Optimizing Druid with Roaring bitmaps

Samy Chambi
Computer Science, UQAM
Montreal, QC, Canada
chambi.samy@gmail.com

Daniel Lemire
LICEF, TELUQ
Montreal, QC, Canada
lemire@gmail.com

Robert Godin
Computer Science, UQAM
Montreal, QC, Canada
godin.robert@uqam.ca

Kamel Boukhalfa
Computer Science, USTHB
Algiers, Algeria
boukhalk@gmail.com

Charles R Allen
Metamarkets
San Francisco, CA, USA
charles@allen-net.com

Fangjin Yang
Metamarkets
San Francisco, CA, USA
fangjinyang@gmail.com

ABSTRACT

In the current *Big Data* era, systems for collecting, storing and efficiently exploiting huge amounts of data are continually introduced, such as Hadoop, Apache Spark, Dremel, etc. *Druid* is one of theses systems especially designed to manage such data quantities, and allows to perform detailed real-time analysis on terabytes of data within sub-second latencies. One of the important *Druid*'s requirements is fast data filtering. To insure that, *Druid* makes an extensive use of bitmap indexes. Previously, we introduced a new compressed bitmap index scheme called *Roaring bitmap* that has shown interesting results when compared to the bitmap compression scheme adopted by *Druid*: *Concise*. Since, *Roaring bitmap* has been integrated to *Druid* as an indexing solution. In this work, we produce an extensive series of experiments in order to compare *Roaring bitmap* and *Concise* time-space performances when used to accelerate *Druid*'s OLAP queries and other kinds of operations *Druid* realizes on bitmaps, like: retrieving set bits from bitmaps, computing bitmap complements, aggregating several bitmaps with logical ORs and ANDs operations. *Roaring bitmap* has shown to improve up to $\approx 5\times$ analytical queries response times under *Druid* compared to *Concise*.

CCS Concepts

•Information systems → Database management system engines; Database query processing; *Data structures;* **•Theory of computation → Data compression;**

Keywords: Bitmap indexes, compression, OLAP, performance.

1. INTRODUCTION

Nowadays, the massive generation of *Big Data* that can come from various sources: organizations, peripheral devices, individuals, etc., instigate companies and puts them in front of a challenge to develop solutions that allow to effectively collect and organize such masses of data, with the aim of extracting new knowledge which, in the end, will play a major role in the company competitiveness. Several solutions of this kind have been introduced by famous companies, such as: Hadoop [12] of Yahoo, PowerDrill [9] and Dremel [10] of Google, Avatara [16] of LinkedIn, [11] of Twitter, etc. *Druid* [17] is one among these systems. It was proposed recently by the *Druid* community and allows to store and analyze in real-time important quantities of data within low latencies. Previous tests have shown that *Druid* was able to scan, filter and aggregate 1 billion of data points in only few milliseconds [13]. Currently, this system is deployed in production by several societies, in particular: Netflix, Yahoo, Alibaba, etc. Besides its colossal computational capacities, *Druid* is also proposed as an open source project for the general public, and leans on a vast community of developers.

Druid's use case is really around powering user-facing analytic applications, contexts where query performance is very important. Among the solutions adopted by *Druid* to improve analytical queries performances are compressed bitmap indexes. Effectively, *Druid* makes an extensive use of this type of indexes to filter data very fast when executing drill-down OLAP queries by performing several operations on bitmaps, like logical ORs and ANDs, bitmap complements, bitmap scans, etc.

Today, several bitmap compression schemes have been proposed, to cite a few: WAH [15], Concise [4], PLWAH [6], SPLWAH [2], SECOMPAX and PLWAH+ [3]. These models and the majority of the bitmap compression schemes introduced these last fifteen years are based on a same hybrid coding that combines a run-length encoding with bit strings aligned by CPU words. Recently, we have proposed a new bitmap compression model, called *Roaring bitmap* [1]. This scheme represents a bitmap as a list of 32-bit integers sorted in ascending order. Each element of the list corresponds to one set bit position of the represented bitmap. To efficiently store such a list of integers, *Roaring bitmap* divides the elements of the list to chunks of 2^{16} integers, where each chunk groups integers sharing the same 16 most significant bits. Chunks are represented with a sorted dynamic array, where each of its entries stores the common 16 most significant bits of the group of integers falling within that chunk, and a pointer to a container which keeps the 16 least significant bits of the group of integers. A container can take one of two

IDEAS '16 July 11-13, 2016, Montreal, QC, Canada

ACM ISBN 978-1-4503-4118-9/16/07.

DOI: http://dx.doi.org/10.1145/2938503.2938515

forms: an uncompressed bitmap or a sorted dynamic array. A specific form is selected corresponding to the number of integers contained within a chunk.

This new compressed bitmap model allows to perform random accesses within a bitmap very fast, by performing at most two binary searches within the structure in a logarithmic time, compared to the linear time needed by the previous models based on run-length encoding. Also, keeping integers within containers indexed by a sorted dynamic array generally leads to combine two bitmaps very fast as it allows to skip the treatments of many integers stored inside non accessed containers. The *Roaring bitmap* scheme has been implemented with the Java programming language and is currently shared as an open source project.[1]

Druid uses bitmap indexes represented with the *Concise* [4] scheme. Benchmarks realized during the *Roaring bitmap*'s implementation project have shown that this scheme has been able to significantly improve many types of operations realized on bitmaps comparing to *Concise*. After introducing *Roaring bitmap* to the literature, it came time to validate the format of the bitmap compression scheme by trying to integrate it within a real database management system (DBMS) to observe and analyze the possible advantages and disadvantages brought by the bitmap compression technique to the global DBMS' performances. After a survey over the literature of such systems, *Druid* was chosen as it has been written in Java and it greatly depends on bitmap indexes to improve OLAP queries running times.

