Open access • Proceedings Article • DOI:10.1145/2064676.2064680

Building cubes with MapReduce — Source link

Alberto Abelló, Jaume Ferrarons, Oscar Romero

Institutions: Polytechnic University of Catalonia

Published on: 28 Oct 2011 - Data Warehousing and OLAP

Topics: Cloud computing, Data warehouse and Online analytical processing

Building Cubes with MapReduce

Alberto Abelló
Universitat Politècnica de Catalunya, BarcelonaTech
aabello@essi.upc.edu

Jaume Ferrarons
Universitat Politècnica de Catalunya, BarcelonaTech
jferrarons@essi.upc.edu

Oscar Romero
Universitat Politècnica de Catalunya, BarcelonaTech
oromero@essi.upc.edu

ABSTRACT

In the last years, the problems of using generic storage techniques for very specific applications has been detected and outlined. Thus, some alternatives to relational DBMSs (e.g., BigTable) are blooming. On the other hand, cloud computing is already a reality that helps to save money by eliminating the hardware as well as software fixed costs and just pay per use. Indeed, specific software tools to exploit a cloud are also here. The trend in this case is toward using tools based on the MapReduce paradigm developed by Google. In this paper, we explore the possibility of having data in a cloud by using BigTable to store the corporate historical data and MapReduce as an agile mechanism to deploy cubes in ad-hoc Data Marts. Our main contribution is the comparison of three different approaches to retrieve data cubes from BigTable by means of MapReduce and the definition of criteria to choose among them.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*data warehouse*; H.2.4 [**Database Management**]: Systems—*query processing, parallel databases*

General Terms

Algorithms, Management

Keywords

BigTable, Data Warehouse, Design, MapReduce, OLAP

1. INTRODUCTION

Nowadays, most companies externalize as many services as possible to reduce costs and be more flexible in front of fluctuations of the demand. Thus, with cloud computing, the time has arrived to IT infrastructures. The National Institute of Standards and Technology (NIST) defines *cloud computing* as "a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".

Cloud computing, in general, is a good solution for medium to small companies that cannot afford a huge initial investment in hardware together with an IT department to manage it. With this kind of technologies, they can pay per use, instead of provisioning for peak loads. Thus, only when the company grows up (if at all), so the expenses will. The only problem is that they have to trust their data to third parties.

In [1], we find an analysis of pros and cons of data management in a cloud. It is found completely inappropriate for transactional processing mainly due to the problems to guarantee ACID properties in such environment. However, it is adequate for analysis environments, since those properties are not needed. It also outlines the problem of having data in an untrusted environment, which would again be unacceptable in transactional processing, but can be easily solved in analytical systems by just leaving out some sensitive data or using an anonymization function. On the other hand, what cloud data management can offer to an analytical environment is elastic compute power (in the form of parallelism), replication of data (even across different regions of the world), and fault tolerance (a new machine automatically taking over from a fallen one without re-executing the whole query or process). As a side effect, we can also serve as many users as needed through the Internet.

Data Warehouses (DW) and On-Line Analytical Processing (OLAP) tools where defined by Bill Inmon in [12] and Edgar Codd in [6], respectively. Thus, they are almost twenty years old and have evolved to maturity by overcoming many limitations in these years. Huge (Terabytes) relational DW exist today benefiting from techniques like materialized views, bitmap indexes, etc [10]. Nevertheless, some challenges remain still open. Mainly, they are related to the management of ETL processes, unstructured data, and schema evolution.

Cloud computing does not mean that we cannot use a relational system. Indeed, well known alliances already exist in the market, like that between Oracle and Amazon. However, as pointed out in [15], there is four to five times as much unstructured data as there is structured data. NoSQL engines like BigTable (presented in [5]) are thought to store this kind of data. For example, with this approach, we could easily incorporate information extraction tools to the management of unstructured data in the sources. A non-formatted chunk of data would be stored associated to the key, and just when

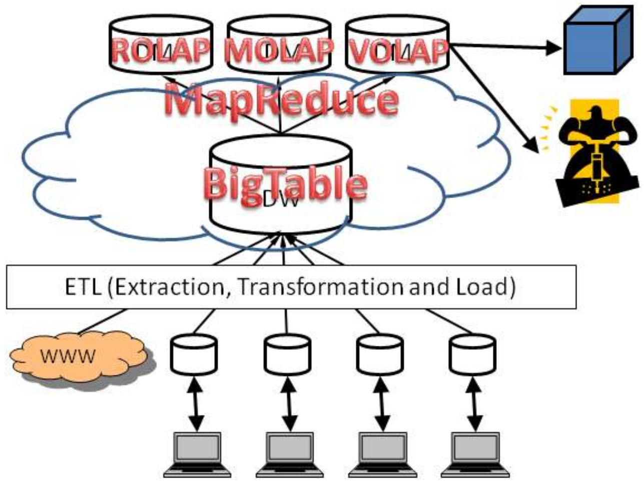

Figure 1: Proposed architecture

the user decides what to do with it and which tool can be used, we do it. Oppositely, in a RDBMS, we format and structure data in a star-shape schema months or even years before it is actually used (or even without knowing whether it will be used or not).

