Building OLAP Data Analytics by Storing Path-Enumeration Keys into Sorted Sets of Key-Value Store Databases

Luis Loyola
R&D Division
SkillUpJapan Corporation
Tokyo, Japan
loyola@skillupjapan.co.jp

Fernando Wong
R&D Division
SkillUpJapan Corporation
Tokyo, Japan
f.wong@skillupjapan.co.jp

Daniel Pereira
R&D Division
SkillUpJapan Corporation
Tokyo, Japan
d.pereira@skillupjapan.co.jp

***Abstract*—In this paper, a novel approach that allows the retrieval of OLAP analytics by means of storing multidimensional data in key-value databases is described and evaluated. The proposed mechanism relies on generated keys based on the path-enumeration model built upon a tree representation of a relational database. Each key represents a combination of OLAP dimension labels and is inserted into a key-value sorted set where its associated values are the metrics of interest, i.e., the OLAP *facts*. Our implementation for a real advertisement server system in Redis and its performance comparison with a relational OLAP (ROLAP) database running on top of MySQL, show that our proposed scheme can answer complex multidimensional queries much faster, while keeping a great flexibility of the data schema. In contrast with general ROLAP schemes, which usually require the pre-computation of tables for specific queries, the sorted sets in the proposed system are not pre-computed but generated on demand, so no further delay is introduced. To the best of the authors' knowledge, this is the first system that deals with mapping relational data structures into sorted sets of key-value store databases to answer complex queries on multidimensional data.**

***Keywords*-OLAP; path-enumeration keys; sorted sets; databases; key-value stores.**

I. Introduction

By making use of object-relational mapping (ORM) [9], object-oriented programming classes on the application level can be represented in a persistent way on a relational database. With ORM, the Entity Relation map of a relational database can be represented by a parent-child relationship tree. In this type of trees, relations between nodes can be represented using verbal expressions like *"has one"*, *"has many"*, *"has and belongs to many"*, *"belongs to"* and so on. In fact, these expressions describing relationships among classes are very popular in abstraction layers of SQL data connectors developed for mainstream scripting languages like Java[2], Python[5] or Ruby[8].

Online Analytical Processing (OLAP) databases are a key tool for the analysis of large amounts of data. An OLAP [20] database corresponds to a multidimensional database where measures or statistics of interest are pre-calculated in the so-called *fact tables*. The flexibility and high-speed with which *multi-dimensional OLAP* (MOLAP) allows to perform complex queries involving several dimensions have turned it into one of the most useful tools for modern data analysis. On the other hand, during the last decade there has been a big hype on no-SQL databases, featuring various software solutions and open-source tools. The technologies, concepts and algorithms used in them differ quite substantially but one can observe the presence of the key-value store concept in many. The basics of such key-value store systems are databases where a hash key can be associated to a value, which can range from a simple text string to more complex data structures such as arrays, lists or even binary files.

This paper introduces an algorithmic approach that maps a relational database into tailored sorted sets in a key-value store database. In a cold start, no data needs to be transferred from the relational database to the proposed system since from the application's point of view it works as a caching layer, not needing to pre-compute tables as they are generated on the fly as the queries arrive. To the best of the authors' knowledge, this is the first system that deals with automatically mapping relational data structures into sorted sets of key-value store databases to answer complex queries on multidimensional data like in OLAP.

The proposed data representation structure is helpful for analyzing multidimensional data in real-time as in OLAP. The advantage of the proposed approach is two-folded: 1) on the practical side, OLAP solutions are quite costly and complex, so they usually require a large investment on dedicated servers, software licenses and staff training. In contrast, there exist a handful of excellent open-source enterprise-level key-value store databases which are easy to configure and use, and those are the target tools of this work. 2) On the technical side, the mechanism proposed in this document differs substantially to the OLAP approach for consolidating the multidimensional data structure for analytics, and furthermore allows the user to still keep her relational database in place to store the almost-static data.

The paper is organized as follows: in Section II, we review some key concepts. Works related to our approach are briefly mentioned in Section III. Section IV provides a more in-depth description of the proposed mechanism. Section V

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-242-4

shows performance measurements and comparison with a ROLAP database running on top of MySQL in a real implementation. Finally, some final remarks are provided in Section VI.