Druid's instances use by default a storage engine that adopts in-memory mapped structures to store and read data manipulated by the system. However, a heap based storage engine can also be configured. In order to make *Concise* operational on this context, an extension of the original library that could only deal with bitmaps residing entirely in main memory was produced by *Druid* developers to make it able to serialize bitmaps on disk and access them with memory-mapping. The source code of the developed library is freely available online.[2]

Having obtained interesting performances with *Roaring bitmap* when compared to *Concise* during previous experiments realized on in-memory bitmaps, the *Roaring bitmap* library was extended to support the management of memory-mapped bitmaps. Experiments comparing time-space performances of *Roaring bitmap* and *Concise* in such a context have followed. Results revealed impressive performances for the benefit of *Roaring bitmap*. The source code of these experiments is shared online.[3] This work was proposed to the community behind *Druid*, and a close collaboration with the *Druid* community followed to integrate *Roaring bitmap* as an indexing solution within *Druid*. Afterward, experiences under *Druid* were led to compare the performances of *Roaring bitmap* to those of the existing bitmap compression scheme: *Concise*. Results were decisive for *Roaring bitmap*. Indeed, *Roaring bitmap* improved up to 5× the execution times of OLAP drill-down queries compared to *Concise*. To simplify the reading of the text, the term *Roaring* will refer to the *Roaring bitmap* scheme in the remainder of the paper.

Other systems dedicated to massive data processing have also integrated *Roaring* as an indexing solution, like: Apache Spark [18], Kylin[4], Solr[5], Elastic [8] and Lucene[6], the latter using an independent implementation of *Roaring*. We leave the analysis of *Roaring*'s performances under these systems for future works.

This paper is organized as follows. First, a presentation of the *Druid*'s system is given in section 2. *Druid*'s OLAP queries that exploit bitmaps to accelerate response times are presented in section 3. Section 4 introduces the benchmarks done under *Druid* to compare *Roaring* and *Concise*'s performances. Benchmark results and an analysis follows. Section 5 presents the conclusion of the work.

2. DRUID

Druid was open sourced by developers from *Metamarkets*[7] in 2012. This system is a data store that guarantees a high availability, even in super-concurrent contexts (1000 users and more), and fast running times for analytical queries (sub-second latencies). One of the *Druid*'s challenges is to allow users to make decisions based on up to date data, by ingesting newly received events in real-time, within sub-second latencies. Currently, *Druid* is adopted in production by several companies[8], such: eBay, Yahoo, Netflix, etc. Furthermore, *Druid* stores data in a columnar format, adopts a distributed and a shared-nothing architecture, and makes a massive use of compressed bitmap indexes to accelerate OLAP queries as they often require lots of data filtering. This system was also designed to be fault-tolerant and to be able to support fast aggregations and flexible filtering techniques.

Druid stores events data that are never modified and read-only exposed to the users. The data format consists of three different components:

A timestamp: the first component represents the date of a given event.

Dimensions: the second component represents all the attributes used by OLAP queries to filter data (drill-down queries). So, these attributes play the same role as the dimensions in a traditional data warehouse.

Metrics: the third component represents all the attributes playing the same role of the measures in a traditional data warehouse. These columns contain numerical values and are used by OLAP queries to make aggregations and calculations, as with the SQL functions, COUNT, SUM, AVG, etc.

Druid stores an event in a data source. The latter is similar to the table concept in a relational database. Furthermore, the events of a data source are divided according to their timestamp attribute into segments. Segments data are stored in columnar format and dimension columns are indexed with bitmaps to quickly resolve filters. A segment stores in general $5-10$ million events that fall within a precise time interval: minute, hour, day, etc. This interval is specified in the initial configuration parameters of a *Druid* cluster. The length of this interval is the same for all segments of a data source. The smallest possible granularity

[1] http://roaringbitmap.org/
[2] https://github.com/metamx/extendedset
[3] https://github.com/samytto/MemoryMappedBitmaps
[4] http://kylin.apache.org/
[5] http://lucene.apache.org/solr/
[6] http://lucene.apache.org/core/
[7] https://metamarkets.com/
[8] http://druid.io/druid-powered.html

for the timestamp attribute kept inside segments is millisecond, but segments events can also be accumulated into a larger granularity, for example: minute, hour, a day. However, this granularity must necessarily be less or equivalent to the segments granularity. Also, segments data associated to a specific data source have all the same granularity.

A *Druid* cluster instance is made up of several types of nodes, each designed to make a very precise work. In a distributed architecture, one node runs per machine in a totally independent way with regard to the other nodes in the cluster (unless a standalone execution mode is adopted). So, nodes share neither data nor material resources with other nodes in a cluster (notion of shared-nothing architecture [14]). The various types of nodes supported by a *Druid* cluster and their important roles are presented in what follows:

Historical node: takes care of loading/deleting segments maintained in its local space, and executing queries on them.

Coordinator node: its important task resides in distributing segments between historical nodes among the cluster.

Broker node: represents the entity which users send their requests to. These nodes know how segments are distributed among the cluster. So, once a query is received by one of these nodes, it is directly sent towards all nodes susceptible to serve segments with valid results for the query. Also, this type of nodes receives results of its delivered queries and merges them, before sending the final result to the users.

Real-time node: represents the entry point of the real-time captured events. This type of nodes is responsible of building segments and releasing them to historical nodes after a certain configurable deadline. As it also answers requests searching about recent data it still serves.

Overlord and HadoopDruidIndexer nodes: entry points for batch inserted data. These nodes use the Hadoop framework to parallelize ingestion tasks and to reduce latencies.

For more information about the roles of the various nodes, the reader can refer to the *Druid*'s white paper [17].

To execute a request, the user sends it to one of the operational broker nodes in the cluster, which will distribute the query towards the various historical and/or real-time nodes selected for that query. A real-time node executes the request on the recent data still maintained within its local space, and a historical node executes it on the segments it stores locally. After this step, each requested node returns back its calculated results to the broker node from which the query was sent. OLAP queries often need to perform fast data filtering, and to improve the processing times of these operations, *Druid* adopts a bitmap index for every dimension attribute figuring inside a segment. Bitmaps turn out to be very efficient regarding the acceleration of drill-down operations that filters data according to dimension values.

3. DRUID'S ANALYTICAL QUERIES

Druid supports various types of analytical queries. They are expressed in JSON and sent by POST requests to candidate nodes. These requests can be divided into two great classes: search and aggregation queries, and metadata queries. Given that only the first class of queries can make use of bitmaps during execution, we will only study that class in this work. A particularity of the first group of queries is that a time interval is always specified in a request body. This time interval indicates the time partition in which fits the data targeted by the query.

This section gives a brief description of the Druid query types figuring in the first class of queries. The reader interested to know more about a specific type of queries can refer to the documentation published online [7], that gives further detailed information with query examples.