It is at this point that MapReduce (presented in [8]) can help, because it was conceived to parallelize the parsing and modification of data as it is being scanned. MapReduce is a framework that hides distribution of data, parallelization, fault-tolerance and load balancing from the programmer. It has been specially designed for scalability, and processing in a cloud. It allows to deal with huge volumes of data, when the schema is not concrete (i.e., variations can eventually appear).

At this point, it is important to clarify that we are assuming the DW architecture in [14], where we have the "Corporate Data Warehouse" (CDW) for generic storage and the "Data Marts" (DM) for the specific analysis of data. As pointed out in [13], the first one is not the addition of several *star schemas* and it does not even need to be multidimensional. Therefore, we can freely choose any DBMS (either relational or not) and data schema that suits our needs.

As explained in [22], MapReduce and Parallel (usually relational) databases are complementary in the sense that neither is good at what the other does well. Thus, as depicted in Figure 1, what we propose is to temporally store data in BigTable and just when we know exactly which is the analysis we have to perform, what are the data quality needs, and which is the more suitable technology (i.e., ROLAP, MOLAP, VOLAP, etc.), then we use the ETL to prepare the data according to our specific necessities. Indeed, our proposal is to store together all data related to a given subject (as stated in Inmon's DW definition), but without concreting the schema, so that it can easily evolve over time. We would just partially integrate and clean them before being stored in an (almost) unlimited CDW in a cloud. Then, on demand, we would extract specific data cubes specially customized for a given analysis, that would be stored in a concrete DM (either also in the cloud or not). For example, as explained in [11] in the presence of missing values, we may adopt several solutions: (i) just ignore the tuples with missing values, (ii) give them a default value (the same one for all tuples), either a global constant or the average of the attribute, (iii) give them the average of the class of the tuple, or (iv) guess the value for each tuple using regression, decision trees, etc. To make different decisions, different approaches may be more appropriate. A MapReduce tool would help to parallelize the ETL process and choose a different cleaning solution depending on the specific need of the time. Even if we would always want to choose option (iv), the more we wait to guess the value of each tuple, the more informed our choice can be.

Thus, we argue that when we want to benefit from the compute power of a cloud, the schema easily evolves, and the cleaning of data is also variable, using a MapReduce framework should be the right choice for the CDW. This would help to an agile deployment. However, the bad news is that since we would be paying per use, the computational cost of our ETL processes will be directly translated into money. Therefore, it is crucial to study and determine, a priori, which is the best (i.e., cheapest) way to process our data. Notice that trying would mean a payment to the cloud provider and we would rarely build the same cube twice (i.e., trial and error has to be clearly discarded). As pointed out in [9], one of the drawbacks of Hadoop is the lack of indexes. In this scenario, our contributions are (i) the algorithms implementing the three typical table access approaches to build a cube benefiting from MapReduce and BigTable technologies, and (ii) an experimental analysis of query characteristics to choose the most appropriate algorithm with regard to the cube we want to build.

The paper is organized as follows: Section 2 contains the related work; Section 3 introduces basic concepts about BigTable and MapReduce technologies; Section 4 explains three different options to build a data cube using MapReduce; Section 5 shows performance results and exemplifies when one of these options is preferable in front of the others; finally Section 6 concludes the paper.

2. RELATED WORK

In the last years, a significant trend has appeared against the "one size fits all" policy of RDBMS vendors (see [21]). On the one hand, most companies use only a small percentage of the functionalities of their RDBMS. On the other hand, using a DBMS devoted to one specific kind of applications that considers its special and distinguishing characteristics usually results in a really significant gain in performance for these applications.

