) have bloomed. Most of these alternatives sit on the cloud and bene t from cloud computing, which is nowadays a reality that helps to save money by eliminating the hardware as well as software xed costs and just pay per use. On top of this, speci c querying frameworks to exploit the brute force in the cloud (e.g., MapReduce) have also been devised. The question arising next tries to clear out if this (rather naive) exploitation of the cloud is an alternative to tuning DBMSs or it still makes sense to consider other options when retrieving data from these settings.

In this paper, we study the feasibility of solving OLAP queries with Hadoop (the Apache project implementing MapReduce) while bene ting from secondary indexes and partitioning in HBase. Our main contribution is the comparison of di erent access plans and the de nition of criteria (i.e., cost estimation) to choose among them in terms of consumed resources (namely, CPU, bandwidth and I/O).

Keywords: Big Data, OLAP, Multidimensional Model, Indexes, Partitioning, Cost Estimation

1. Introduction

The relevance of informed decision making has al-ready shifted the focus from transactional to decisional databases. Nowadays, it is out of question that decision making must be supported by means of objective evidences inferred from digital traces gathered from the day-by-day activity of the organizations. Up to date, data warehous-ing has been the most popular architectural setting for de-cisional systems and it is nowadays a mature and reliable technology stack present in many big companies / organi-zations and already making its way on SMEs. However, we are currently witnessing a second paradigm shift due to the success of data warehousing: the need to incorpo-rate external data to the data warehouse. In short, many works have discussed the relevance of the context in nowa-days decision making that cannot be just focused on sta-tionary data (i.e., that owned by the decision maker) and must deal with situational data (i.e., any non-stationary data relevant for decision making) as rst-class citizen [1]. This new paradigm shift has given raise to the so called Business Intelligence 2.0 and is inevitably coupled with the concept of Big Data.

Although Big Data has been around for a while and has modi ed the agenda of many research communities, its de nition is still far from being agreed and it usually refers to decisional systems characterized by the 3 V's: vol-ume (large data sets), variety (heterogeneous sources) and velocity (referring to processing and response time)[1]. As discussed in [2], Big Data analytics can either mean Small or Big analytics. Small analytics focus on providing ba-sic query capabilities (typically related to SQL aggregates such as count, sum, max, min and avg) on very large data sets, whereas Big analytics entails the use of computation-ally expensive and more advanced algorithms implement-ing data mining and machine learning techniques. This is re ected in [3], where the 43:3% of the workload used corresponds to the former and the 56:7% to the latter.

Indeed, Small and Big analytics naturally map to tra-ditional data warehousing analytics. Typically, OLAP [4] has been rstly used to gain quick insight into the data and spot interesting data sets in a rst step to, later, in a second stage and by means of Data Mining / Machine Learning, identify and foresee trends in such data sets. In this paper, we focus on the former and use OLAP and the multidimensional model [5] to analyze the performance of Small Analytics on Big Data.

The multidimensional model (MD) represents data as if placed in an n-dimensional space (i.e., the data cube, which allows to compute the most usual Small Analytics - i.e., sum, count, avg, max, min, etc.), and facilitates the understanding and analysis of data in terms of facts (the subjects of analysis) and dimensions forming the mul-

[1]

tidimensional space where to place the factual data. A dimension is formed by a concept hierarchy representing di erent granularities (or levels of detail) for studying the fact data or measures. A fact and a set of dimensions form a star schema (usually implemented following a star-join relational pattern). Nowadays, the MD model is not only the de facto standard for data warehousing modeling and OLAP but it is also increasingly gaining relevance for data mining mainly because of its powerful foundations for data aggregation. More speci cally, the MD model in-troduces the Roll-up operator [4], which enables dynamic aggregation (i.e., group by) on measures along dimension hierarchies.

In this paper we explore how to perform Small Ana-lytics in Hadoop by means of OLAP queries and analyze the performance of di erent approaches while at the same time diving into the HDFS technical details to explain the results.

Related Work. Querying star-join schemas in a user-friendly manner is one of the main claims of OLAP. This is still badly needed for data scientists querying Big Data [6]. For this matter, a high-level declarative language abstract-ing the user from technical and implementation details is a must. However, this is no longer true for MapReduce and the Hadoop ecosystem [7], the most popular archi-tectural setting for Big Data. MapReduce requires user-created code to be injected in a Java framework (i.e., the map and reduce functions). These functions are seen as a blackbox by the Hadoop ecosystem, which does not im-plement any relational-like query optimizer. Thus, query answering is

purely based on the brute force of the cloud. Some e orts, such as Hive[2], have introduced a declara-tive SQL-like language to automatically create MapReduce jobs. Hive translates each high-level SQL-like operator into MapReduce job(s), which are then sequentially sched-uled to consume the output (to be persisted in HDFS) of the previous MapReduce job. Such approach incurred in a high latency and the execution of several redundant tasks. Consequently, the Stinger initiative[3] focused on im-proving these execution plans by means of rewriting and pruning rules. Pig![4] introduces a high-level ETL-like lan-guage called Pig Latin. Pig Latin statements are then automatically translated into MapReduce jobs. Like Hive, Pig! addresses the optimization of its execution plans by de ning some optimization rules and hints. The optimiza-tion solutions presented by Hive and Pig! resemble those of early RDBMS based on rule-based optimization rather than current cost-based solutions [8]. To the best of our knowledge, the only cost-based optimization attempt in the Hadoop ecosystem is the Optiq project[5]. However, Optiq was only recently accepted in the Apache Incubator community and it is still in a very preliminary status [9].

2http://hive.apache.org/
3http://hortonworks.com/labs/stinger/
4http://pig.apache.org/
5http://incubator.apache.org/projects/optiq.html

Indeed, up to now, most e orts have focused on tuning and further develop the Hadoop framework internals (e.g., [10, 11, 12, 13]) rather than apply traditional database tuning, to which little attention has been paid and forms the main scope of this paper.

Contributions. In this paper, we consider the con-vergence of the most popular setting for Big Data (the Hadoop ecosystem) and the MD model to activate Small analytics on large data sets. Since we assume a Hadoop en-vironment, it is unfeasible to expect a well-formed star-join schema in terms of fact and dimension tables. For this rea-son, we assume a fully-denormalized fact table approach (i.e., measures and dimension attributes are denormalized in a single table). Our contributions are as follows:

Inspired by a traditional data warehousing setting, we study two database design techniques that have shown a big impact on data warehouses:

- { partitioning (either horizontal or vertical) the fact table and
- { the e ective use of secondary indexes on dimen-sional data to solve the selection predicates of the queries.

Next, we study how to map these design techniques on a database sitting on a HBase cluster and study their impact by means of exhaustive empirical tests.

Finally, we have characterized our ndings in terms of cost formulas for each of the MapReduce al-gorithms to compute multidimensional data cubes, which represent the seed of a query optimizer for OLAP querying on Hadoop.

Relevantly, the use of a well-known technology such as the MD model for computing Small Analytics on Big Data will enable further and advanced navigation capabilities by implementing a multidimensional algebra on top of the two algorithms here presented and considering the best execution plan according to our cost formulas.

The paper is organized as follows. Section 2 intro-duces the Hadoop ecosystem and its main features. Sec-tion 3 discusses how to build cubes on Hadoop by means of two algorithms: the IRA (Index Random Access) and FSS (Full Source Scan) algorithms. These algorithms ben-e t from two main tuning features: partitioning and sec-ondary indexes and we also discuss how to implement them in Hadoop. As subsequently discussed, partitioning is na-tively supported in HBase but secondary indexes must be simulated. At this point, we introduce the IRA and FSS al-gorithms in detail. Next, Section 4 characterizes the most relevant cost factors in order to estimate each algorithm cost, which then it presents them in terms of cost formu-las. Section 5 presents the experimental setting backing up our ndings described in previous sections, which is nally discussed in Section 6. This section also presents several potential enhancements for HBase, including the new IFS algorithm (Index Filtered Scan) explained in Section 7.

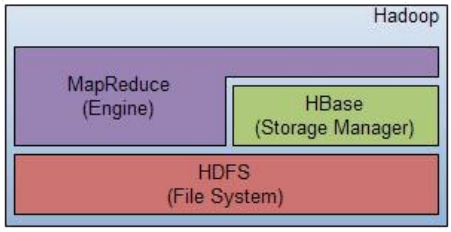

Figure 1: Logical architecture

Technology	Master	Slave
HDFS	NameNode	DataNode
HBase	HMaster	RegionServer
MapReduce	JobTracker	TaskTracker

Table 1: Node names for every technology

2. Hadoop Environment

As de ned in [7], \the Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from sin-gle servers to thousands of machines, each o ering local computation and storage."

The Hadoop ecosystem used in this paper is imple-mented as a three level architecture in which we nd HDFS (the le system) running at the lowest level, HBase (the storage manager) running on top of HDFS and nally MapReduce (the query execution engine) wrapping them so that data processing can be performed at both the le system and the database level (Figure 1 shows this logical architecture). All these technologies follow a master-slave architecture. The master node is responsible for tracking the available state of the cluster and it basically coordi-nates the slave nodes, which are those doing the actual work (Table 1 shows the di erent nomenclature used for each technology).

As consequence, these technologies are relatively inde-pendent from each other in the sense that they do not form a single process running in a machine, but there is one independent process for each of them interacting to each other through the network. In a traditional setting a tuple is retrieved by querying the RDBMS, which for-wards the message to the le system, which, in turn, re-trieves the corresponding disk block and sends it back to the DBMS. Typically, these communication costs are disre-garded since there is a strong coupling between the DBMS and the le system and such communication is performed in main memory. However, this is no longer true in an ar-chitecture like Hadoop since the le system (HDFS), the storage manager (HBase) and the query engine (MapRe-duce) do not form a single unit and their communication is implemented via much more expensive network commu-nication.

2.1. HDFS

As de ned in [14], \the Hadoop Distributed File Sys-tem (HDFS) is a distributed le system designed to run on commodity hardware. It has many similarities with ex-isting distributed le systems. However, the di erences from other distributed le systems are signi cant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets."

This fault-tolerance and high throughput access re-quirements are achieved by means of balancing and repli-cation, which are the two strongest points of HDFS. When a new le is to be written in the le system, it is rst split into blocks of a given size (64MB by default, but con g-urable). Afterwards, (i) each block is stored in a DataN-ode (i.e., balancing) and (ii) it is replicated in di erent nodes (i.e., replication). Balancing allows HDFS to have a great performance working with large data sets, since any read/write operation exploits the parallelism of the cloud. Replication implies mainly high availability, since di erent replicas can be used in case any of them becomes tempo-rally unavailable, but it may also boost performance by choosing the closest replica and reducing communication costs. When it comes to synchronizing replicas, HDFS applies an eager/primary-copy strategy. Thus, writing can only happen on the primary-copy and its replicas are blocked until they are synchronized.

Note that HDFS also follows a master-slave architec-ture as stated in Table 1. Thus, DataNodes are those storing the data, while the control ow responsibility is taken by the master node NameNode.

2.2. HBase

\Apache HBase is an open-source, distributed, ver-sioned, non-relational database modeled after Google's Bigtable: A Distributed Storage System for Structured Data by Chang et al. Just as Bigtable leverages the dis-tributed data storage provided by the Google File System, Apache HBase provides Bigtable-like capabilities on top of Hadoop and HDFS " (see [15]).