3.1 The GroupBy query

A GroupBy query has the same objective as that defined in the SQL language, which is generally to return metric values aggregated by distinct dimension values. The syntax of the *Druid*'s GroupBy query is presented below:

:

```
{
  "queryType": "groupBy",
  "dataSource": <datasource name>,
  "granularity": <granularity value>,
  "dimensions": [<dimension1>, <dimension2>, etc.],
  "filter": {<filter>},
  "aggregations": [
    { <aggregator1> }, { <aggregator2> }, etc.
  ],
  "postAggregations": [
    {<postAggregator1>}, {<postAggregator2>},   etc
          .
  ],
  "intervals": [<query interval>],
 "limitSpec": {         "type": "default",
 "limit": <limit's integer value>,
 "columns": [{OrderByColumnSpec1}, {
     OrderByColumnSpec2}, etc.]
        },
  "having": {
    "type": "having clause type",
        <Having clause's remaining fields>
  },
        "context": <context's properties>
}
```

A GroupBy query header consists of the "queryType" attribute, which indicates the type of the formulated query, followed by the "dataSource" field, that represents the data source on which the request will be executed. The next field "granularity" specifies the granularity to which final results would be accumulated. Within the "dimension" field, are declared the dimensions on which groupings would be made. For example, if two dimensions were specified, "Country" that possesses n distinct values and "province" that counts m different values, then the result would be formed of $n \times m$ different groups presented by day aggregations (if "granularity" equals to "day"). The following field allows to specify filters to use to drill-down more detailed analysis on the requested data. This part plays the same role as the WHERE clause for an SQL query. Here's an example of such a filter:

```
" filter ": {
" type ": " and ",
" fields ": [
{ " type ": " selector ", " dimension ": "job ", "
    value ": " student " },
```

```
{ " type ": "or",
" fields ": [
{ " type ": " selector ", " dimension ": " city ",
    " value ": " Montreal " },
{ " type ": " selector ", " dimension ": " city ",
    " value ": " Toronto " }
]}]}
```

Filters are declared, generally, over dimension attributes, but they remain also applicable on metric ones. To quickly reach the data subset targeted by a filter, the bitmaps associated with the dimension values specified in the filter come into play, and several types of operations could be performed on bitmaps, such: logical ORs, ANDs, or NOTs, bitmap scans, etc. In this example, a logical OR is executed between bitmaps associated with both dimension values, "Montreal" and "Toronto", then one logical AND is computed between the resulted bitmap from the previous step and that of the "student" dimension value. These logical operations are executed on every segment reached in a data source.

Afterward, comes the part specifying the metric attributes, on which aggregations would be computed. One of the few operations supported by OLAP engines is the one specified by the "postAggregations" field. This one allows to generate new metric values in the result by performing operations on the aggregated values specified in the "aggregations" field.

The next field "intervals" is used to define the time interval the requested data fits in. This value helps the OLAP engine to access only segments falling in this time interval. The next field, "limitSpec", allows to specify a limit on the number of returned results and an attribute to sort the resultant elements with. The last field is "Having", whose function remains similar to that of the "HAVING" clause defined in the SQL language, and which allows to specify restrictions on the resultant groups.

3.2 Timeseries queries

Timeseries queries are not very different from GroupBy ones, they also allow to filter data and to calculate aggregations over a precise time interval, but group results according to the specified query granularity only, as no grouping dimensions can be specified for the query. The formulation of a Timeseries query differs from the GroupBy one in the impossibility to declare dimensions to perform groupings over, a "limitSpec" field as well as a "HAVING" field.

3.3 TopN queries

The TopN query is also near similar to a GroupBy one, with the difference that it allows to make groupings only over a single dimension. In fact, TopN queries have been proposed as a more efficient alternative to GroupBy queries when groupings are made over a single dimension. The syntax of the query is similar to that of the GroupBy's one, except that it uses two more fields, the "Threshold" which indicates the maximum number of elements to return in the result, and the "metric" field that specifies the metric with which the TopN query results would be sorted. However, the "Having" and the "limitSpec" fields are not supported yet for TopN queries.

3.4 Search queries

Search queries allow to select values from attributes (dimensions or metrics) that satisfy specific research criteria. They contain three new fields not met in the previous query models. The "searchDimensions" field, that indicates the dimensions on which the search criteria will be applied to filter data. These dimensions also form the columns of the resultant elements. The "query" field allows to specify the search criteria to apply on the "searchDimensions". Entries from these dimensions that match the criteria will be returned in the result. The last one is the "sort" field, which indicates the sorting type to be applied over the resultant set.

3.5 The Select query

The Select query has the same role as that of the SQL language. It allows to filter data in order to select a subset of a data source according to a defined search criteria. The two fields "filter" and "interval" are used to specify those criteria. The request also allows to indicate the columns to return in the result by specifying the two fields, "dimensions" and "metrics".

4. BENCHMARKS

Experiments have been performed to compare *Roaring* and *Concise*'s performances under *Druid*. These benchmarks use a *Druid* cluster launched on a single node with an eight cores AMD FX^{TM}-8150 processor of a 3.60 GHz clock rate and 32 GB of RAM. We have used a 64-bit Oracle JVM server on an Ubuntu 12.4.1 LTS Linux system.

A 1 GB relational table containing about 6 million rows was generated from the TPC-H [5] benchmark as a data set for these experiments. After that, the Data was loaded into *Druid* and stored in two different data sources. The first data source stores segments indexed by *Concise* bitmaps, and the other one maintains segments indexed with *Roaring* bitmaps. First, we measure the average execution time of every query type previously presented on these both data sources, in order to compare the query execution times when accelerated with *Concise* and *Roaring*. The attribute "l_shipdate" has been taken as the timestamp on which segments would be built. Events and segments granularities were fixed to a day, as the "l_shipdate" attribute has a daily granularity.