For example, it is well known (see [7]) that for some kind of environments it would be much better to store data together per column instead of per row, like RDBMSs do. For DW, [2] shows that column storage clearly outperforms RDBMSs (showing the comparison with three possible simulations of columns in a RDBMS). Indeed, Dremel (developed by Google and presented in [18]) is an implementation based on column storage, and some products are already in the market like the open source MonetDB or Vertica, which is the commercial version of [20]. Nevertheless, we advocate that, while we can use column storage for the DM (where columns are well defined), we still need some more flexible storage for the CDW (whose data have not a clear structure or are under continuous evolution). Thus, for analytical purposes, Google also developed MapReduce and BigTable technologies and opened them to the community in 2004 and

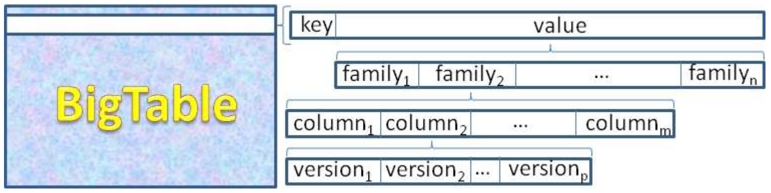

Figure 2: BigTable organization

2006, respectively. They can be considered young technologies whose power today is mainly based on the brute force provided by cloud computing.

Regarding the performance of MapReduce, we would name two relevant works: [19] and [16]. Firstly, [19] outlines the start-up cost of MapReduce and the high load throughputs compared to two commercial DBMSs at the leading edge of the market. The reason for the high load throughput is that it just stores data as it comes (which means faster loading time than any DBMS). Then, it is at query time we have to pay the price of formatting them (depending on the specific needs at the time). Moreover, MapReduce is able to recover from faults in the middle of query execution (which is quite common in big clouds) while other systems cannot. On the other hand, [16] shows that despite currently being slower that some DBMSs at query processing, MapReduce has a great margin for improvement by just incorporating well known techniques. They conclude that MapReduce-based systems are not inferior to parallel DBMSs in terms of performance. The problem is probably that they are not mature enough. For example, Hadoop (the most popular open source MapReduce implementation) did not release version 1.0, yet, and it does not benefit from any kind of indexing techniques, i.e., adding dynamic indexes to the framework is still an open issue (addressed in [9]). Summarizing, we could say that MapReduce is appropriate to serve long running queries in batch mode over raw data (which is precisely the case of ETL processes). A similar proposal to build cubes using MapReduce can be found in [25], but in this case the authors just rely on brute force to build the cubes and do not provide any algorithmic improvement beyond fragmenting data and distributing it among the nodes in the cloud.

Finally, [9] also shows that MapReduce and Relational algebra have the same expresiveness. Thus, some may argue that MapReduce has a too low level programming interface instead of a declarative language like SQL. Indeed, Hive project (see [23]) is working to solve this problem, by developing a SQL-like declarative language (i.e., HiveQL). However, it is again in its early stages and only provides a naive rule-based optimizer with a small number of simple rules. It is in this sense that our research is relevant, because offers different ways to evaluate a given kind of queries (i.e., multidimensional) based not on rules but on their selectivity factor and other physical characteristics.

3. GOOGLE CLOUD TECHNOLOGIES

To understand the rest of the paper, the reader may need some basic knowledge on the tools we are using. In this section we just summarize the main concepts of BigTable and MapReduce as they where presented by Google in [5] and [8], respectively.

3.1 BigTable

BigTable is a distributed storage system designed to scale to very large size (petabytes). Figure 2 sketches data organization inside it, whose main structure is [key,value] pairs. The main characteristics we would like to outline here are:

- Data are indexed by row and column values (which are arbitrary strings).
- Columns can be grouped into families to be physically stored together.
- Versions are automatically generated for each value (which are timestamped).
- Data are treated also as uninterpreted strings.
- Only single-row transactions are supported.
- Data is clustered (i.e., physically sorted) by key.

Note that only families of columns are part of the schema and have to be stated on creating a table. Oppositely, the columns are dynamically defined on inserting data. Moreover, since a family actually corresponds to a separate storage, adding them does not modify data already inserted (which means this is really efficient, order of seconds). Regarding data retrieval, it provides random access to one key, as well as parallel scan of the whole table or a range of keys.

3.2 MapReduce

MapReduce is a programming framework that allows to execute user code in a large cluster. It hides parallelization, data distribution, load balancing and fault tolerance from the user. All the user has to do is writing two functions: Map and Reduce. As sketched in Figure 3, those functions are injected in the framework.

Thus, the signature of those two functions is as follows:

$$map(key_{in}, val_{in}) \rightarrow \{[key^1_{tmp}, val^1_{tmp}], ..., [key^n_{tmp}, val^n_{tmp}]\}$$

$$reduce(key_{tmp}, \{val^1_{tmp}, ..., val^m_{tmp}\}) \rightarrow$$
$$\rightarrow \{[key^1_{out}, val^1_{out}], ..., [key^p_{out}, val^p_{out}]\}$$

The execution would be:

1. Map function is automatically invoked for each pair [key,value] in the source table (it also works for plain files, in which case, one pair is generated per line, having as key the position inside the file). Notice that the source table is distributed in a cloud. Therefore, the framework takes advantage of this and tries to execute each map call locally to the data. Each call can generate either one new pair (potentially different from that in the input), many pairs or none at all.