Data are stored in HBase by following [key,value] struc-tures. In such pairs, the key represents the row identi er and the value contains the row attributes. The [key,value] pairs are stored using the equivalent to well-known pri-mary indexes for RDBMS, which physically sort rows on disk and build a B+ tree on top of it (see [8]). In HBase, this sorting is done on the key of the pair.

HBase also performs horizontal partitioning [16] based on the keys. Such partitions are called \regions", which are the minimal balancing unit used by HBase. Data distribu-tion is done according to the number of regions per node (i.e., RegionServers in HBase). Tuples are distributed de-pending on the region they belong to, but, in principle, regions are not guaranteed to be of the same size and hence data is not completely evenly distributed across the

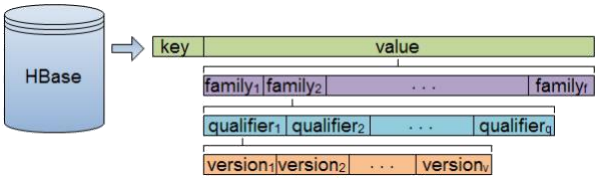

Figure 2: Internal structure of an HBase row

cluster. Additional features such as region splits and com-pactions (see [17]) were introduced to eventually achieve, in the presence of large enough data volumes, an even dis-tribution among RegionServers.

Moreover, HBase further structures the value to sup-port vertical partitioning [16]. Figure 2 sketches how a table row is stored in terms of families and quali ers. Fam-ilies must be explicitly created by modifying the schema of the table. However, one quali er belongs to a family and it is only declared at insertion time. Thus, providing enough exibility as expected in a schemaless database. Then, for each family and quali er, there are versions (timestamps). Each combination of a family, quali er and version deter-mines an attribute value for a given key. For instance, a table could have the family \building", and this fam-ily could have \price" and \surface" as quali ers (i.e., di erent attributes). Versioning keeps track of the n (con-gurable) most recent values of these attributes.

HBase physically stores each family in a di erent le and thus, natively supports vertical partitioning. Vertical partitioning is relevant for read-only workloads since it improves the system performance, because non-relevant families (for the current query) are not read [16]. Note that quali ers play a key role to decide which attributes must be stored together on disk by placing them in the same family.

Data belonging to the same region must be stored in the same DataNode in HDFS (in order to avoid degrad-ing performance). Otherwise, data would be unnecessarily spread all over the cluster regardless of vertical and hor-izontal partitioning strategies applied. Accordingly, there must be some communication between HDFS and HBase so data are stored where they are managed (data locality principle). This implies that a RegionServer must always run on top of one DataNode. Figure 3 presents a UML diagram depicting how HDFS and HBase are coupled.

As shown in this gure, HBase tables are horizontally partitioned in regions that, in turn, are vertically parti-tioned (according to families) in stores. There is exactly one store per region and family. Data are physically stored in stores. First, in in-memory bu ers (memstores), which are then ushed to disk as store les. Store les are rep-resented as HFiles (having speci c metadata), which are divided into HBase blocks. Finally, these store les need to be written in HDFS, so they are chunked into HDFS blocks (note that in Hadoop they refer to HDFS blocks as synonym of HDFS chunks, which would be more appropri-ate, since they are not physical disk blocks) and replicated across di erent DataNodes. Note that this is a logical schema and thus, the physical settings in terms of which HDFS blocks are stored are not depicted here (indeed, they depend on the cluster con guration). In this paper, we will normally talk about HBase blocks and therefore, when referring to a \block", it must be read as a HBase block unless the opposite is explicitly said.

In order to guarantee the data locality principle (i.e., a DataNode stores the HDFS blocks of the store les it holds as RegionServer), the control ow between HBase and HDFS is as follows. When a RegionServer writes on disk it asks to its DFS client to open a writer stream. As the RegionServer writes, the DFS client packages these data until it reaches the maximum HDFS block size. At this point, the DFS client communicates to the NameNode the need to materialize such block and it is the latter who decides where to place the master copy of such block (as well as its replicas). The NameNode applies an internal policy to do so (see [18]) that rstly checks if there is a DataNode running on the same node as the DFS client who asked for writing the block. If so, the local DataNode stores the master copy.

Relevantly, HBase implements a cache to store recently read blocks. This way, HBase may save reading a block from disk if recently read and still cached. Last, note that HBase tuples can only be accessed using the HBase scan object, which retrieves tuples by means of the distributed B+ index and thus, e ciently supports retrieving a sin-gle key or a range of (consecutive) keys (i.e., typical B+ accesses).

Finally, Zookeper [19] is \a centralized service for main-taining con guration information, naming, providing dis-tributed synchronization, and providing group services.". In HBase it is basically used to keep track of the dis-tributed B+ index. ZooKeeper points at the B+ root table (-ROOT-) and whenever a tuple must be retrieved from HBase it nds out where to look for the tuple by exploring the B+. The HBase B+ has three levels and it is stored as a regular HBase table. At the rst level there is the B+ root. The next level corresponds to the regions of the catalog table (.META.), which points to RegionServers. Finally, the third level contains the region where these data logically belong to.

2.3. MapReduce

As stated in [20], \Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hard-ware in a reliable, fault-tolerant manner. A MapRe-duce job usually splits the input data-set into independent chunks which are processed by the map tasks in a com-pletely parallel manner. The framework sorts the outputs of the maps, which are then input to the reduce tasks. Typ-ically both the input and the output of the job are stored in a le-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks."

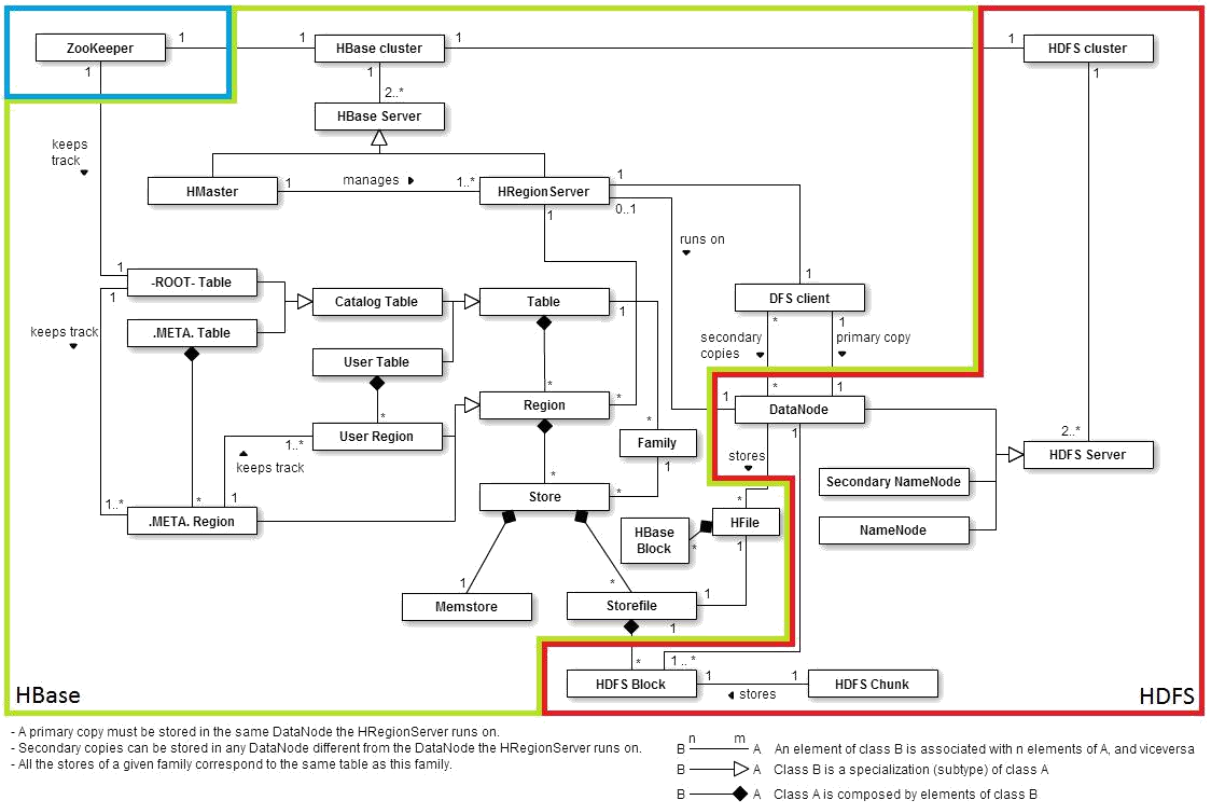

Figure 3: HBase and HDFS

MapReduce is a programming framework. The programmer must de ne the task input and output, and implement the map and reduce functions. Then, paralleliza-tion is transparent.

Figure 4 sketches an easy example of a MapReduce execution for aggregating data. The map and the reduce functions must be provided and this is where the program-mer injects his / her code. In this example, only those rows of interest (i.e., rows from \EUROPE" or \AFRICA") are sent to the map functions. Then, the map rearranges the [key,value] pairs received and produces new [key,value]s useful for the aggregation. Afterwards, the Merge-Sort process gathers all these [key K,value V] produced and groups the values V corresponding to the same key K in a new [key K,value L], where L is a list containing all these values V . Finally, the reduce function receives these key-value pairs and iterates over L to properly aggregate the data. Note however the di erence between a mapper and a map (respectively, a reducer and a reduce). A mapper is the class distributed (i.e., the query shipped) and the map is the instance function processing input elements. When HBase serves as input for MapReduce, there is exactly one mapper for each region and each mapper executes one map function for each row in the region (in the default setting). Note that the row is properly joined back from the di er-ent family les prior to be sent to MapReduce. The same applies to reducers and reduces, but in this case a reducer does not depend on how the input is split but on the task con guration (where the number of suggested reducers is stated). In this paper we use the default input and output split con guration in order to focus on tuning design is-sues rather than parameters of the framework, which other works thoroughly studied (e.g., [10, 11, 12, 13]). Tempo-ral results produced by MapReduce are stored in HDFS (e.g., the Merge-Sort output). Thus, MapReduce just uses HBase for reading / storing the input / output. Therefore, for intermediate steps the HDFS con guration applies.

In our experiments, we have used version 1.0.4 for Hadoop (HDFS and MapReduce) and 0.94.4 for HBase.