Before calculating an average query running time, a warming-up phase is performed, which consists on repeating a query execution until stabilization of its running times. In our tests, 10 repetitions were sufficient. After this step, a query is launched 100 times, and finally, the average time of these executions is presented. The code of these benchmarks is freely available online.[9]

In order to evaluate *Roaring* and *Concise*'s performances on several types of data, the queries running times are calculated with bitmaps of four different densities: very low, low, average and high. In these tests, bitmap density corresponds to the cardinality of the related bitmap, which refers to the number of 1 bits contained within the bitmap. To catch reliable measures, the "context" field has been specified with the following values for every benchmarked query, "context": {"useCache": false, "populateCache": false}. This enforces *Druid* to compute a query result at each execution by avoiding it from caching that result after the first query execution and directly selecting it from cache during the remaining repetitions, which would corrupt the captured measures.

As an interval value, the following "intervals" field has been specified for all benchmarked queries, "intervals": ["

[9]https://github.com/samytto/BenchmarkingRoaingOnDruid

1980-12-31T23:59:59.999/2005-01-30T00:00:00.000 "].

4.1 Experimenting aggregation queries perforrming logical ORs

A first series of experiments was driven to evaluate queries execution times that first filter data by performing boolean ORs between bitmaps, then calculate aggregations on the data subset selected after the first operation. These requests are of types: *GroupBy*, *TopN* and *Timeseries*.

These first benchmarks start by evaluating the response times of a *GroupBy* query that performs logical ORs between bitmaps of very low densities, with cardinalities that vary between $(200, 3000)$, to select a reduced data subset from the requested data source. 22 bitmaps are aggregated during each query execution. After that, a GroupBy is realized on the attribute dimension "l_shipmode" by aggregating all values of the "l_tax" metric attribute appearing with a same distinct "l_shipmode" dimension value. Two attributes were selected in the results, one of dimension type and the other one of metric type. This reduced number of attributes was taken to attenuate the amount of groupings and aggregations performed when answering a query, and to let the bitmaps aggregations dominate the overall query execution time, what would otherwise prevent us from distinguishing between *Roaring* and *Concise*'s performances. Due to space limitation, no example has been presented for the benchmarked GroupBy queries.

For tests on low cardinality bitmaps, the "filter" part of the GroupBy query has been changed in order to only select bitmaps of cardinalities varying between $(100\,000, 900\,000)$.

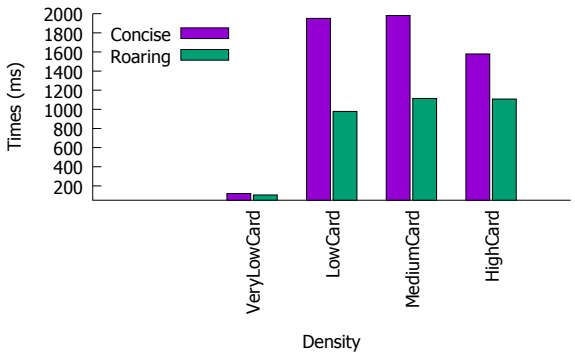
In the case of the experiments conducted over average cardinality bitmaps, eight bitmaps from the previous query "filter" part have been replaced by new bitmaps of average cardinalities from $(1\,000\,000, 1\,800\,000)$. Only eight bitmaps have been changed because the tested data set contains only that number of bitmaps with such cardinalities.

For tests over high cardinality bitmaps, a similar approach to the previous one has been followed where three average density bitmaps were replaced by high density ones, with cardinalities belonging to $(2\,200\,000, 3\,000\,000)$. Only three bitmaps with such cardinalities have been found in the whole used data set.

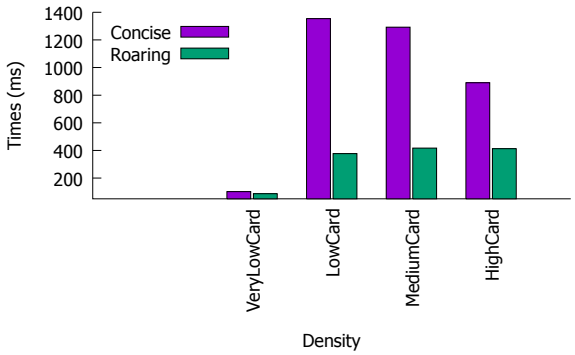
The second benchmarked query is TopN. This query seems nearly identical to a GroupBy query. The same values used for the previously benchmarked GroupBy query has been taken in the TopN query fields. For the "metric" part of the TopN query, the "l_tax" metric attribute was used to determine the order in which results would be presented. Also, a threshold of 100 was chosen to specify a limit on the number of the resultant elements, which in this case would consist of the first 100 returned hits. The same bitmaps adopted for the GroupBy query have been used to evaluate the TopN query performances with very low, low, average and high density bitmaps.

The third evaluated query is Timeseries. The same values used for the GroupBy query have been adopted for this request on its various fields. Also, the same bitmaps selected for the GroupBy query have been used for this query to evaluate its performances on bitmaps of several densities.

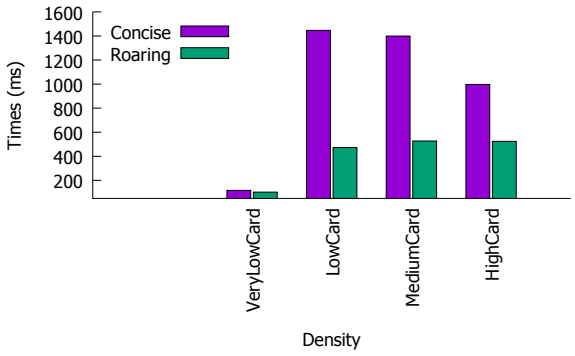
Figure 1 illustrates the graphics representing the running times of the three various types of aggregation queries supported by Druid when performing logical ORs between bitmaps. The results show that queries executed on segments



(a) GroupBy queries response times performing logical ORs between bitmaps of different densities.



(b) Timeseries queries response times performing logical ORs between bitmaps of several densities.



(c) TopN queries response times computing logical ORs between bitmaps of several densities.

Figure 1: Aggregation queries response times operating logical ORs between bitmaps.

indexed by *Roaring* were faster than those launched on segments indexed with *Concise*. Indeed, by operating logical ORs between very low density bitmaps to filter data, queries that use *Roaring* were, on average, 16%[10] faster in the case of Timeseries and TopN queries, and 12% more efficient for the GroupBy query compared to requests using *Concise* compressed bitmaps.