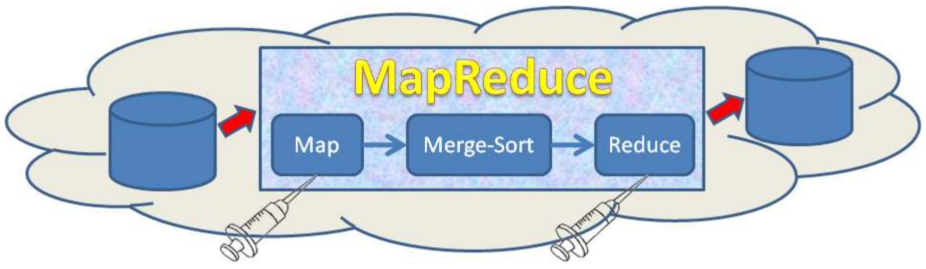

Figure 3: MapReduce overview

2. The intermediate pairs [key,value] generated by the different executions of the Map function, which are temporally stored in the distributed file system, are then ordered using a distributed merge-sort algorithm.

3. For each different intermediate key, the Reduce function is invoked once receiving together all values associated to it. Each call can generate again either one new pair (also potentially different from that in its input), many pairs or none at all.

4. BUILDING A CUBE

In this section we explain the three approaches we have used to build cubes. They correspond to the typical options relational optimizers take into account on accessing a table, namely "full scan", "index access", and "range index scan". The first one just uses HBase facilities to scan the whole source filtering it by the attributes the user indicates. The second builds indexes beforehand to easily obtain the identifiers of the desired tuples and then retrieve the data by random access. Finally, the last option mixes the other two (i.e., it implements indexes, retrieves the identifiers of the tuples, builds an in-memory bitmap, and uses it to filter the tuples in the map function, while scanning the whole table).

In all three cases, data is assumed to be completely denormalized in one universal relation containing all data related to the subject object of analysis. By doing it so, we incur in some extra space, but on the other hand, we avoid random access to the hypothetical dimensions and having to join them with the factual data; and on the other hand, we store the snapshot of the dimensional data at the time the fact occurred (i.e., facilitating the tracking of slowly changing dimensions). MapReduce configuration details have been mostly omitted in the algorithms, since default values are being used. Moreover, without loss of generality, we assume that the aggregation function is always "SUM". Also for the sake of simplicity in the pseudo-code provided, we consider that there is only one measure in the output cube (otherwise, a list of measures should be provided in the input, instead of only one; and a nested loop over an array implemented in the last reducer to aggregate each one of them separately).

4.1 Full Source Scan (FSS)

The idea behind this option is using just the brute force of parallelism in a cloud. It has only been slightly improved by using the filter facilities of BigTable. However, since this filtering is not performed by the key of the table, but by values in the different columns, it does not result in any significant improvement in the overall performance, because as explained before indexing is not implemented in BigTable.

BigTable scan configuration as well as Map and Reduce functions for this approach are sketched in Algorithm 1. Firstly, we state that only those columns of interest for the final cube (i.e., dimensions, measure, and slicers) must be retrieved from the source (Line 2). Then, we state that those pairs [key,value] whose columns do not match the condition must be filtered out (Line 3). The map function in this case just redefines the key as the dimensions of the output cube, and the value as the measure to be aggregated (Line 7). Finally, after all pairs [key,value] generated by the Map function have been transparently sorted and grouped by key, the reduce function is called once for each combination of the dimensions' values. Thus, the reducer only has to mirror the input key in the output (Line 10), and aggregate the measure's values corresponding to that cell (Line 12). Note that val_{tmp} is a set of all values in the different pairs sharing the same dimension coordinates.

4.2 Indexed Random Access (IRA)

As explained in [17], although analytical queries usually apply aggregation techniques over non-very-selective rows, they may exhibit quite selective predicates. For example, the use of bitmaps (a common indexing technique in DW) is usually worth only when the overall selectivity factor of the query is below 1% (for example, regarding all queries in [24] the median of the selectivity factor is 0.8%, with a range of values between 0.00000003% and 99.9%). Thus, the idea behind this approach is to use some kind on indexing technique to avoid a full scan of the source. Therefore, we introduce a phase "MapReduce0" that aims at building such indexing structure. We assume this is done before hand, since it can be reused to build many cubes if incrementally maintained to reflect the successive updates of the CDW.