II. Background

In this section, a brief overview of key concepts like OLAP, sorted sets, path enumeration as well as a brief description of the Redis database are provided.

A. OLAP

OLAP stands for Online Analytical Processing, a category of software tools that provides analysis of data stored in a database. OLAP tools enable users to analyze different dimensions of multidimensional data, and allow flexible and fast access to the data and those multiple dimensions. Such dimensions are stored as a side of what is called an OLAP cube, or hypercube, building that way a multidimensional cube which stores information, also known as facts or measures. Such facts store no other than the aggregated totals of relevant key information that one wants to analyze. All this data is, typically, created from a star or snowflake schema of tables in a relational database (RDB) such as the simplified one shown in Figure 1, which depicts the RDB for a real advertising server application deployed by our company. Our star schema features six *dimensions*, represented as the peripheral tables, and we are interested in measuring the aggregated results from the central *facts* table, all relevant to the presented dimensions. Each *measure* can be thought of as having a set of *labels*, or meta-data associated with it. A *dimension* is what describes these *labels*; it provides information about the *measure*. As a practical example, in our systems, a cube contains the clicked advertisements as a measure and the advertisement space as a dimension. Each clicked advertisement can then be correlated with its campaign, publisher, advertiser, time, campaign resource dimensions and any other number of dimensions can be added, as long as data to correlate those dimensions with the click are added in the structure, such as a foreign key for instance. This allows an analyst to view the *measures* along any combination of the *dimensions*. Additionally, OLAP systems are typically categorized under three main variations:

- MOLAP
  It is the standard way of storing data in OLAP and that is why it is sometimes referred to as just OLAP. It stores the data on a multi-dimensional array storage where fact tables are pre-computed. It does not rely on relational databases.
- ROLAP
  It works on the top of relational databases. The base data and dimension tables are stored as relational tables and new tables are created to store the aggregated information. It gives the appearance of traditional slicing and dicing OLAP functionalities working with the data stored in the relational database.
- Hybrid
  There is no clear definition of what a "hybrid" OLAP system means, but in all cases the data is stored in both a relational database and special array storage with pre-computed data.

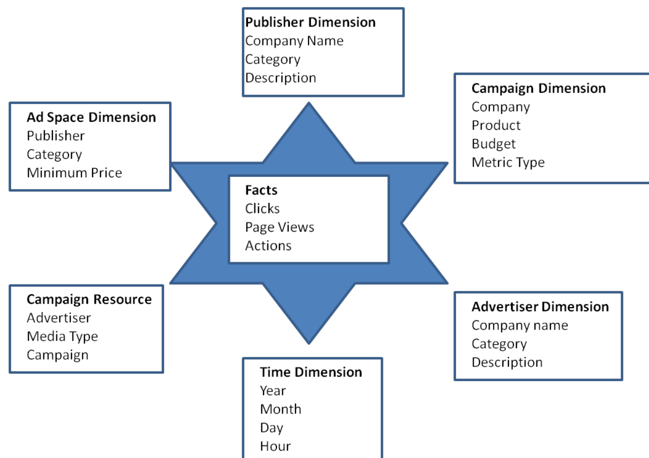

Figure 1. Sample star data model in OLAP.

B. Sorted Sets

A sorted set is a data structure defined by a list where each element has an associated score. In contrast with a normal set, a sorted set can be ordered based on its elements' score. Hence, a sorted set $\mathbf{Z} = \{(e_1, s_1), \ldots, (e_n, s_n)\}$, where each element e_i has score s_i, can be represented as a hash where each key has an associated score value and the elements in the set can be ordered by their score.

C. Path Enumeration Model

One of the properties of trees is that there is one and only one path from the root to every node. The path enumeration model stores that path as a string by concatenating either the edges or the keys of the nodes in the path. Searches are done with string functions and predicates on those path strings. There are two methods for enumerating the paths, edge enumeration and node enumeration. Node enumeration is the most common, and there is little difference in the basic string operations on either model. In this paper we apply the node enumeration method.