By increasing the bitmaps density, *Roaring* became even more faster than *Concise*. Effectively, on low density bitmaps, *Roaring* improved by 72%, 67% and 50% the running times for Timeseries, TopN and GroupBy queries, respectively, compared to the results obtained for the same queries when running over *Concise* bitmaps. Similarly, *Roaring* results on average cardinality bitmaps exceeds those of *Concise* ones by 68%, 62% and 43%, respectively, for the following requests: Timeseries, TopN and GroupBy. However, on high density bitmaps, the difference between *Roaring* and *Concise*'s performances shrinks. This is due to the long 1-bit sequences that compose the bitmaps introduced in this step. Such bit sequence can be effectively compressed with the RLE encoding employed by *Concise*. Also, *Druid* employs a strategy to effectively process the union of several bitmaps compressed by *Concise*, which consists in handling first the 1-fill CPU words during logical OR computations, what allows to jump many treatments of literal and 0-fill CPU words during such operations, resulting in significant processing times accelerations. However, *Roaring* remains more effective than *Concise* on these densities, by offering 26%, 22% and 11% better running times than those calculated with *Concise* for the following queries: Timeseries, TopN and GroupBy, respectively.

4.2 Benchmarking aggregation queries operating logical ANDs

A second series of experiments have been realized to compare the execution times of the three previous types of aggregation queries when launched on segments indexed with *Roaring* and *Concise*, but by filtering data this time using logical ANDs between bitmaps. As for the previous tests, four sets of bitmaps have been selected, each with a specific density: very low, low, average and high. A query filters data by calculating the logical AND of seven bitmaps. Bitmaps used for a query belong to distinct dimensions, this ensures to always obtain an non-empty resultant set. An importance is given to that case because if an empty bitmap is encountered during logical calculations, the OLAP engine would skip the treatment of the remaining bitmaps non considered yet, before returning an empty result. Such a case is to be avoided in order to take reliable measures.

An example of the GroupBy query executed on the *Concise* indexed data set and that operates logical ANDs between very low density bitmaps, characterized by cardinalities falling in $(0, 3000)$, is presented below:

```
{
  "queryType"  : "groupBy",
  "dataSource" : "TPCH_benchmark_concise",
  "granularity": "all",
  "dimensions": ["l_shipmode"],
  "filter": {
    "type": "and",
    "fields": [
    {"type": "selector", "dimension": "l_shipdate",
         "value":"1997-06-01"},
    {"type": "selector", "dimension": "l_commitdate
        ", "value":"1997-05-12"},
    {"type": "selector", "dimension": "
        l_receiptdate","value":"1997-06-02"},
    {"type": "selector", "dimension": "l_suppkey",
        "value": "4623"},
    {"type": "selector", "dimension": "l_comment",
        "value": "c packages"},
    {"type": "selector", "dimension": "l_partkey",
        "value": "22120"},
    {"type": "selector", "dimension": "l_orderkey",
         "value": "5050562"}
    ]
      },
  "intervals": [ "1980-12-31T23
      :59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
    { "type":"doubleSum", "name":"l_extendedprice",
         "fieldName":"L_EXTENDEDPRICE_doubleSum" }
  ]
}
```

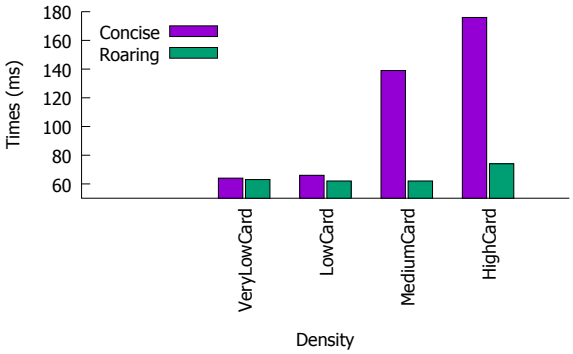
To test the precedent GroupBy query on low density bitmaps, their five least dense bitmaps were replaced by bitmaps with cardinalities belonging to the interval $(100\,000, 900\,000)$. With a similar manner, the three least dense bitmaps of the query used in low densities have been replaced by average density bitmaps, with cardinalities from $(1\,000\,000, 1\,800\,000)$. For the tests on high density bitmaps, the query of the average density tests was modified by replacing its two least dense bitmaps with 2 high density bitmaps having cardinalities belonging to $(2\,000\,000, 3\,000\,000)$. The two added bitmaps provide from the two unique dimensions of the benchmarked data set having bitmaps of such cardinalities.

The two remaining queries, Timeseries and TopN, have also been benchmarked on the same data sets with the same bitmaps of varied densities used to test the GroupBy query. The Timeseries and TopN queries fields have been chosen to be the same as those of their GroupBy counterpart on each density, with the following additional fields for the TopN query: "threshold": 100, "dimension": "l_shipmode", "metric": "l_extendedprice".

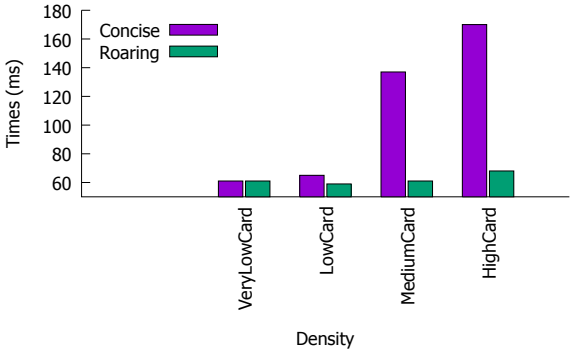
Figure 2 presents the average execution times of the three types of aggregation queries evaluated in these experiments using bitmaps of various densities. On very low densities, logical calculations are done very quickly between tiny bitmaps, so making queries response times independent from bitmap logical operations. This statement explains the nearly identical performances obtained for each query on these densities when executed on segments indexed by *Roaring* and *Concise*.

On little more denser bitmaps, low densities, *Concise* and *Roaring* performances began to distinguish from each other, and revealed that *Roaring* allowed to answer 10%, 9% and 6% more quickly than *Concise* the respective requests: TopN, Timeseries and GroupBy. As densities increase, response times substantially raise for *Concise*, contrary to *Roaring* which is more efficient on denser data. Indeed, on average density bitmaps, queries answered with *Roaring* are about 55% faster than those using *Concise* for each of the three query types. On high densities, the performance difference between both compressed bitmap models increases even more, and *Roaring* reached a 60% acceleration for the Timeseries query and a 58% one for both TopN and GroupBy queries compared to *Concise*.