The three phases of this approach are sketched in Algorithm 2. The preliminary MapReduce job builds the indexing structure by just scanning the whole source table. Since only those columns involved in the aggregation hierarchy need to be retrieved, scan is configured so (Line 2). Afterwards, the map function just puts in the temporal key the retrieved value (Line 6) with the following format: the schema and values of the hierarchy from top to bottom, preceded by the name of the dimension (e.g, "`Time;year:2005;month:December;day:7`"). By doing so, we benefit from the locality of BigTable due to [key,value] pairs being indexed and clustered by key (e.g., all days of Dec. 2005 will be stored together, and later on retrieved in a single access by just stating the range "[Dec. 2005, Jan.

Algorithm 1 Filtered Source Scan

```
Input:
    sourceName, M: String; //Name of the source BigTable and Measure
    D, S: Set of String; //Dimensions, Slicers
Output: A data cube
 1: function Config does //Configure the source BigTable scan
 2:   scan.addColumns($D \cup \{M\} \cup S$);
 3:   scan.addFilter($S$);
 4:   initTableMapperJob(sourceName, scan, Mapper, Text, Text, job); //Call stating table and scan procedure
 5: end function
 6: function Mapper does
 7:   $key_{tmp}$:= $val_{in}$[D]; $val_{tmp}$:= $val_{in}$[M];
 8: end function
 9: function Reducer does
10:   $key_{out}$:=$key_{tmp}$;
11:   for each String $v$ in $val_{tmp}$ does
12:     $val_{out}$+=(float)$v$;
13:   end for
14: end function
```

2006)"). The output value in this case is just the key in the source table, i.e., the ID of the pair (Line 6). The reducer will, after all pairs being sorted and grouped by the dimension values, just put in the output pairs where the key is the dimension value (Line 9), and the value is a list of IDs in the source corresponding to that dimension value (Line 11).

Eventually, when a cube needs to be actually built, we will execute the other two phases. In the first phase, based on the slicers (provided in a plain file in Line 15), the indexing structure is randomly accessed (Line 18). The map function just generates one new temporal pair for each ID in the input value (Line 20). Note that the value of those pairs is always "1", which represents the number of slicers where we found the ID. After the sort step, all we need to do in the reducer is check whether the ID was present in all slicers or not. If so, we put it in the output (Line 29), because it belongs to the intersection of all the slicers and the corresponding measure deserves to be considered in the output cube. Note that we are assuming that all slicers of the same dimension are disjoint (if more than one) and that slicers of different dimensions have been *and*ed in the predicate. Otherwise, more complex comparison that just counting should be implemented at this point to check whether the corresponding data in the source table makes the slicing predicate to evaluate true or false. Finally, the second and last phase just scans the output BigTable of the previous phase and randomly access the source table for each ID found (Line 36). From this on, it acts with the retrieved pair as in FSS (Section 4.1).

4.3 Index Filtered Scan (IFS)

This third option, sketched in Algorithm 3, is a mix of the other two. The aim is to use the index but avoid random access to the source (which would be costly for high selectivity factors). Thus, we only scan from minimum to maximum keys and those disk blocks without any key will be skipped. To do so, we use the same preliminary and first phase exactly as IRA (Section 4.2). Afterwards, we configure the scan of the source BigTable and create an in-memory bitmap based on those IDs in the intermediate BigTable generated by MapReduce1 (Line 4). The bitmap is used in the mapper step to filter the pairs generated (Line 10). This is done so, because given the number of IDs generated, it results much more efficient than using the filter facilities provided by BigTable (see Section 5). Finally, we use Reducer2 as the one introduced for FSS and IRA (Sections 4.1 and 4.2).

This algorithm results in less disk accesses than the others but needs to build an in-memory (bitmap) structure to store the desired keys.

5. IMPLEMENTATION AND TESTING

We have implemented all these algorithms on Hadoop (see [3]), which is Apache's open source implementation of MapReduce. Moreover, test data as well as indexing information has been stored in HBase (see [4]), which is Apache's open source implementation of BigTable. Experiments were run on a single machine, Core2 Duo (2.2GHz), 4Gb RAM and HD SATA 5400rpm. We did not use a cluster to avoid its management complexity and eliminate, by now, the number of active nodes in the experimental variables.