3. Building Cubes

In this section, we present two algorithms used to re-trieve cubes from Hadoop, which correspond to the typ-ical options relational optimizers take into account when accessing a table, namely \Index Random Access" (IRA) and \Full Source Scan" (FSS). In our approach, data is stored in HBase and the algorithms are implemented as MapReduce jobs. As discussed in Section 1, we assume a fully denormalized fact table containing all data related to the subject of analysis. This solution incurs in extra space, but it avoids joins. In addition, it allows storing the snapshot of the dimensional data at the time the fact occurred (i.e., facilitating the tracking of slowly changing dimensions). The two algorithms implemented are as fol-lows:

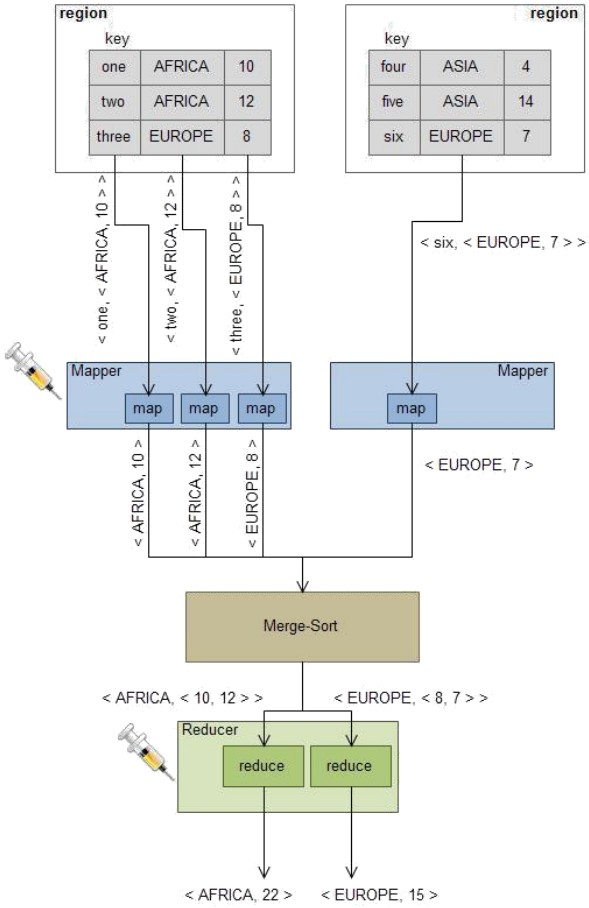

Figure 4: MapReduce execution example

IRA: This algorithm uses prede ned indexes to solve selection predicates in the query and obtains the identi ers of the needed tuples meeting such predicates. Finally, it retrieves the necessary fact table data through random accesses. Thus, IRA mirrors the typical access plan used with primary indexes [8].

FSS: This algorithm is the baseline to check whether using secondary indexes on HBase makes sense or not. Essentially, it scans the whole fact table and lters it by exploiting the parallelism provided by the cloud.

These algorithms were theoretically presented in [21] and now have been adapted for large distributed scenarios. Prior to introduce these two algorithms in detail we rst elaborate on the two tuning features we aim at exploiting when implementing the IRA and FSS algorithms: data partitioning and secondary indexes.

3.1. Tuning Features

In this section, we go through the details of each tuning feature previously mentioned. Firstly, we discuss how both horizontal and vertical partitioning are achieved in HBase in order to nally identify what factors are playing a key role in this matter so they are taken into account when tuning. Secondly, we simulate secondary indexes in HBase.

3.1.1. Data Partitioning

As discussed in Section 2, HBase horizontal partitioning distributes data across regions. When reading from HBase, MapReduce splits the input data addressing each region to a di erent mapper and thus, the number of regions (i.e., horizontal partitioning) directly a ects the degree of parallelism of MapReduce tasks.

HBase allows DBAs to manually partition the relations instead of using an automatic policy. This resembles the situation for distributed RDBMS where data distribution is done at design time. However, the Hadoop ecosystem is thought to provide highly scalable settings and thus a static/prede ned partitioning would not always be the best choice. For this reason, HBase can be con gured to use di erent policies for dynamic/automatic partitioning and even provides tools to let DBAs implement their own. For the sake of simplicity, we will focus on the default policy. This systematically checks if there is a store le larger than a given threshold. If so, a new region split is triggered and a new partition (i.e., region) is created. Importantly, if a store le is split, all store les (i.e., fam-ily les) belonging to the same region will also split (even if they did not reach the set threshold) to preserve data locality. Formula 1 shows how this threshold size is set.

$$split:threshold = min(R^2 mem:size; max:size) \quad (1)$$

The splitting threshold is de ned as the minimum of (i) a function of the number of regions in the corresponding RegionServer (R) and the maximum size of the memstore (mem:size), and (ii) a constant value max:size. The rationale behind such formula is to use max:size as splitting factor in the long term. However, purely using a con-stant may lead to low performance in many cases. On the one hand, a large value would generate few parti-tions, and therefore very large amounts of data would be needed to exploit the parallelism of the cloud. On the other hand, a small value would lead to too many parti-tions that would impact on the nal execution cost due to the startup time of too many parallel tasks. Since set-ting the right max:size would not be easy, this formula is thought to deal with this trade-o . Thus, at the begin-ning, the rst element is used and data split at a faster pace (regardless of max:size). Eventually, that value will increase until surpassing max:size after a certain amount of splits have taken place. Only from then on, the splitting step will remain constant.

Accordingly, the partitioning strategy tested in this pa-per depends on a combination of the following factors: (i) the number of RegionServers, and (ii) the vertical parti-tioning and compression strategies that will impact on the growth pace of the store les. We kept the memstore size mem:size to its default value (i.e., 128 MB). Examples of this are as follows:

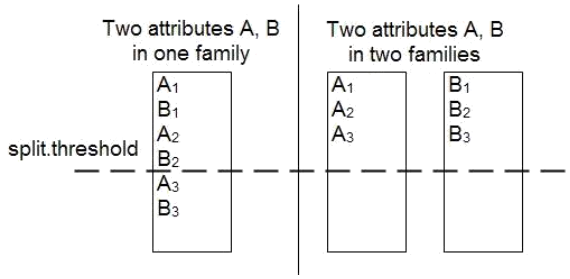

Figure 5: E ect of vertical partitioning on region splits

Let's assume a situation with ve regions (i.e., partitions) in total:

{ If the number of RegionServers is ve and the regions are evenly distributed (i.e., every RegionServer stores one region), then R = 1 and according to the previous formula, the next region split will occur when any of the store les reaches:

$$split:threshold = 1^2 mem:size$$

{ If the number of RegionServers is one, and therefore all the regions are stored in the same RegionServer, then:

$$split:threshold = 5^2 mem:size$$

Thus, the more RegionServers we have, the more regions (i.e., partitions) are created.

Each store le contains exactly one family and consequently the number of vertical partitions (i.e., fami-lies) determines the number of store les. Thus, the larger the number of families the harder for a store-le to reach the splitting threshold, since, with less partitions, each family will contain more attributes and therefore it is faster for any of them to reach the splitting threshold (Figure 5 shows this graphically). Note that this implies that horizontal partitioning pace depends on the vertical partitioning design.

Compression has a similar e ect on the store le size. A strong compression makes the store les to use less space, so it takes more data to reach the splitting threshold. This e ect is the other way round with lower compression (or no compression at all).

Summing up, since max:size and mem:size are two constant values, the number of RegionServers set the split threshold, whereas the vertical partitioning and the compression algorithm used (if any) determine how fast the split threshold is reached (e.g., with no compression and one family the split threshold will be reached faster than with ten families and a heavy compression algorithm).

An important issue the reader may note about this policy is that it does not guarantee an even distribution of data. More precisely, such even data distribution can only be assumed to take place eventually, when the constant value max:size is used as main splitting factor (bear in mind that distribution in HBase is performed based on the number of regions each RegionServer holds as pointed out in Section 2.2). Consequently, HBase does not take into ac-count the amount of data each RegionServer contains when distributing, but the number of regions. Furthermore, the rst argument of Function 1, which is the main splitting factor in the short time, is quadratic and may lead to sensible di erences in the data distribution between nodes. The poor performance of HBase and MapReduce when distributing data has already been highlighted in previous works (e.g., see [22]).

Section 5.3 further elaborates on the distribution of data in HBase in our experimental settings.

Key (hierarchy member)	Value (list of fact keys)
Region%Europe%France%Lyon	key_a, key_c
Region%Europe%Italy%Milan	key_d, key_e
Region%Africa%Kenya%Nairobi	key_b, key_f , key_g
Gender%Male	key_c, key_d, key_f
Gender%Female	key_a, key_b, key_e, key_g

Table 2: Snapshot of a secondary index

3.1.2. Secondary Indexes

Although analytical queries usually perform aggregations over non-very-selective rows, they exhibit selective predicates rather often. Accordingly, we aim at exploit-ing indexing techniques to avoid full scans of fact tables. However, note that HBase only provides a distributed B+ on the keys and no further support for customized indexes is provided. Therefore, we assume a traditional approach for indexing where indexes are built before querying data, since they can be reused to answer disparate queries if incrementally maintained to re ect the subsequent updates.

Setting-Up. In our approach, secondary indexes are implemented as HBase tables containing [key K,value V] pairs such that the key K refers to a point at the atomic level of the dimension and the list of fact keys stored in V points to the fact table rows corresponding to that dimen-sion member. An example can be found in Table 2. There, it is shown that the \Region" dimension contains three ag-gregation levels (from coarser to ner level: \Continent", \Country" and \City"), whereas the \Gender" dimension only contains one level (\Gender").

Relevantly, an index key instantiates a whole aggregation path (i.e., a dimensional value for each level in the hierarchy). For example, the tuples from the fact table with key key_d and key_e correspond to \Europe", \Italy" and \Milan" members of the \Region" dimension. Thus, given that HBase tables are physically stored sorted by key, we can easily pose queries at di erent aggregation levels. For instance, retrieving the \All" aggregation level would mean to scan the HBase table implementing the secondary index using the dimension name as pre x (e.g.,

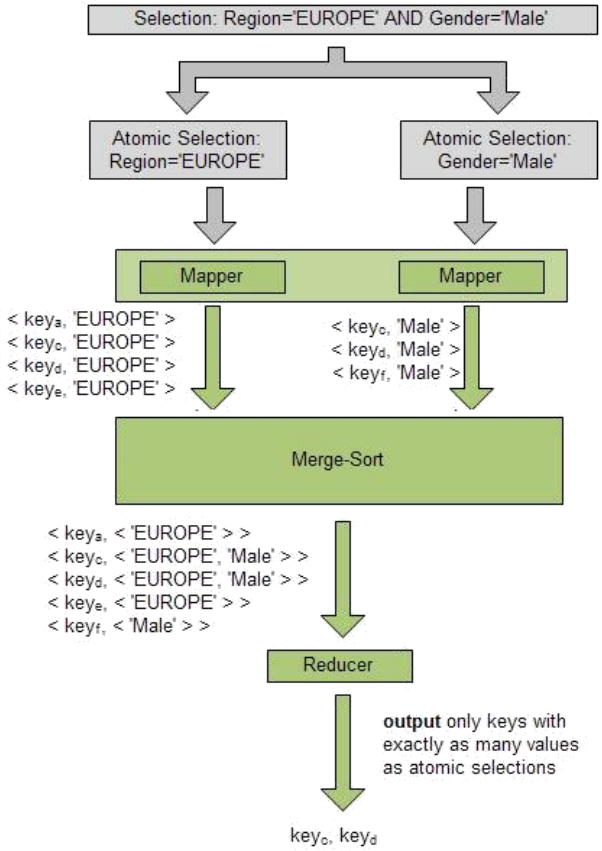

Figure 6: Using secondary indexes with MapReduce

Region). Alternatively, retrieving a ner aggregation level would mean to set the pre x to the desired granularity (e.g., Region%Europe%Italy). Note that this approach re-sembles that of traditional multiattribute indexes [8].

Usage. Without loss of generality, in our implementa-tion, we assume conjunctive selection predicates and, accordingly, our selection algorithm has been implemented as a MapReduce job reading from the HBase tables implementing secondary indexes, with the execution ow as follows:

(i) The selection predicates in the input query are split into atomic clauses. Each clause is stored in an HDFS le.

(ii) A MapReduce job is set to read such les as input and there is a mapper for each le (thus, note the number of atomic clauses impacts on the parallelism provided for this MapReduce job).

(iii) Each mapper reads the corresponding entries of the secondary index in HBase by using a scan object to retrieve the keys corresponding to a certain dimensional member according to the pre x con guration previously discussed (see Section 3.1.2).