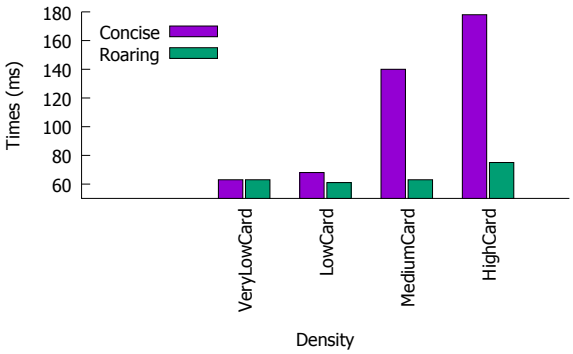
[10] A percent value, v, is calculated as follows: if x is the smallest measured average time and y the greater one, than $v = (1 - (x/y)) \times 100$. Finally, the result is rounded to the closest integer.



(a) GroupBy queries response times performing logical ANDs between bitmaps of different densities.

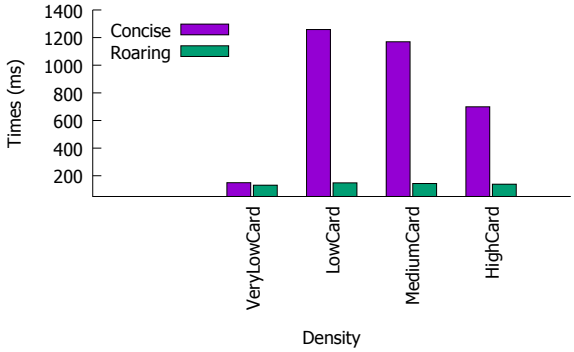


(b) Timeseries queries response times performing logical ANDs between bitmaps of several densities.

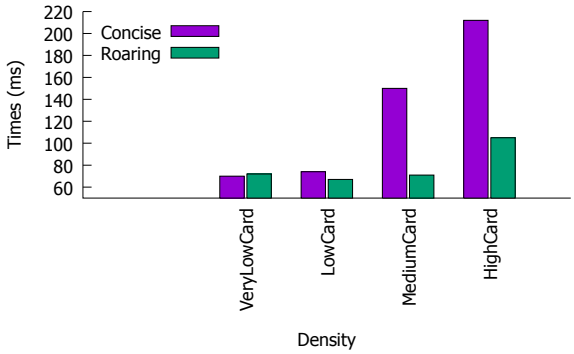


(c) TopN queries response times performing logical ANDs between bitmaps of varied densities.

Figure 2: Aggregation queries running times perfoming logical ANDs between bitmaps of varied densities.



(a) Select queries response times performing logical ORs between bitmaps of different densities.



(b) Select queries response times performing logical ANDs between bitmaps of various densities.

Figure 3: Select queries response times performing logical ORs and ANDs between bitmaps of several densities.

4.3 Benchmarking Search queries

In this series of experiments, *Roaring* and *Concise* performances were compared for answering the two types of search queries supported by *Druid*: Select and Search, when filtering data with bitmaps of different densities. Two versions of experiments have been conducted for the Select query. In the first one, logical ANDs have been executed between bitmaps to filter the data set, and in the second one, logical ORs have been performed. The bitmaps used to benchmark aggregation queries when performing logical ORs and ANDs on several densities have been used to test the Select query. The attributes, "l_shipmode" and "l_shipdate", were specified within the "dimensions" field, and the attributes, "l_tax" and "l_quantity", were mentioned within the "metrics" field for all Select queries tested. The granularity of the queries was fixed to a day. Figure 3 presents the response times of all the Select queries benchmarked.

Results have shown that *Roaring* has boosted logical ORs processing by 12% compared to *Concise* on very low density bitmaps, by 88% on low and average density ones, and by 80% on high density bitmaps. For Select queries calculating logical ANDs between bitmaps, *Roaring* has shown similar performances with those of *Concise* on very low density bitmaps, whereas it has improved the running times by 9%, 53% and 50% with regard to *Concise* on low, average and high density bitmaps, respectively.

Experiments were also done to evaluate Search queries performances on bitmaps of various densities represented with *Roaring* and *Concise*. An example of the request tested

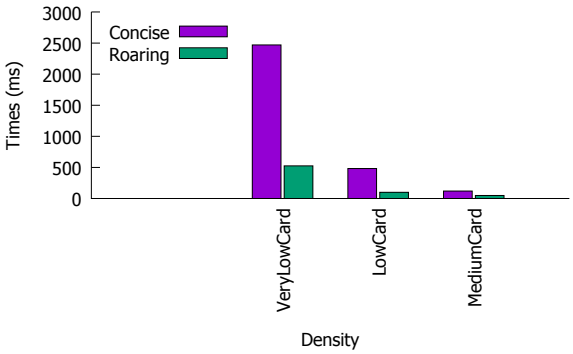


Figure 4: Search query running times performing logical ORs between several density bitmaps

on very low density bitmaps is given below:

```
{
  "queryType": "search",
  "dataSource": "TPCH_benchmark_roaring",
  "granularity": "all",
  "searchDimensions": [
    "l_receiptdate",
    "l_suppkey"
  ],
  "query": {
    "type" : "insensitive_contains",
  "value" : "9"
  },
  "intervals": [ "1980-12-31T23
       :59:59.999/2005-01-30T00:00:00.000" ],
  "context":{"useCache":false,"populateCache":false
       ,"finalize":false}
}
```

To evaluate Search query performances on various densities, we select two different dimension attributes to test with on each density. On the very low densities, both dimensions, "l_receiptdate" and "l_suppkey", whose bitmaps possess average cardinalities of 2645 and 282, respectively, have been chosen. The pattern "9" has been used as a search criteria to select bitmaps associated with values from the chosen dimensions and for which the "9" pattern figures in their representations. On the low densities, the dimensions, "l_discount" and "l_quantity", have been taken, which are characterized by bitmaps of 500 000 and 120 000 average cardinalities, respectively. The pattern "0" has been adopted as a search criteria. Bitmaps of 1 500 000 average cardinalities have been selected from both dimensions, *l_returnflag* and *l_shipinstruct*, to realize tests with average density bitmaps. Figure 4 presents the average calculated running times.