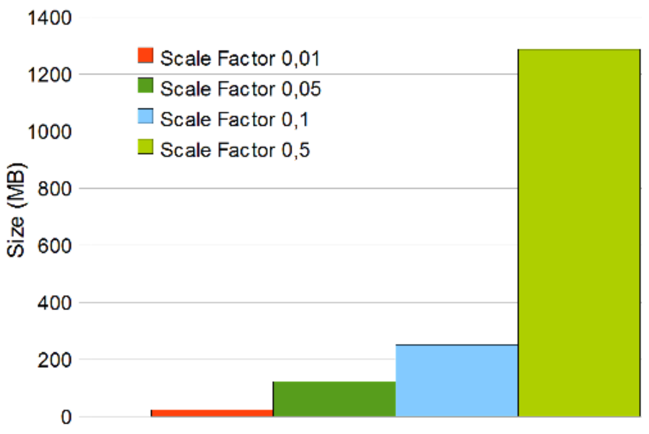

Figure 4: Source table space

One big denormalized table has been filled with data following the TPC-H database population specification (as stated in [24]). Compression has been enabled and one family of columns has been defined for each TPC-H column (this improves the compression rate given that physically consecutive repetitions of values are much more likely). Figure 4 shows, for each Scale Factor (SF from here on) the size of

Algorithm 2 Indexed Random Access

```
Input:
    sourceName, indexName, slicerName: String; //Name of source and index BigTable, and file containing the slicers
    D: Set of String; //Dimensions
    H: Ordered list of String; //Slicer dimension hierarchies
    M: String; //Measure
Output: A data cube
 1: function Config0 does //Configure the source BigTable scan
 2:   scan.addColumns(H);
 3:   initTableMapperJob(sourceName, scan, Mapper0, Text, Text, job); //Call stating table and scan procedure
 4: end function
 5: function Mapper0 does
 6:   $key_{tmp}$:= format($val_{in}$); $val_{tmp}$:= $key_{in}$;
 7: end function
 8: function Reducer0 does
 9:   $key_{out}$:=$key_{tmp}$;
10:   for each String $v$ in $val_{tmp}$ does
11:     $val_{out}$=$val_{out}$.concat($v$);
12:   end for
13: end function
14: function Config1 does //Configure the load of the slicers
15:   setInputPaths(job, slicerName); //Call stating input file for the mapper
16: end function
17: function Mapper1 does
18:   currentVal=BigTable.get(indexName,$key_{in}$);
19:   for each String $v$ in currentVal does
20:     output [$v$,1];
21:   end for
22: end function
23: function Reducer1 does
24:   $key_{out}$:=$key_{tmp}$;
25:   for each String $v$ in $val_{tmp}$ does
26:     counter+=(int)$v$;
27:   end for
28:   if $counter =\mid file(slicerName) \mid$ then
29:     return [$key_{out}$,null];
30:   end if
31: end function
32: function Config2 does //Configure the temporal BigTable scan
33:   initTableMapperJob(temporalName, scan, Mapper2, Text, Text, job); //Call stating table and scan procedure
34: end function
35: function Mapper2 does
36:   currentVal=BigTable.get(sourceName,$key_{in}$, $D \cup M$);
37:   $key_{tmp}$:= currentVal[D]; $val_{tmp}$:= currentVal[M];
38: end function
39: function Reducer2 does
40:   $key_{out}$:=$key_{tmp}$;
41:   for each String $v$ in $val_{tmp}$ does
42:     $val_{out}$+=(float)$v$;
43:   end for
44: end function
```

the source table once denormalized and loaded into HBase. Given that $SF = x$ means that TPC-H generates xGb, we need approximately double space for our denormalized table. As depicted in Figure 5, indexes generated by IRA and IFS need approximately 10% of the size of the source table, independently of the SF.

On the other hand, query configuration has been also decided based on that of TPC-H queries, which is summarized in Table 1, and taking the maximum size (i.e., $SF = 0.5$). For each one of these four characteristics of the queries (i.e., selectivity factor, number of attributes in the select clause, number of attributes in the group by, and number of clauses in the selection predicate, aka slicers), we have generated a set of queries to measure their independent influence on the performance. Each set contains queries varying exactly one factor from minimum to maximum, and fixing the other three factors to the median. Note that we use the median instead of the average, because it is a better indicator of central tendency in the presence of skewed distributions and outliers (which is the case in the TPC-H queries).

	Min	**Median**	**Max**
Selectivity factor	0.00000003%	0.8%	99.9%
Projected attributes	1	3	9
Grouping attributes	0	1	7
Predicate attributes	1	2	6

Table 1: TPC-H query statistics

Figure 6, which is in logarithmic scale for vertical as well as horizontal axis, shows the results (in seconds) for varying the selectivity factor. For IRA and IFS, we launch two MapReduce processes, in front of only one for FSS. Thus, given that Hadoop has high start-up cost (as recognized in