(iv) The map functions emit the keys that match the cor-responding clause.

(v) The reducer functions receive each key as many times as the number of atomic predicates this key satis es. As output, it only emits those keys received as many times as the number of clauses in the predicate.

This selection algorithm could be extended so that step (iv) informs whether a certain key matches or not the cor-responding logic clause and step (v) evaluates the parse tree corresponding to the whole predicate. Figure 6 exem-pli es the selection algorithm considering the secondary index depicted in Table 2.

3.2. IRA and FSS Algorithms

In this section, we focus on the MapReduce implementation of the algorithms previously introduced (namely, IRA and FSS) and how they produce the desired data cube according to the input query. We assume input queries following the cube-query pattern [23] (thus, with a multidimensional avour). In terms of SQL, a cube-query statement contains a SELECT clause with a set of (aggregated) measures and dimension descriptors, a conjunction of logic clauses (typically known as slicers) and a GROUP BY clause setting the desired granularity and producing the data cube multidimensional space. Section 5 further elaborates on the characteristics of the queries used in our tests.

Indexed Random Access (IRA). This approach uses secondary indexes to solve the selection predicate in the input query. Thus, it rstly triggers one MapReduce job to query the secondary index (see Section 3.1.2) and then, in a second MapReduce job, it performs a random access to the fact table for each key retrieved by the rst job. Figure 7 depicts the execution process of this algorithm.

Once the set of keys matching the selection predicates has been found (rst MapReduce execution), they are stored in a temporal HBase table where each key is in a dif-ferent row. Note that this automatically sorts the keys, al-lowing then to exploit block cache since those keys accesses are also sorted (see Section 2.2). This temporal table is the input for the second MapReduce job, which builds the data cube by retrieving the right attributes, grouping and nally aggregating. Relevantly, grouping and aggregation are automatically performed by the MapReduce frame-work and thus, in this second MapReduce job we focus on retrieving the needed values. Here, each map function is responsible for looking for the desired attribute values, by means of a random access following the input fact key. Finally, the map emits a [key,value] pair (as shown in Fig-ure 4) and it goes on through the rest of the MapReduce phases to group and aggregate such data.

Full Source Scan (FSS). This algorithm is purely based on the brute force of the cloud by exploiting par-allelism as much as possible. It reads the whole HBase table, nds the tuples matching the selection predicate in the map function and uses the subsequent phases of the

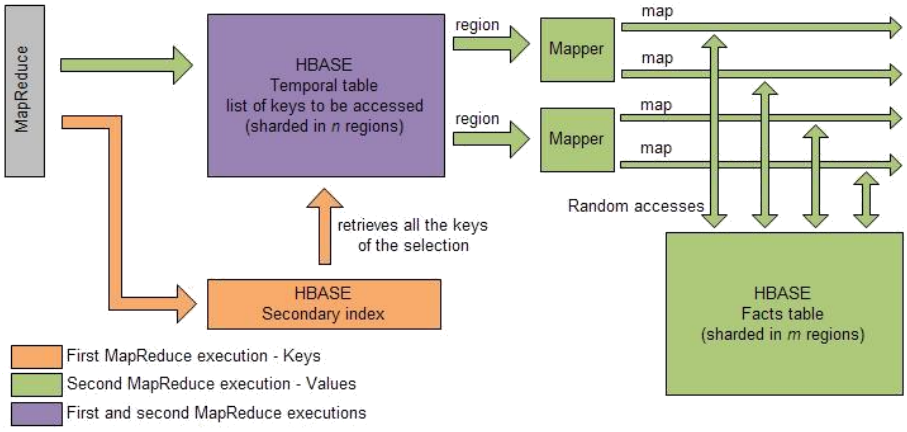

Figure 7: Index Random Access (IRA)

MapReduce framework to group and aggregate data. The example shown in Figure 4 sketches a typical FSS execu-tion. Note that, unlike the previous algorithm, FSS only triggers one MapReduce job.

For this algorithm we just implemented a small opti-mization with regard to traditional MapReduce jobs by using the HBase scan object to lter out those rows not matching the selection predicate. Thus, the map func-tion just needs to rede ne (i) the key as the data cube di-mensional data (i.e., GROUP BY attributes) and (ii) the value as the measure values to be aggregated in the reduce function. Like in the previous approach, the MapReduce framework automatically does the grouping and the aggre-gation is implemented in the reduce phase.

4. Cost-based Formulas

Our ultimate goal is to estimate the cost of each of the algorithms presented in Section 3.2 (depending on the partitioning in Section 3.1.1). Consequently this section introduces, in a rst step, the factors that will take part on the formulas that, in a second step, will be used to perform such estimations.

4.1. Cost Factors

In this section, we focus on the two main cost factors detected, which deserve further discussion to be precisely de ned in terms of Hadoop.

4.1.1. Read Cost

This is a well-known cost (also for RDBMS) related to retrieving blocks from disk. The more blocks to read, the higher the cost. Here, disk blocks refer to HBase blocks (see Section 2.2) and it corresponds to the overall number of blocks to be read by the algorithm.

A relevant factor a ecting the read cost is the verti-cal partitioning strategy applied by HBase. In presence of vertical fragmentation, when it comes to reading a certain attribute, HBase may not need to read the whole tuple but just the stores containing such attribute. As explained in Section 2.2, vertical partitioning is performed after hori-zontal partitioning in HBase and thus, we should not talk about families, but about stores (which there is exactly one per family and region). For instance, if the attributes a and b belong to the same family f, then only those stores related to f must be read. However, in case they are stored in di erent families f_a and f_b, respectively, then all the stores for both families must be read.

4.1.2. Fetch / Flush Cost

The le system (HDFS), the database (HBase) and the query answering engine (MapReduce) are three di erent processes so they do not share memory. Consequently, they communicate to each other through the network by means of Remote Procedure Calls (RPC), which means that a call to this communication protocol happens each time a certain amount of rows is sent from one component to another and thus, it must also be considered as a main factor as well as the involved network costs.

HBase data are ultimately stored in HDFS chunks and wrapped in a speci c format (see HFile in Figure 3), so HDFS reads these data from the le system but it is unable to understand them. Therefore, it is HBase responsibility to interpret the data received from HDFS and properly apply the scan properties (i.e., those of the HBase scan object) on the tuples. Afterwards, the fetch cost pops up again when sending data from HBase to MapReduce (i.e., when a MapReduce job is con gured to read from HBase tables). Thus, this cost should be considered in both cases. However, for the sake of simplicity, we will only consider the transmission cost between HBase and MapReduce. Note that by doing so, we do not diminish the cost of moving data between HDFS and HBase but we contemplate it as part of the read/write cost explained above (sending data to the client asking for it is normally considered part of the read/write task).

The fetch/ ush cost becomes more important when it comes to moving data across the cloud, but it is even rel-evant when source and target sit in the same machine.

Variables	Description
t_D	Time to access a disk block
t_{RPC}	Time of one RPC call
t_{byte}	Time to transfer one byte through the network
t_{MR}	MapReduce start-up time
$t_{shuffle}$	Time involved since mappers write their temporal results until reducers read them
P_X	Parallelism provided by \X" (i.e., the maximum number of MapReduce subtasks running at once)
B_X	Number of blocks of \X"
R_X	Number of rows per block of \X"
$\|T\|$	Cardinality (i.e., number of rows) of the fact table T
family row length$_i$	Average overall length of the attributes to be retrieved from family i^{th}
fkey length	Average space per fact key in the index
block_size	Size of a disk block (i.e., 64Kb)
#f	Number of families to be read
#	Number of slicers in the predicate
$S_{f[i]}$	Selectivity factor (i.e., percentage of tuples in the output wrt the input) of the predicate [or i^{th} slicer]
C	HBase scan bu er size

Table 3: Variables used in the costs formulas

This is a well-known bottleneck in the Hadoop ecosystem and nowadays we can nd Hadoop-derived products, such as Cloudera Impala [10], that reduce the impact of this cost by coupling the di erent components and communi-cating through main memory. Hadoop v2.2.0 also tackles this issue by implementing \Short-Circuit Local Reads" in HDFS, as explained in [24]. This technique allows a DFS Client (HBase) to directly read data bypassing the DataN-ode (HDFS); see Figure 3. Of course, this can only be used when they are both located in the same machine. How-ever, note that this solution does not solve the fetch/ ush cost between HBase and MapReduce, but between HDFS and HBase.

4.2. Cost Formulas

In this section, we aim at estimating each algorithm cost. These formulas come from the knowledge gathered at studying the Hadoop ecosystem and build on top of the main cost factors discussed in the previous section. In the spirit of relational query optimizers, these formulas are meant to be the seed of a cost-based model to deploy a query optimizer for Hadoop, which is the main objective of our future work.

Prior to introduce the costs formulas, we would like to start de ning the variables used in this section. Table 3 shows their meaning.

Index Random Access (IRA). IRA consists of two MapReduce jobs: (i) for accessing the secondary index, and (ii) for retrieving those values of interest from the fact table. The cost formula of IRA should be the sum of these two tasks plus the cost of starting two MapReduce jobs (which is only relevant when we are processing small amounts of data), as shown in Formula 2.

$$IRA = IRA_{index} + IRA_{table} + 2t_{MR} \quad (2)$$

Thus, since the rst MapReduce accesses the secondary index (which is an HBase table) once per slicer, we rst estimate the amount of blocks read (Formula 3). For each access, at least, one block is read but in general additional blocks may be read depending on the number of keys to retrieve and the number of keys stored per block in the index. Thus, to compute the number of blocks we rst weight the cardinality of the table with the slicer selectivity factor (i.e., the number of fact keys we need to retrieve from the index) and multiply this value by the average size of each key, which is the size of all the keys in bytes. Note that, in the worst case, the rst key is always read when accessing the rst block and that is the reason to subtract one to the number of keys to be read in subsequent blocks. Finally, we compute the number of blocks by dividing the size of the keys read by the size of the block (in bytes).

$$\overline{B_i} = 1 + \frac{(S_{f_i}|T| - 1)fkey\_length}{block\_size} \quad (3)$$

In Formula 4, we estimate the overall cost of accessing the index as, in rst term, (i) the blocks read for all the slicers in the query predicate and (ii) transferring them to mappers (i.e., fetch cost), plus the shu e cost (i.e., the cost of merging and sorting the output from mappers and the cost of storing and reading intermediate results from HDFS), plus the cost of transferring the nal result to HBase (one RPC call per key, the network cost of sending the data, and the cost of writing the temporal table storing the selected keys).

The rst factor is weighted by the parallelism pro-vided when querying the secondary index. Note hence that P_{index} describes the workload portion that can be run si-multaneously in the mapper task accessing the secondary index. However, such parallelism degree depends on how the MapReduce job input is split. According to what has been explained in Section 2.3, it is the minimum of the number of slicers, the number of regions in the index, and the number of RegionServers. For instance, if the number of regions in the index is one (or the number of slicers is one), it does not matter how many RegionServers there are in the cluster since the MapReduce input will not split and no parallelism would be provided at all. Note that, in general, this is di erent from the parallelism of ushing and writing the temporal table, which is bounded by both the number of reducers, and the number of regions gen-erated in that table. The fetch cost F_{index} also involved in this part of the formula depicts the cost of moving the

$$IRA_{index} = \frac{(t_D \sum_{i=1}^{\#} B_i) + F_{index}}{P_{index}} + t_{shuffle} + \frac{S_f \, T \, (\frac{t_{RPC}}{C} + t_{byte} \, fkey\_length) + B_{temp} \, t_D}{\min(P_{reducer}; P_{temp})} \quad (4)$$

selection keys from the secondary index to mappers (see Formula 5). For each slicer, there is one RPC call needed to request that secondary index entry plus the cost of send-ing through network as many bytes as the whole set of keys related to such slicer occupies.