On the very low density bitmaps, logical ORs between 992 bitmaps have been performed. The execution times reached with *Roaring* were 79% ($\approx$ 5 times) faster than those calculated with *Concise*. On the low densities, the aggregation of 16 bitmaps has been calculated, and *Roaring* results have been 79% ($\approx$ 5 times) better than *Concise* ones. On average density bitmaps, *Roaring* calculated logical ORs between 3 bitmaps 60% ($\approx$ 3 times) faster with regard to *Concise*.

4.4 Benchmarking performances for retrieving set bits from bitmaps

Another performance aspect indicating the efficiency of a compression bitmap library is the speed at which it allows to iterate over a bitmap to extract the positions of its 1 bits (set bits). These set bits map the positions of the records satisfying the query criteria and are used by the OLAP engine to access and retrieve the final query result. A series of experiments was put into practice to compare the speed of both *Roaring* and *Concise* when performing such operations under *Druid*. To evaluate this performance aspect, we have opted for the Timeseries query, as it is among the requests allowing to process aggregations and thus can limit itself to the display of a single aggregated value, that helps to decrease the I/O costs needed to return out the final results. Also, Timeseries queries do not perform groupings on dimension values, as it is the case for the GroupBy and TopN queries, which can consume important processing times and dominate the global running times. Therefore, the Timeseries query was considered as the ideal request that lets execution times to entirely depend on the time spent for scanning bitmaps.

Logical operations have also been moved away to do not penalize either of the two compared schemes, given that one of them can sometimes be less efficient than the other one. So, a single bitmap is used at the filter level for every query. To verify *Roaring* and *Concise* performances on various densities, the tests handle very low, low, average and high density bitmaps. The code below presents a Timeseries query filtering data with a bitmap of very low density containing 2707 bits at 1:

```
{
  "queryType"  : "timeseries",
  "dataSource" : "TPCH_benchmark_roaring",
  "granularity": "all",
  "context":{"useCache":false,"populateCache":false
       },
  "filter": { "type": "selector", "dimension": "
       l_shipdate", "value": "1997-06-01" },
  "intervals": [ "1980-12-31T23
       :59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
        { "type": "count", "name": "count"}
  ]
}
```

For tests on the remaining densities, only the bitmap at the filter level of the request is changed by another one corresponding to the tested density. The bitmap of the low density is the one indexing the value "5" of the "l_linenumber" dimension, and which contains 643 287 set bits. On the average density, the selected bitmap is associated to the value "1" of the "l_linenumber" dimension, and contains 1 500 000 set bits. For the high densities, the bitmap of the value "N" belonging to the "l_returnflag" dimension has been taken, which has 3 043 852 set bits. Figure 5 gives an overview of the average running times captured for the Timeseries query when using *Roaring* and *Concise* bitmaps of various densities.

On the most weak densities, results of both bitmap compression schemes are almost nil and equivalents. On low densities, *Roaring* has iterated over bitmaps 18% faster compared to *Concise*. On the average densities, response times obtained with *Roaring* surpassed those of *Concise* by 11%. On the high densities, the *Concise*'s capacity to compress long 1-bit sequences allowed this bitmap scheme to answer queries 11% more quickly than what has been calculated for *Roaring*.

4.5 Experimenting times performances to calculate bitmap complements

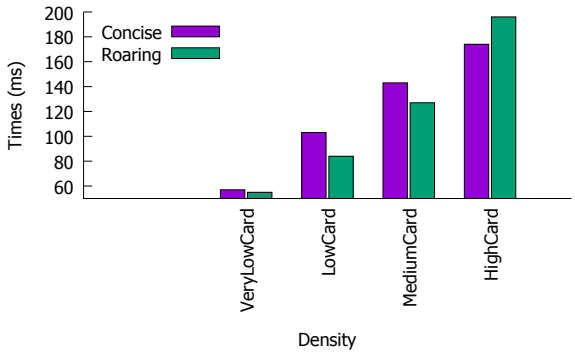


Figure 5: Timeseries queries running times when iterating over bitmaps of several densities

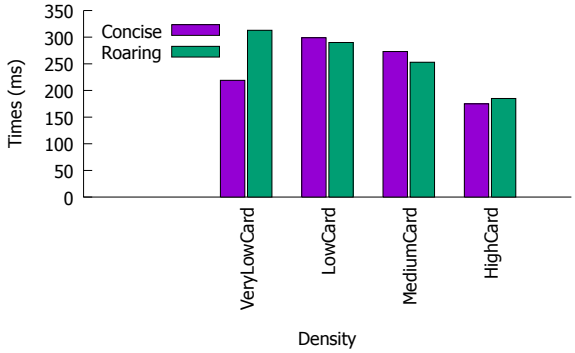


Figure 6: Running times of Timeseries queries computing the complement of bitmaps with several densities

These benchmarks evaluate the average running times of a Timeseries query aggregating values from the metric attribute "count" reached by means of the bitmap complement computed from the filter. Bitmaps of the previous evaluations have been used to compare query performances obtained with *Roaring* and *Concise* on various bitmap densities. Results have been reported on figure 6.

Results show that *Concise* performances are clearly superior to those of *Roaring* on the lowest and highest densities (by 30% and 5%, respectively). This is explained by the presence of long sequences of 0 bits and 1 bits, respectively on both densities, that are well compressed by the RLE encoding adopted by *Concise*. Effectively, *Concise* can compress such a bit sequence in a single CPU word, on which it becomes very easy to derive the complement by only inverting one bit of the CPU word that indicates the sense (value) of its compressed bits (0 or 1). Whereas, *Roaring* generally requires to create new containers representing the unset bits of the original bitmap (operations very often met on very low densities), to scan all new containers in order to populate them with the missing integers, and to convert the existing containers to another format (arrays to bitmaps or bitmaps to arrays). However, on the low and average densities, such bits sequences become very rare. In these cases, *Concise* is mainly constituted of literal words that need to be complemented by calculating their opposite, allowing *Roaring* to show a light performance advance compared to *Concise*, which is about 3% on the low densities and of 7% on the average ones.