Algorithm 3 Indexed Partial Scan

```
Input:
    sourceName, indexName, slicerName: String; //Name of source and index BigTable, and file containing the slicers
    D: Set of String; //Dimensions
    H: Ordered list of String; //Slicer dimension hierarchies
    M: String; //Measure
Output: A data cube
 1: Run MapReduce0 and MapReduce1 as in Algorithm 2;
 2: function Config2 does //Configure the source BigTable scan
 3:     for each int $k$ in temporalBigTable does
 4:         bitmap[$k$]=true;
 5:     end for
 6:     scan.addColumns($D \cup M$);
 7:     initTableMapperJob(sourceName, scan, Mapper2, Text, Text, job); //Call stating table and scan procedure
 8: end function
 9: function Mapper2 does
10:     if $bitmap[key_{in}]$ then
11:         $key_{tmp}$:= $val_{in}$[D]; $val_{tmp}$:= $val_{in}$[M];
12:     end if
13: end function
14: function Reducer2 does
15:     $key_{out}$:=$key_{tmp}$;
16:     for each String $v$ in $val_{tmp}$ does
17:         $val_{out}$+=(float)$v$;
18:     end for
19: end function
```

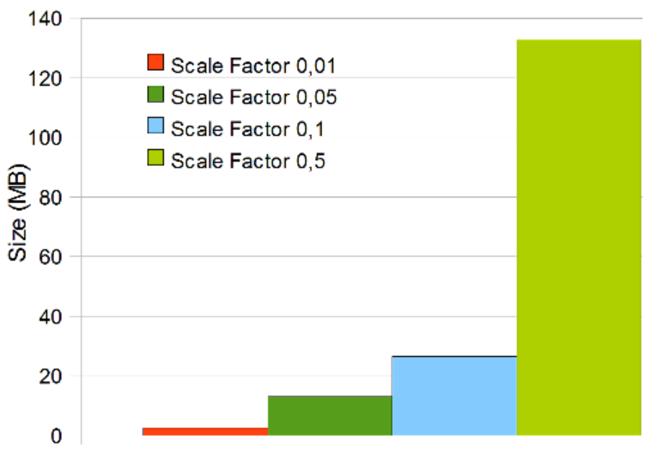

Figure 5: Index space

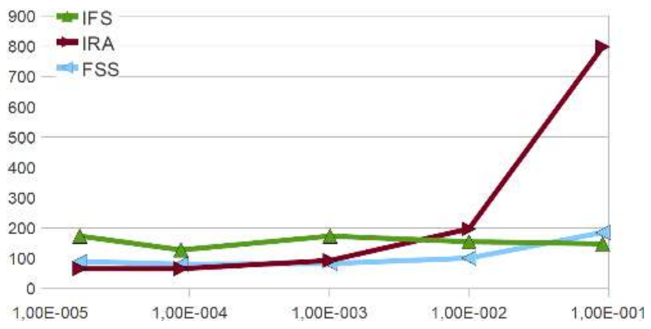

Figure 6: Performance by selectivity factor

[19]), we need a fair volume of data to compensate this. Nevertheless, IFS always compensates it for high selectivity factors, even for a small SF. As SF is increased, this handicap is compensated for lower and lower selectivities. Regarding IRA, we can observe that with just $SF = 0.5$ it is already the best option for the lowest selectivity.

On the other hand, Figure 7 shows the variation of the execution time (in seconds) depending on the number of slicers. We can observe that it also affects the performance of IRA and IFS. This happens because they have to manage one more index access per slicer.

Figures 8 and 9 show the results for varying the projected attributes and those in the group by. We can observe that, given a SF, the corresponding three lines are mainly parallel, which means that these characteristics of queries do not affect the relative performance of the algorithms. Thus, we can conclude that the dominant characteristic is clearly the selectivity factor which makes performance grow exponentially, while the other makes it grow linearly.

6. CONCLUSIONS

In this paper we have shown how to benefit from cloud computing technologies to build OLAP cubes by using MapReduce and BigTable. Specifically, three different algorithms have been proposed and empirically compared, based on TPC-H benchmarking schema, data generator and query patterns. Our experiments show that the dominant performance factor is the selectivity of the queries. However, also the number of slicers affects the performance of those algorithms benefiting from indexes. The number of attributes being projected and used for grouping results to be irrelevant for the performance. To simplify the analysis and avoid the complexity of using a cluster we left as future work the variation in the number of nodes.

7. ACKNOWLEDGEMENTS

This work has been partly supported by the Spanish Ministerio de Ciencia e Innovación under project TIN2008-03863.