Afterwards, the MapReduce shu e cost comes to play. Firstly, note that the keys stored in the secondary in-dex are lexicographically sorted and thus, when processing these keys in this MapReduce job we do not need to con-sider the full cost of the Merge-Sort phase since the output of the mappers is already sorted as their input is. In other words, having the keys already sorted in the secondary index means there is no Sort cost at all during the Merge-Sort, but yet keys outputted from di erent mappers need to be merged so there is still need of considering the Merge cost. For the sake of simplicity, we do not go through the details of this cost (e.g., see [8] for more details on the Merge-Sort cost). Secondly, and as it was pointed out in Section 2.3, intermediate MapReduce results are written in HDFS. The $t_{shuffle}$ variable re ects the cost of interacting with HDFS to store the intermediate results.

$$F_{index} = \sum_{i} (t_{RPC} + t_{byte} S_{fi} \, T \, fkey\_length)$$

The value for B_{temp} can be estimated by using For-mula 6. We just multiply the number of keys in the output by the size of each fact key, then this is divided by the size of the block. This value is rounded up since it corresponds to the precise number of blocks needed (thus, this value is not an average like B_i). Note that data inserted in HBase is rst stored in in-memory bu ers (i.e., memstores) as stated in Section 2.2. According to this, it could be the case that the whole temporal table ts in a single mem-store so there would be no need of ushing it to disk. In such situation, our formulas should take $B_{temp} = 0$ since no physical blocks are written and no memory costs are considered. For the sake of simplicity, we then assume this temporal table is fully either in disk or in main memory, though a real-world scenario could contemplate a situation where it is partially in disk and partially in memory. Con-sequently, the condition to whether enable B_{temp} or not is $S_f \, T \, fkey\_length > mem:size$.

$$S_f \, T \, fkey\_length$$

After accessing the index it is time to retrieve the right data from the fact table. This second cost is depicted in Formula 7 and resembles how we accessed the index. In this case, P_{temp} refers to the parallelism provided by MapReduce when accessing the temporal table, and it is de ned as the minimum of the number of RegionServers, and the number of regions of the temporal table. In other words, P_{temp} corresponds to the input split available for the second MapReduce job. However, we may expect a low number of regions for the temporal table, since this is several times smaller than the fact table (it only stores fact keys) and as consequence, the HBase horizontal par-titioning strategy barely splits the table (see Formula 1 in Section 3.1.1), which, in turn, a ects the parallelism provided at this step.

$$B_{temp} = \frac{block\_size}{}$$

The numerator then corresponds to reading the blocks from the intermediate table, plus the cost of reading the necessary blocks from the fact table, plus the fetch cost of retrieving those data from HBase to MapReduce. Despite coming from random accesses, there is a probability that two fact keys fall into the very same block of the fact table. Since the input is sorted by key and HBase implements a cache (see Section 2.2), it may happen that the second key does not produce any real disk access but a hit in the cache. Thus, we estimate the percentage of distinct blocks to be read as $(1 - (1 - S_f)^{R_i})$. Note that this scenario depends on the selectivity factor (the more tuples to be retrieved, the higher the chance), but also on the number of rows per block in each family (R_i). (5)

The cost F_{table} depicts the fetch cost when it comes to the fact table and also depends on the number of tuples and the row length in the corresponding families as shown in Formula 8. Firstly, note that, since we are using a secondary index for the selection in this algorithm, neither the rows of no interest nor the selection attributes must be considered. Thus, we only have to send the measures and dimensions of the rows that matched the selection during the rst MapReduce execution.

Moreover, when it comes to sending data from HBase to the client, the HBase scan object can be con

gured to pack a certain amount of tuples C together, and send them at once. By doing so, HBase bene ts from network bandwidth, but it uses more memory to implement the needed bu er. Note that IRA would not bene t from the bu er, because it retrieves one row per map function from the mapper, which implies C = 1. Nevertheless, we have included C in the fetch cost formula F_{table} to show that it is generic and can be then reused for FSS (by just using a di erent value of C).

(6) Thus, the nal IRA fetch cost for the fact table is given by the cost of performing one RPC call plus sending as many data as needed in each packet (which depends on the number of families and their row length), multiplied by the number of packets (i.e., one per row in the output).

$$IRA_{table} = \frac{B_{temp} t_D + \sum_{i=1}^{\#f} (1 - (1 - S_f)^{R_i}) B_i t_D + F_{table}}{P_{temp}} \quad (7)$$

$$F_{table} = S_f \left\lceil T_j \frac{t_{RPC}}{C} \right\rceil + t_{byte} \sum_{i=1}^{\#f} X_i \cdot \text{family row length}_i \quad (8)$$

Full Source Scan (FSS). The baseline for the comparison is the full scan. As its name suggests, the cost of this approach consists basically in reading the whole table. Thus, the execution cost is as denoted in Formula 9. Note that this performs the selection by reading the whole table, so there is no need for a previous MapReduce job accessing the index. This also means that those families containing the selection attributes must be read as well and, unlike IRA, #f additionally includes those families containing selection attributes.

$$FSS_{table} = \frac{t_D \sum_{i}^{\#f} P_i B_i + F_{table}}{P_{table}} + t_{MR} \quad (9)$$

The fetch cost F_{table} in this formula is given by shipping families containing measures, dimensions and selection attributes for those rows matching the selection. Rows are first read from HDFS and sent to HBase. Then, HBase applies the object scan configuration to the received rows and filters out the undesired tuples. Since this selection process is done at this reading time, these non-matching rows are not sent to the MapReduce task. Accordingly, the fetch cost is computed as in Formula 8. In this case, we can benefit from buffering the tuples by configuring a high value of C. Last, but not least, note that we do not consider the Merge-Sort cost of the MapReduce job since we aim at comparing IRA and FSS and, at this stage, in both cases the same amount of data will go through the Merge-Sort. Consequently, this factor has been simplified from the IRA_{table} and FSS_{table} formulas.

As a matter of fact, note that our formulas assume an even distribution of data across the cluster. If this were not the case, the skewed distribution would affect the P_X variables, which represent the parallelism achieved in the MapReduce jobs. Additionally, note that P_X are also affected by the MapReduce configuration parameters, which we keep at their default values (e.g., number of mappers and reducers). In this sense, our work mainly focuses on database tuning (i.e., at the HBase level), impacting on the variables in the numerator of the formulas, and it complements previous works working at the MapReduce framework (such as [12, 13]), which introduce tuning techniques that would maximize the value of each P_X.

Test parameters	Values
Scale Factor (SF)	2 (60 GB), 4 (120 GB) and 6 (180 GB)
Number of queries	15 (see Table 5)
Number of Region-Servers	2, 5 and 8
Vertical partitioning strategies	ColumnFamily, A nityMatrix and SingleColumn
Compression	GZ and none

Table 4: Summary of the factors and values to test

5. Experimental Setting

Next, we aim at validating the cost formulas discussed in the previous section by means of empirical testing and, accordingly, we devised a thorough battery of tests. As previously discussed, we focus on database tuning and we avoid playing with the configuration parameters of the Hadoop ecosystem. The experiments were devised considering the following primary factors: (i) the database size, (ii) the query topology, and (iii) data partitioning. First, we present the parameters used to generate different configurations of these factors. Note that replication is set to 1 and is not tested in our experiments. The reason is that testing the system availability and robustness is out of the scope of this paper. Then, for each resulting combination the IRA and FSS algorithms were triggered and we kept trace of the performance obtained in each case.

All tests have been performed in an homogeneous user-shared cluster but limited to one CPU per machine (since the other is exclusively used for the cluster management). The number of machines used is variable. Thus, we run the same experimental setting but using 2, 5 and 8 nodes (more details in Section 5.2). Nevertheless, as stated pre-viously, the machines used are homogeneous and the spec-i cations are as follows:

2 CPUs Intel Xeon Dual-Core 2,333 GHz, FSB 1333 MHz, 4 MB Cache.

12 GB RAM.

Hard disk SEAGATE Barracuda 320 GB S-ATA-2.

2 NICs Intel Pro/1000 Gigabit Ethernet

As a mere summary of what is going to be explained next, Table 4 shows the test parameters.

Cardinality of	Min	Max	Mean	Median
Projection	1	9	5	3
Grouping	0	6	3	1
Selection	2	8	5	2
Selectivity factor	10^{-5}	1	NA	10^{-2}

Table 5: TPC-H query statistics

5.1. Database Size

The input database was populated according to the TPC-H speci cation (see [25]). However, the insertion process was modi ed to load a single fully denormalized fact table. The data volumes chosen (so called Scale Factor, SF from here on, in the TPC-H benchmark) were 2, 4 and 6. In the normalized TPC-H, these SF s correspond to 2, 4 and 6 GBs, respectively. However, in our case, these SF s turned approximately into 60, 120 and 180 GBs, respec-tively. The reason of such di erence is mainly data denor-malization but also because HBase stores for each attribute value, the key, the family, the quali er and the version it belongs to (i.e., all its metadata). This also means that both read and write costs are topped by these additional metadata, and we include these in the family row length and fkey length values.

5.2. Query Topology

The queries have been de ned as a summarization of the real TPC-H queries and are aimed at testing the three main predicates of a cube-query: the cardinality (i.e., number of atttributes) of grouping, projection and selection attribute sets, plus the query selectivity factor. In order to do such summarization, the process applied has been to test the minimum, the maximum and the mean of each of these values (according to the TPC-H queries), while other features remain at the median (which measures the centrality of the distribution much better than the mean). The selectivity factors tested are powers of ten, between the minimum and the maximum in TPC-H. For instance, if the TPC-H query with the lowest projection cardinal-ity is one, and the highest is nine, we have de ned three queries with one, ve and nine projection attributes, while the rest of features are set to their median when projection attributes are studied. Table 5 shows the values to test for each characteristic.

5.3. Data Partitioning

In section 3.1.1, it is explained what factors a ect the HBase data partitioning policy. Accordingly, the values assigned to each of these factors are as follows:

The number of RegionServers to test is 2, 5 and 8. We chose 2 because it is the minimum number of RegionServers to deploy a distributed system. Then, we choose 8 as a number large enough as to test the difference between both settings (as rule of thumb, previous works argued that an 8-machine Hadoop cluster competes in performance with parallel databases; e.g., [13]) but at the same time being reasonable as to be able to trigger a large amount of tests. Finally, ve RegionServers because it is the mean of the other two.

The vertical partitioning is also tested by three different strategies. The rst one is to use one family per attribute. Since there are approximately sixty attributes in the TPC-H, we are then using sixty families as well. We will refer to this vertical partitioning strategy as ColumnFamily from now on. The second strategy is the other way round and thus a single family stores all the attributes (SingleColumn strategy). Finally, in order to test out some intermediate strategy between these two, we use the a nity matrix algorithm to compute a nities between attributes and decide how to partition [16]. The result after applying the a nity matrix is a family group-ing six attributes (more precisely, the six attributes used for the projection, grouping and selection medi-ans, which are repeated in 12 out of 15 queries each) whereas the rest remain in an attribute per column (meaning their a nity is too low as to group them). We will refer to this strategy as A nityMatrix.