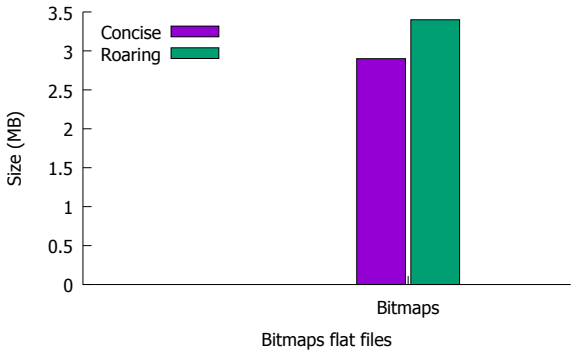


Figure 7: *Roaring* and *Concise*'s disk space consumptions

4.6 Evaluating Roaring and Concise's memory usage

After the work realized to evaluate the temporal performances of *Roaring* and *Concise* when executing analytical queries under *Druid*, a question which can not go unnoticed is the one concerning the space usage of both compression bitmap techniques. To give a brief idea of both models space performances, we bring back the total size of the flat files storing the serialized bitmap indexes of both data sources (one being indexed with *Roaring* and the second one with *Concise*) used during the previous tests. The figure 7 shows sizes occupied by flat files storing *Roaring* and *Concise* bitmaps.

Although *Roaring* generally offers remarkable answer times, results show that this scheme is about 15% greedier than *Concise* regarding memory space occupation. However, this very rarely affects the speed of reading bitmaps from the disk, because *Roaring*'s high-level index often avoids loading non-used containers during processing, contrary to *Concise* for which a bitmap has to be completely loaded into memory to perform computations on it. Also, *Roaring*'s model allows the CPU to process containers very quickly by exploiting super-scalar calculations that can handle several bit sequences in parallel, which is impossible to realize with *Concise* as it requires to perform conditional branching for every represented CPU word, that results on slowing down the running times.

5. CONCLUSION

After integrating *Roaring bitmap* as an indexing solution into the *Druid* OLAP engine, this work has been realized in collaboration with *Druid* developers to evaluate and analyze the time-space performances of *Roaring bitmap* under *Druid*. First, an introduction of the main facts having led to the realization of this project has been given. Then, a detailed presentation of *Druid*'s important concepts has been made. A section giving a brief overview of the various types of analytical queries supported by *Druid* and that require bitmaps to fast filter data has followed. The last section presented the benchmarks conducted to evaluate and analyze the time-space performances of the two compressed bitmap schemes adopted by *Druid*, *Roaring bitmap* and *Concise*, under this system. Results have shown that *Roaring bitmap* has been able to improve up to $\approx 5\times$ Search queries processing times, and up to $\approx 2\times$ aggregation queries running times when compared to *Concise*.

Having noticed that it happened to *Concise* to be more

compact and to run more quickly than *Roaring bitmap* on data formed of long 1-bit sequences, a new container model has been developed for *Roaring bitmap* which is especially adapted to this kind of data and is able to compress such bit sequences by applying an RLE encoding. An evaluation of this new *Roaring bitmap*'s version on the *Druid* OLAP engine is planned for future work.

6. ACKNOWLEDGEMENTS

This work was supported by NSERC grant number 261437. We are grateful to Druid developers, specifically X. Léauté and N. Bangarwa for their help and feedback.

7. REFERENCES

[1] S. Chambi, D. Lemire, K. Owen, and R. Godin. Better bitmap performance with Roaring bitmaps. *Software Practice and Experience (SPE)*, 46(5):709–719, may 2016.

[2] J. Chang, Z. Chen, W. Zheng, J. Cao, Y. Wen, G. Peng, and W. Huang. SPLWAH: A bitmap index compression scheme for searching in archival internet traffic. In *2015 IEEE International Conference on Communications (ICC)*, pages 7089–7094, London, England, 2015. IEEE.

[3] Z. Chen, Y. Wen, Y. Cao, W. Zheng, J. Chang, Y. Wu, G. Ma, M. Hakmaoui, and G. Peng. A survey of bitmap index compression algorithms for big data. *Tsinghua Science and Technology*, 20(1):100–115, 2015.

[4] A. Colantonio and R. D. Pietro. Concise: Compressed 'n' composable integer set. *Information Processing Letters*, 110(16):644–650, jul 2010.

[5] T. P. P. Council. TPC BENCHMARK H. http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf, 2014.

[6] F. Deliège and T. B. Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*, pages 228–239, New York, NY, USA, 2010. ACM.

[7] Druid. Querying. http://druid.io/docs/0.8.2/querying/querying.html, 2015.

[8] A. Grand. Frame of reference and Roaring bitmaps. https://www.elastic.co/blog/frame-of-reference-and-roaring-bitmaps, 2015.

[9] A. Hall, O. Bachmann, R. Bussow, S. Ganceanu, and M. Nunkesser. Processing a trillion cells per mouse click. In *The 38th International Conference on Very Large Data Bases (VLDB)*, volume 5, pages 1436–1446, Istanbul, Turkey, 2012. VLDB.

[10] S. Melnik, A. Gubarev, J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *The 36th International Conference on Very Large Data Bases (VLDB)*, volume 3, pages 330–339, Singapore, 2011. VLDB.

[11] M. Rios and J. Lin. Distilling massive amounts of data into simple visualizations: Twitter case studies. In *The 6th International AAAI Conference on Weblogs and SocialL Media (ICWSM)*, pages 22–25, Dublin, Ireland, 2012. ICWSM.

[12] K. Shvachko, K. Hairong, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Incline Village, NV, USA, 2010. IEEE.

[13] E. TSCHETTER. Introducing Druid: Real-Time Analytics at a Billion Rows Per Second. http://druid.io/blog/2011/04/30/introducing-druid.html, 2011.

[14] Wikipedia. Shared-nothing architecture. https://en.wikipedia.org/wiki/Shared_nothing_architecture, 2015.

[15] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)*, 31(1):1–38, March 2006.

[16] L. Wu, R. Sumbaly, C. Riccomini, G. Koo, H. Kim, J. Kreps, and S. Shah. Avatara: OLAP for web-scale analytics products. In *The 38th International Conference on Very Large Data Bases (VLDB)*, volume 5, pages 1874–1877, Istanbul, Turkey, 2012. VLDB.

[17] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: a real–time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 157–168, New York, NY, USA, June 2014. ACM.

[18] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, pages 1–7, Boston, MA, USA, 2010. ACM.