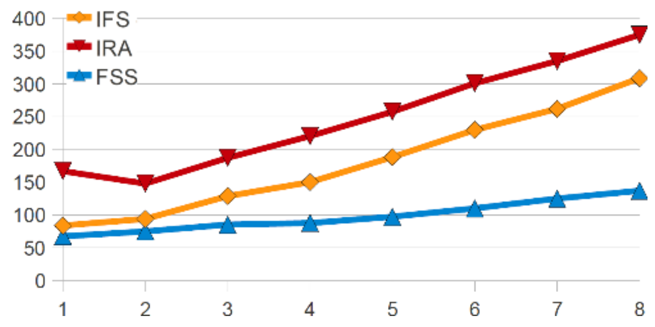

Figure 7: Performance by number of slicers

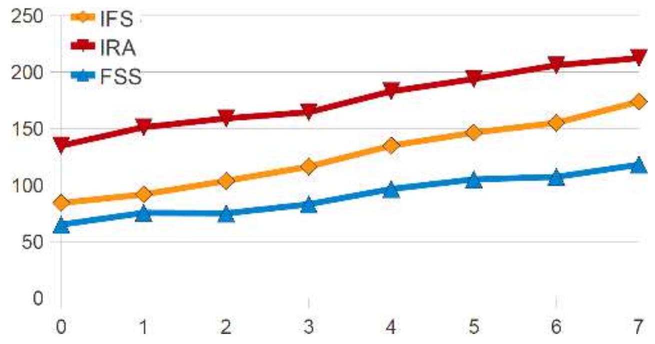

Figure 9: Performance by group by size

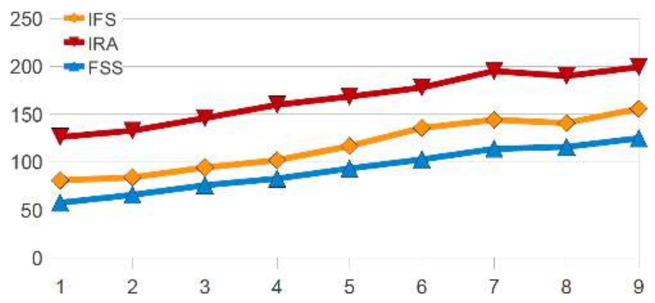

Figure 8: Performance by projection size

8. REFERENCES

[1] D. J. Abadi. Data management in the cloud: Limitations and opportunities. *IEEE Data Engineering Bulletin*, 32(1):3–12, 2009.

[2] D. J. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD Int. Conf. on Management of Data*, pages 967–980, 2008.

[3] Apache. Hadoop, http://hadoop.apache.org/.

[4] Apache. HBase, http://hbase.apache.org/.

[5] F. Chang et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2), 2008.

[6] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP to user-analysts: An IT mandate. Technical report, E. F. Codd & Associates, 1993.

[7] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *SIGMOD Int. Conf. on Management of Data*, pages 268–279. ACM Press, 1985.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[9] J. Dittrich et al. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proceedings of the VLDB Endowment (PVLDB)*, 3(1):518–529, 2010.

[10] M. Golfarelli and S. Rizzi. *Data Warehouse Design. Modern Principles and Methodologies.* McGraw-Hill, 2009.

[11] J. Han and M. Kamber. *Data Mining: Tools and Techniques.* Morgan Kaufmann, 2006.

[12] W. Inmon. *Building the Data Warehouse.* John Wiley & Sons, Inc., 1992.

[13] W. Inmon. Data Mart Does Not Equal Data Warehouse. *DM Review magazine*, May 1998.

[14] W. Inmon, C. Imhoff, and R. Sousa. *Corporate Information Factory.* John Wiley & Sons, 1998.

[15] W. Inmon, D. Strauss, and G. Neushloss. *DW2.0.* Morgan Kaufmann, 2008.

[16] D. Jiang et al. The performance of mapreduce: An in-depth study. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1):472–483, 2010.

[17] W. Lehner. Query Processing in Data Warehouses. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 2297–2301. Springer, 2009.

[18] S. Melnik et al. Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1):330–339, 2010.

[19] A. Pavlo et al. A comparison of approaches to large-scale data analysis. In *SIGMOD Int. Conf. on Management of Data*, pages 165–178. ACM, 2009.

[20] M. Stonebraker et al. C-Store: A Column-oriented DBMS. In *31st Int. Conf. on Very Large Data Bases (VLDB)*, pages 553–564. ACM, 2005.

[21] M. Stonebraker et al. The end of an architectural era (it's time for a complete rewrite). In *33st Int. Conf. on Very Large Data Bases (VLDB)*, pages 1150–1160, 2007.

[22] M. Stonebraker et al. MapReduce and parallel DBMSs: friends or foes? *Communication of ACM*, 53(1):64–71, 2010.

[23] A. Thusoo et al. Hive - a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment (PVLDB)*, 2(2):1626–1629, 2009.

[24] Transaction Processing Performance Council. Decision suport benchmark (TPC-H), http://www.tpc.org/tpch.

[25] J. You, J. Xi, C. Zhang, and G. Guo. HDW: A High Performance Large Scale Data Warehouse. In *International Multi-Symposium of Computer and Computational Sciences (IMSCCS)*, pages 200–202. IEEE, 2008.