For compression only two values are tested (either no compression or using GZ). The reason is that the GZ algorithm is the only one natively o ered by HBase.

As discussed in Section 4.2, our formulas assume an even distribution of data. As we have seen in Section 3.1.1, however, the default split policy in HBase has some de - ciencies that do not guarantee an even distribution of data. Such de ciencies are put into numbers in Figure 8. This Figure depicts the standard deviation of the distribution of data obtained when varying the SF and the number of RegionServers in a ColumnFamily scenario. Lower standard deviations indicate evener distributions of data. On the one hand, the higher number of RegionServers we have, the lower standard deviation we obtain (see Formula 1). On the other hand, increasing the data volume always worsens the uniformity of data distribution as long as regions do not split at constant pace.

For this reason, in our experimental setting we guaranteed an even distribution of data by using the presplit functionality HBase provides. This functionality allows a table to split before inserting data based on some criteria. In our setting the keys are designed consecutively (although not generated/inserted consecutively) and used to presplit the table and distribute it. Once the insertion process starts, the default split policy takes place but, comparatively to the previous situation, several regions are now created beforehand, populated in parallel and therefore growing and spliting at a similar pace. Oppositely, without presplitting, the table was initially composed by one region placed in one RegionServer. Thus, that region (respectively, that RegionServer) received all the insertions until the rst region split took place. Indeed, many splits were needed

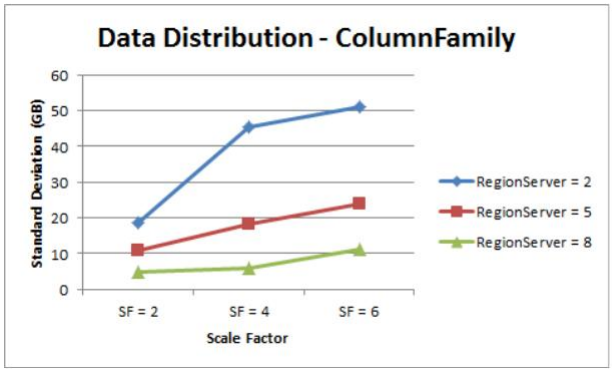

Figure 8: Standard deviation of the data distribution for the ColumnFamily strategy

Scale Factor	6
Number of RegionServers	8
Compression	NONE

Table 6: Test factors xed

before all the RegionServers in the cluster came to play and hence being detrimental to parallelism.

One may be tempted to think that such approach is only valid when knowing the keys beforehand. How-ever, carefully designing the key to evenly distribute the workload is a well-known technique known as key-design [26, 27]. For example, a poor key design would be to use the insertion timestamp as key because the rows would then be always stored in the most recent region. Oppo-sitely, a good key design must guarantee that all regions are constantly active (i.e., storing new data) and there-fore leveraging the distribution of data. For example, we may use salted[6] timestamps, where the salt is generated arti cially, uniformly and proportionally to the number of machines. In general, the key-design problem is an orthog-onal issue to be carefully considered for each system.

6. Discussion of Results

In this section, we discuss the conclusions drawn from the battery of experiments carried out. First, we argue about the correctness of the formulas presented in Section 4.2 by justifying that (i) no relevant parameter has been omitted to devise the cost formulas and then we show that (ii) these formulas properly predict the best algorithm in 98,15% of the cases, given an even distribution of data in the RegionServers. All in all, validating the feasibility of using these formulas to predict the behaviour of Hadoop.

6.1. Relevant Cost Parameters

In Section 5, we have discussed what parameters were used to characterize the query topology and data partitioning. Here we discuss the conclusions drawn for each

[6] A salt is random data used as additional input of a function. For instance, in cryptography, salts are used to wrap hash function inputs into more complex inputs.

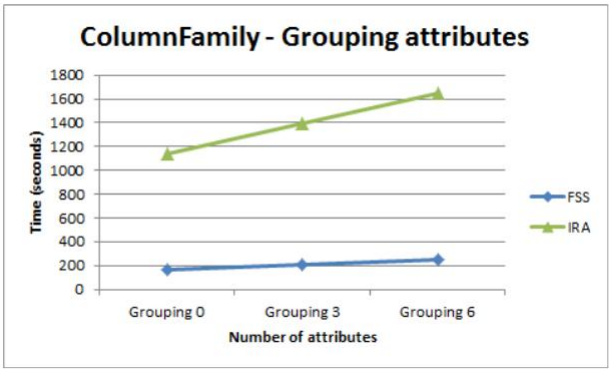

Figure 9: Grouping attributes in ColumnFamily

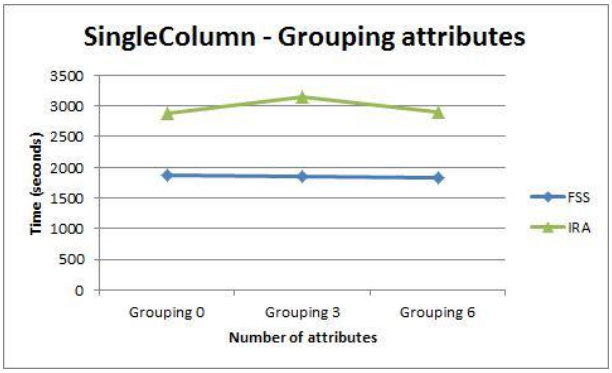

Figure 10: Grouping attributes in SingleColumn

of these factors. For the sake of simplicity, those factors related to partitioning are xed to their highest value (as shown in Table 6), when testing the query topology. Oppo-sitely, on testing partitioning, values modifying the query topology are xed to the median.

6.1.1. Query Topology

From the results obtained for the query topology study we draw the following conclusions:

Grouping and Projection cardinality. Fig-ure 9 depicts the behavior of each algorithm with queries evaluating the grouping cardinality under the ColumnFamily vertical partitioning strategy.

This gure clearly shows that reading more attributes increases the cost since more families must be read. Indeed, the ColumnFamily strategy raises a 100% e ective read ratio since there is one family per attribute and only relevant attributes are read. Oppositely, Figure 10 shows that the SingleColumn strategy is not a ected by the number of attributes to be read.

In case of using the A nityMatrix strategy, the tests show an intermediate e ect, as expected. When reading a new attribute from a family already read leads to no additional cost, but if the attribute is stored in another family not yet read it increases the read cost.

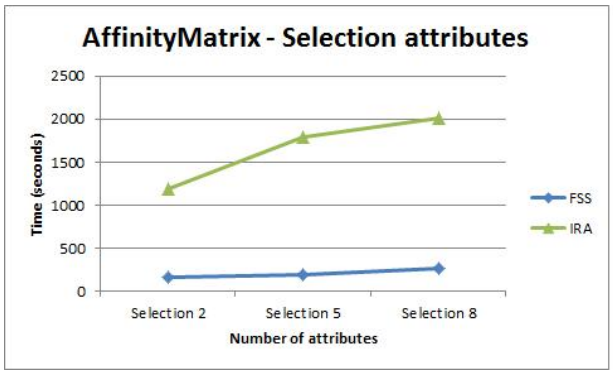

Figure 11: Selection attributes in A nityMatrix

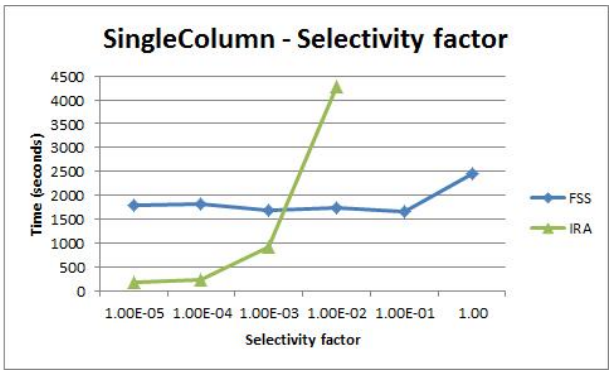

Figure 12: Selectivity factor in SingleColumn

Importantly, the same explanation provided for grouping attributes holds for projection attributes.

Selection cardinality. The number of selection clauses (i.e., slicers) impacts on the cost depending on the selection algorithm we are applying: either (i) using secondary indexes, like IRA or (ii) by accessing the fact table and evaluating the selection predicate on the values of the tuple, like FSS. On the one hand, (i) represents the number of random accesses to be performed to the secondary index (one access per slicer in the query). On the other hand, (ii) means reading more or less families (depending on the vertical partitioning strategy), so the ratio-nale presented for grouping and projection attributes also holds here.

Our tests show that accessing the index becomes more costly as the number of attributes to be read increases (see Figure 11). Note that in this case we are focusing on a relative comparative between (i) and (ii) and how these two scenarios a ect the read cost and it must not be understood as an overall query performance discussion. Indeed, since some factors have been xed to constant values (see Table 6), the overall performance cost refers to this scenario. The impact of those other factors will subsequently fol-low.

Selectivity factor. The selectivity factor showed to be the most relevant parameter for the query topology. While the three previous factors tell us the num-ber of families / attributes to be read, the selectivity factor tells us the number of rows to be read. Thus, the selectivity factor allowed us to perform a rst approach to how each algorithm performs compared to the other.

Consider now Figure 12, which clearly shows that the selectivity factor plays a crucial role to choose between the two algorithms. On the one hand, IRA performs better when dealing with low selectivity factors (which is the expected outcome since IRA was precisely designed to match the behaviour of indexes in RDBMS and perform random accesses instead of a full table scan). Note it grows exponentially as the selectivity factor does. The last two values have been removed from this Figure in order to avoid detracting it (but they respectively correspond to 11800 and 56295 seconds). On the other hand, FSS o ers better results as the selectivity factor grows. As more data has to be retrieved from the table, random accesses become more costly and sequential reads become more e cient (even if the whole table is to be read). Thus, note that the same behaviour as in RDBMS is shown for the Hadoop ecosystem. Note though the steep increase in the tail of the FSS graph. This increase is not due to the read cost (FSS always reads the whole table) but to the fetch cost, which is strongly related to the query selectivity factor. In this gure, the selectivity factor increases by powers of 10 and for $S_f = 1$ the whole table is shipped to MapReduce, whereas only 10% of the tuples are sent for $S_f = 10^{1}$. Comparatively, this gure also shows the lack of parallelism behind the IRA approach (which depends on the number of atomic selection clauses and the size of the intermediate table) and its quick performance degradation, since (i) the temporal table produced as an intermediate step in IRA does not split in enough regions as to match the parallelism of FSS (which depends on the number of regions of the table), and (ii) the fetch cost is computed in IRA by means of C = 1 (see Section 4.2).

6.1.2. Data Partitioning

For the data partitioning study we draw the following conclusions regarding the three factors impacting on how data is partitioned in Hadoop (see Section 3.1.1): the ver-tical partitioning strategy, number of RegionServers, and compression rate.

Vertical partitioning. As outlined in the query topology discussion, the vertical partitioning strategy resulted to be a crucial parameter in our tests. Indeed, it impacts on the data volume to be read

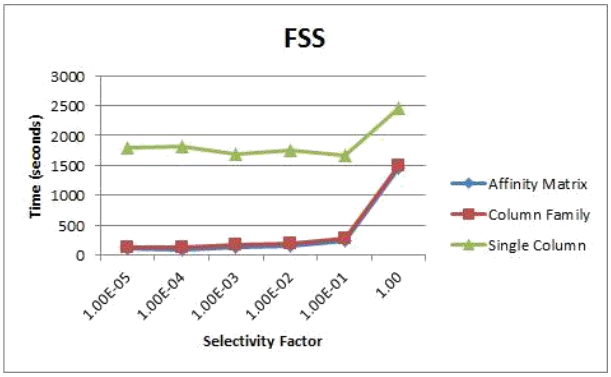

Figure 13: FSS performance regarding the vertical partitioning

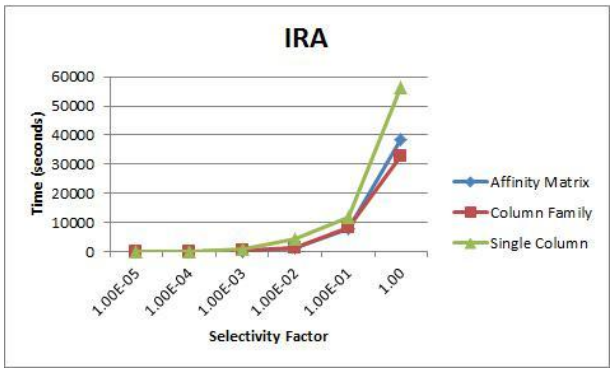

Figure 14: IRA performance regarding the vertical partitioning

for a query. In general, a strong vertical partition-ing leads to an optimal read cost. To better ex-emplify this, Figure 13 depicts FSS performance for the three vertical partitioning strategies introduced in Section 3.1.1. There, the performance clearly im-proves when data is partitioned in a precise manner with regard to the attributes required by the query at hand (i.e., ColumnFamily and A nityMatrix). Note that, again, we relate the vertical partitioning strat-egy to the selectivity factor, as they are clearly cor-related, whereas the number of grouping and pro-jection attributes, as well as the number of selec-tions are xed to the median. Using a SingleColumn strategy clearly worsens the performance, regardless the selectivity factor, since the amount of attributes read (including those not requested by the query) is bigger. In addition, the fetch increase previously discussed is re ected in all three vertical partitioning strategies.

Figure 14 shows the behaviour of IRA. Oppositely, the performance of IRA does not clearly depend on the vertical partitioning strategy. This result is sound because IRA relies on random accesses and it exploits the HBase B+ index to nd the target row. However, the vertical partitioning strategy still has a certain impact on the algorithm performance, because the index tells us the region where to nd

RS/SF	2	4	6
2	AM	CF	CF
6	AM	AM	CF
8	AM	AM	AM

Table 7: Decision table for the best vertical partitioning

that row but it depends on the vertical partitioning strategy to either read one store le containing one needed attribute or a larger one containing several unneeded attributes.

Table 7 further elaborates on the best vertical partitioning strategy regardless of the algorithm used and based on the Scale Factor (i.e., size of the workload to deal with and roughly speaking the parallelism consequently needed) per column, and the number of RegionServers (parallelism provided) per row. The abbreviations are as follows: AM stands for (A nityMatrix) and CF for (ColumnFamily). Note that SingleColumn does not even appear in the table.

Relevantly, this table holds for all the selectivity factors and the two algorithms tested. Speci cally it shows that when the workload is too large for the parallelism provided (i.e., more parallelism would be needed) a ColumnFamily strategy is preferred as the e ective read ratio increases (no unneeded at-tributes are read). Oppositely, when the provided parallelism is enough to deal with the workload pro-vided then using an A nityMatrix strategy results in a better performance (since reads are more se-quential and therefore they bene t from parallelism, even though the 100% attrubute e ectiveness ratio is not achieved like in ColumnFamily). This table pro-vides valuable guideline for the designer. Following this reasoning it means that if we extend this table by adding new rows representing experimental settings with more RegionServers (i.e., larger amounts of par-allelism provided) the SingleColumn strategy should eventually appear in the table as the best option. To verify this assumption we triggered a testbed with all the machines in the cluster (i.e., 23 nodes). At this point, the SingleColumn strategy was not yet able to improve the performance of the A nityMa-trix but we veri ed that with a greater number of RegionServers the relative performance gap between both strategies drastically diminishes. According to this evidence, the SingleColumn strategy is likely to appear in Table 7 in the presence of a large num-ber of RegionServers, although we were not able to determine the precise number.

Figure 15 elaborates on the insertion performance regarding the three vertical partitioning strategies. There, a huge gap in performance can be seen when inserting, especially between ColumnFamily and the other two. This is related to the number of write operations needed regarding the vertical partition-

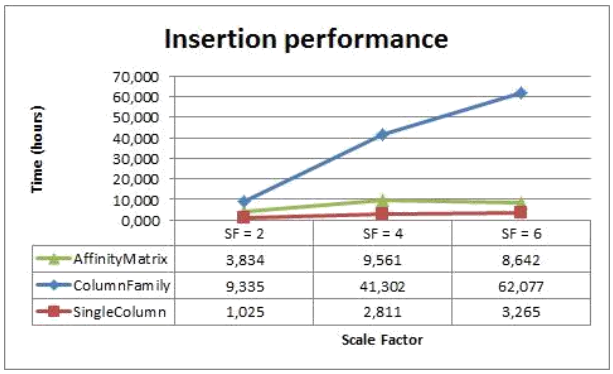

Figure 15: Insertion Performance

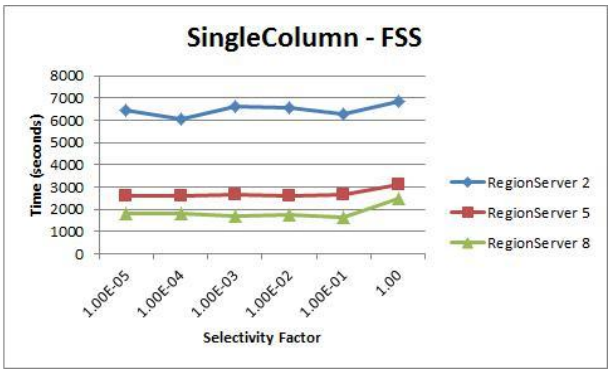

Figure 16: Performance given by the number of RegionServers

ing and accordingly the ColumnFamily is largely a ected, whereas the SingleColumn raises as the cheapest solution. As usual, the A nityMatrix remains as a middle ground solution.

As conclusion, the decision to apply a certain verti-cal partitioning strategy must be taken with regard to the size of the database, the number of machines available and the frequency of writes. On the one hand, when it comes to reads, the more parallelism provided by the system, the lower the a nity threshold to use when grouping attributes in families. Alternatively, grouping attributes with very high a ni-ties is mandatory. On the other hand, when it comes to write, the lower number of families, the bet-ter. Thus, the SingleColumn strategy is preferable for write intensive workloads (e.g., OLTP), whereas stronger partitioning strategies are preferred for read only workloads (e.g., OLAP).

Number of RegionServers. The number of RegionServers, in the presence of enough regions, has a positive e ect on the nal performance, as shown in Figure 16, where the con guration shown in Ta-ble 6 also applies. There, the overall performance of FSS with a SingleColumn strategy drastically im-proves as we pass from 2 to 5 servers. Similarly, we still have a gain when passing from 5 to 8 servers. However, the gain is relatively smaller. This result is an empiric evidence of a well-known trade-o of distributed systems formulated in di erent laws such as the Universal Scalability Law [28], which argue that the performance gain is not linear due to contention.

In our tests, due to the small size of the intermediate temporal table containing only keys (which is not really partitioned), IRA performance remains mostly una ected by the number of RegionServers. Indeed, the same conclusions drawn for the e ect of vertical partitioning strategies on IRA can be mapped to this scenario.

Compression and data volume. The tests carried out for compression raise the same evi-dences previously discussed. Compression reduces the amount of data to read and send from HDFS to HBase, but it trades with the additional cost of decompression since HBase is responsible for decom-pressing data and ship it to MapReduce. Conse-quently, the fetch cost is una ected by compression. Our results show that compression must only be con-sidered when designing very large tables and there is a explicit need for reducing the amount of data stored on disk. In any other case, the decompres-sion would add an additional cost that would over-take the bene ts of compressing stored data since, from the point of view of MapReduce, compression reduces the size of the data stored in HBase so hori-zontal partitioning is a ected and, in turn, the num-ber of regions, eventually hurting the overall paral-lelism achieved in the system. For these reasons, we decided not to consider compression in our formulas. However, adding compression would simply mean to add the decompression CPU cost to the reading cost in our formulas, and considering the compressed sizes.

6.1.3. Final Discussion

Summing up, the main factors to be considered when choosing between IRA and FSS are (i) the query selectivity factor and the database size, (ii) the vertical partitioning strategy applied to data and (iii) the number of RegionServers available. Importantly, all these factors are considered in our formulas, which do not simply consider the parallelism provided but show the relevance of database tuning in Hadoop. However, even if the price of having an expert DBA able to perform such tuning may put sev-eral organizations o and rather use the brute force on the cloud, our formulas show that the impact of the tuning in cloud databases is not to be diminished as it has nowadays been systematically done.

Indeed, a careful look at our formulas shows that adding more machines (i.e., RegionServers) would increase the values in the denominator of the formulas, while database tuning, in a smarter move, would reduce the values in the numerators. Consequently, improving the

Parameter	Value
t_D	0,002 s.
t_{RPC}	0,001 s.
t_{byte}	0,000001 s.
t_{MR}	30 s.
block_size	64Kb
C	100

Table 8: Parameter values for cost estimation

overall performance means to either (i) decrease the values computed in the numerator, and / or (ii) increase the denominator. We accordingly claim that database tun-ing is still relevant, and not to be ignored. However, it is also true that there is a limit for the optimization ob-tained by database tuning. In short, the numerator sets the workload for each machine in the system (represented in the denominator) and, for this reason, there will always be a point where no further optimization can be achieved without adding more machines into the system.

6.2. Predicting the Right Access Plan

Next we proof the accuracy of our cost formulas by comparing their results against the empirical tests conducted. To do so, we triggered an exhaustive testbed con-sidering the main factors discussed in Section 6.1.3. The values appearing in Table 8 are those used for computing the formulas. Some values (such as block size, number of families, size of the families, etc.) are precisely de ned but times and lengths have been empirically estimated[7], which may have introduced an error when computing our predictions. Con guration parameters were kept at their default value.

The results obtained are shown in Figure 17, where the rst column refers to the selectivity factor. The yellow and red cells correspond to the cases where our formulas failed to predict the best algorithm, while the rest show agreement. Thus, the number of right predictions (blue and green) is 159 out of 162 queries, which yields a 98.15% of accuracy. Accordingly, taking these formulas for a cost-based optimizer would correctly predict the best access plan in more than nine out of ten queries. This result holds for a balanced workload distributed throughout the cluster. Oppositely, if the default balancing mechanism is used, these same tests yield an accuracy of 90.12% because of the reasons discussed in Section 5.3.

The prediction errors appeared are part of the trade-o between complexity and accuracy when devising the cost formulas since, in general, by making our formulas simpler we incur in less computational cost but, in turn, the overall accuracy is worsened. Therefore, prediction errors are unavoidable to pop up in some cases.

[7] For the t_{byte} value we considered a very small value trying to simulate the cost of sending one byte through network, though this is not possible in a real scenario due to the TCP/IP protocol (i.e., packet size, headers, etc.).

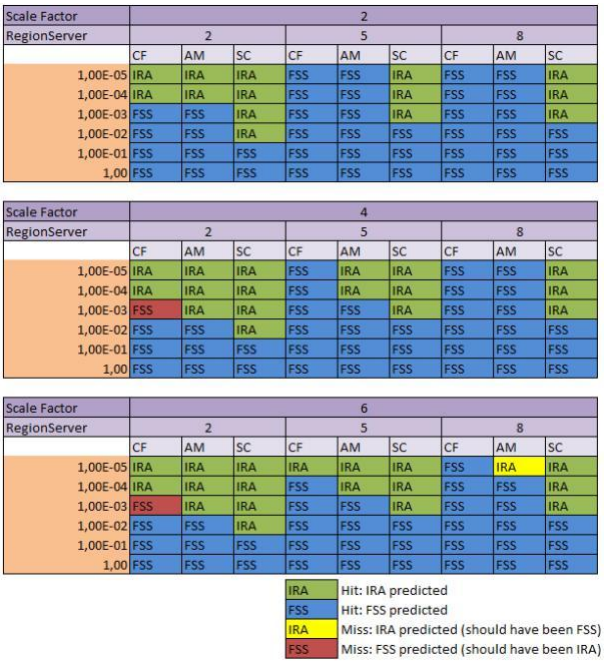

Figure 17: Prediction of the Best Access Plan

All in all, these results justify the feasibility of building a cost-based optimizer for Hadoop. Previous experiences with RDBMS showed that cost-based query optimizers are preferable to rule-based ones, due to the di culty to iden-tify optimization rules properly characterizing the system main factors. In addition, alternative algorithms to re-trieve data from the sources are also needed. For instance, the IRA performance improvement ratio in those scenar-ios providing less parallelism (i.e., SF = 6 with only two RegionServers and SingleColumn con guration) is x35.06, x34.08 and x9.9 with respect to the performance of FSS for the lowest selectivity factor queries (respectively, 10^{-5}, 10^{-4} and 10^{-3}).

7. A Hybrid Solution: The Index Filtered Scan

The formulas and empirical tests performed raised some de ciencies on the FSS algorithm (the baseline algorithm in Hadoop). Indeed, when the selectivity factor is high enough, the HBase scan object does not lter any row out before sending data to MapReduce. Consequently, the fetch cost in this scenario is at its peak. After a thorough analysis of the results obtained, and in order to smooth its impact, we propose the Index Filtered Scan (IFS), which is an improved version of the FSS algorithm. In short, IFS exploits indexes, meant to be previously created, to identify the keys satisfying the selection predicates with-out accessing the data in the table. Thus, before sending data to MapReduce it will lter out those rows not meet-ing the selections in HBase. Note that IFS resembles the typical access used with bitmaps [8].

As previously discussed, HBase does not support any kind of secondary index natively. Thus, as proof of con-

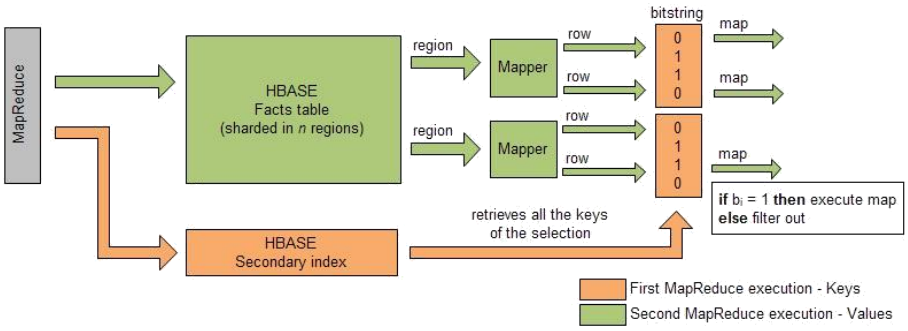

Figure 18: Index Filtered Scan (IFS)

cept, we simulated IFS as follows (see Figure 18). It rst uses the indexes but avoiding random accesses to the fact table (which would be costly for high selectivity factors). Alternatively, it aims at scanning the whole fact table. Thus, we use secondary indexes in a rst MapReduce job to nd out the fact keys. Then, we created an in-memory bitstring (which is created once and transparently dis-tributed to all nodes in the cluster) based on those keys obtained from the secondary index. In the second MapRe-duce job, the bitstring is checked in the mapper and only if the bit representing that row is enabled the map function is then executed. As in the other algorithms, we automat-ically group and aggregate the nal values by means of the MapReduce framework.

7.1. IFS Cost Formulas

In the spirit of our cost formulas, the IFS algorithm is characterized as follows. Since IFS, like IRA, performs a preliminary access to the secondary index, its execution cost function is composed by two MapReduce jobs (see Formula 10).

For the rst MapReduce, the very same explanations given in Section 4.2 also hold here.

The second phase of this algorithm consists in retriev-ing the right data from the fact table. Thus, the cost formula is similar to that of FSS. Note the subtle (and rel-evant) di erence. As selection predicates are computed by means of the secondary index there is no need to consider attributes used to lter (i.e., in the selection predicates) in the fetch cost. Hence, the #f variable only counts those families containing measures and dimensions (and not se-lection attributes), just like IRA does.

7.2. IFS Empirical Testing

In order to test IFS, we repeated the tests discussed in Section 5 for this new algorithm. However, even if the per-formance of IFS is rather close to that of FSS for large se-lectivity factors, it does not manage to beat FSS. The rea-son is the overhead introduced when simulating bitmaps. Unlike IRA, we were not able to exploit bitmaps at the HBase level and as consequence, the FSS fetch cost (i.e., cost of sending data from HBase to MapReduce) tends to be lower than in IFS because rows not matching the selec-tion predicates can be ltered out within HBase. Instead, IFS does the ltering at the MapReduce level. Figure 19 exempli es these di erences in the fetch cost. \P", \G" and \S" in this gure refer to projection, grouping and selection attributes, respectively.

Nevertheless, as said, IFS is still able to compete with FSS because under certain circumstances (i.e., with high selectivity factors), IFS is more e cient when dealing with selection predicates, specially in highly partitioned tables. Indeed, FSS sends all the query attributes (i.e., projection, grouping and selection attribute) to MapReduce. Op-positely, IFS does not need to retrieve slicer attributes and the mapper, checking the bitmap, only triggers the map function if the tuple ful lls the selection predicates of the query. FSS su ers from this drawback because the HBase scan object does not distinguish between lter-ing attributes (to select tuples) and the rest of attributes (data to be retrieved to build the cube). Ideally, the rst ones should be checked at the HBase level and not sent to

$$IFS = IRA_{index} + FSS_{table}^{0} + 2t_{MR} \quad (10)$$

MapReduce reducing, this way, the fetch cost. Nevertheless, the real gain would be implementing the

ltering bitmap natively inside HBase. This would allow us to lter tuples at the HBase level and do not even send them to MapReduce (see Figure 19). In this way, the IFS algorithm would clearly beat FSS in the presence of queries with several slicers and selective predicates.

8. Conclusions and Future Work

In this paper we have presented the impact of sec-ondary indexes and partitioning on Hadoop. To do so, we have described in detail two access plans, namely IRA (which exploits secondary indexes and random accesses) and FSS (the baseline algorithm typically used in Hadoop), in terms of cost formulas, as typically done in cost-based optimization in RDBMS. We then have devised a thorough testbed to validate our formulas by showing that (i) no rel-evant cost factor was omitted and (ii) their correctness to foresee the best access plan according to the cost factors identi ed.

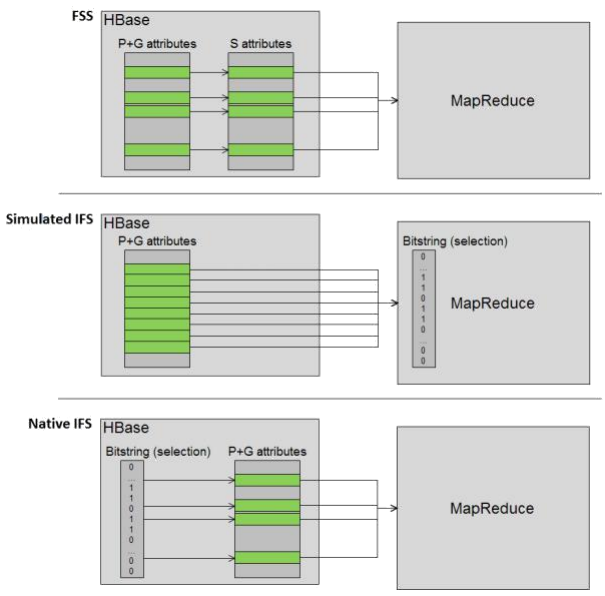

Figure 19: Di erences in the fetch cost for FSS and IFS

Although secondary indexes and partitioning are well-known tuning techniques for RDBMS they have been systematically ignored in distributed settings, where parallelism is massively exploited in the cloud and seen as the only alternative to improve performance. In this paper, we have shown how these techniques can help to drasti-cally improve the performance of OLAP queries to com-pute Small Analytics on Big Data by means of vertical fragmentation (i.e., the de nition of families in HBase) and the creation of secondary indexes.

8.1. The Impact of Partitioning and Indexing

On the one hand, we have shown the huge impact vertical partitioning strategies may have in HBase even if the HBase o cial documentation states that no more than three families should be de ned (see [29]). With our approach we have shown just the opposite, and in our tests partitioning in sixty families combined with se-quential reads have resulted in a much better performance. Nevertheless, when it comes to writing, using that many families resulted in a worse insertion performance because of the need to write in 60 di erent les (one per family). Our claim though is that there should not be a universal vertical partitioning strategy for HBase and it should de-pend on the kind of workload, the database size and the number of machines in the system. Indeed, like in a rela-tional DBMS, it is crucial to properly design the database according to its workload. On the other hand, secondary indexes resulted as e ective as in relational settings and the IRA algorithm systematically beat the FSS algorithm for low selectivity factors. All in all, we have shown the feasibility to characterize each access plan in terms of cost formulas, which foresee the need for a query optimizer in Hadoop/HBase.

8.2. Outlining Improvements for HBase

With the testbed carried out, we have also shown that HBase still su ers from several de ciencies that deserve further improvements. Firstly, we have shown that there is an important execution cost (fetch cost) due to the fact that the three main technologies in Hadoop are loosely coupled, which results in shipping data from HBase to MapReduce through the network. Secondly, HBase must develop native secondary indexes. The tests we conducted simulated indexes. However, these algorithms were in a clear disadvantage in front of the baseline algorithm, since two MapReduce jobs were needed. Ideally, the secondary indexes should be integrated in HBase as a primary structure. This way, the indexes would have their own namespace separated from tables and the temporal table created after processing the index in IRA (the second MapReduce job input) could have its own split policy (e.g., for cre-ating smaller regions) and boost the parallelism within MapReduce by enabling more mappers. Also, a native index would reduce the IFS execution to one MapReduce job and, in turn, its fetch cost[8]. Actually, as discussed, IFS should be seen as an improvements of FSS since the selection predicates would not be checked in MapReduce but in HBase.

All in all, the main nal conclusion is that Hadoop is still a relatively immature technology compared to RDBMS and there is much room for improvement. For ex-ample, by reconsidering well-known physical design tech-niques applied in RDBMS. However, a good database de-sign is not enough by itself and there is always a turn-ing point in which the next performance improvement can only be obtained by means of adding more nodes, which, in turn, should entail rethinking the database design to reach the optimal performance for this new number of ma-chines.

[28] 9.