D. Redis

Redis [7] is an open-source memory-only key-value store database with a handful of useful data structures. Similar to other key-value data stores, Redis has as data model a dictionary where keys are mapped to values, with the essential differences that values can be other than strings of text and that it can handle collections, or sets, in an

unordered and ordered manner, which behave in the same way as the sorted sets presented in Section II-B.

At this moment, there is no effective way to distribute Redis other than manually distributing chunks of data in independent servers, typically known as *sharding*. Therefore, measuring the scalability of our proposed scheme highly depends on future work inherent to Redis development. As Redis stores the whole database in memory it is faster than conventional solutions, while maintaining some control over the data persistence. If needed, Redis allows us to take *snapshots* of data as well as asynchronously and periodically transfer data from memory to disk, however this is not relevant to the idea presented in this work. Finally, another benefit of using Redis is the lack of necessity to perform any data schema alteration providing us with higher flexibility to introduce changes on demand.

III. Related Work

Related work for OLAP analytics in key-value store databases is primarily focused on distributed algorithms like MapReduce, cloud services and in-memory OLAP architectures. Bonnet et al. [14], Qin et al. [28], Wang et al. [29], analyze and improve upon existing MapReduce algorithms while Abelló et al. [12] make use of distributed data stores like Bigtable [18], GFS [23] or Cassandra [26]. The reasoning behind the adoption of those architectures is due to the large amount of resources needed when dealing with large data sets. Such architectures also allow parallelization of queries and provide easy means of hardware scaling. However, solutions based on MapReduce are less efficient, due to lack of efficient indices, proper storage mechanisms, lack of compression techniques and sophisticated parallel algorithms for querying large amounts of data [27].

Some related literature focuses on using cloud services, like Dynamo [22] from Amazon, to improve upon current resource utilization and reducing costs [15], [16], [19], as several cloud platforms began to give access to cheap key-value stores that allow easy scaling. Such cloud services can, however, be problematic as privacy issues can arise, if the data cannot be anonymized [10]; data is often stored in untrusted hosts and replicated across different geographic zones, without much control from the users. Moreover, those cloud services lack important data structures, such as the presented sorted sets, for instance. Also, they are proprietary and closed to improvements from the developers, making it impossible to add or develop features when needed or customize a certain part of the engine.

More literature [17], [21] emphasizes the need to combine the benefits of NoSQL database systems with traditional RDBMS, in order to effectively handle big data analytics. With the growth of data stored in RDBMS databases, it is deemed impossible to cover all the needed scenarios and still expect timely reporting of results. By using key-value store databases, data analytics can be offloaded from RDBMS, specially if the operations are simple lookups of objects, with no urgency of having consistent data at a very granular level. Recently, the decline in memory price has allowed for new solutions, entirely based on memory, to become financially feasible. Therefore, architectures providing in-memory OLAP capabilities have arisen, such as DRUID [1], QlikTek [6], SwissBox [13] architecture, H-Store [24] and Cloudy [25]. However, not much detail on their architectures and internal implementation aspects are known and no comparison to any state of the art solution has been published. Most of the solutions provide very controlled environments, often specific to an operating system and are business oriented.

IV. Proposed Scheme

In a tree representing parent-child-relation classes mapped into an underlying relational database, we can generate a basic unique key for each branch by hierarchically listing in a top-to-bottom order the node ids found all the way down from the root to the bottom leafs along every branch. This is equivalent to picking up the deepest paths of every branch from the path enumeration model applied to general trees. In order to store all possible combinations of relevant dimensions for our reports it is essential that every path from the path enumeration model has an associated key. This defines a set of deepest paths $\mathbf{D} = \{\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n\}$ for n root-through-bottom-leaf paths where each $\mathbf{d}_i$ corresponds to a path represented by a list of nodes ordered from top to bottom in the tree hierarchy. Each of the nodes in the tree represents a dimension that can be included in the reporting system. After the set of deepest paths $\mathbf{D}$ is constructed, another set $\mathbf{D'} \supseteq \mathbf{D}$ is defined, so that $\mathbf{D'} = \{\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n, \mathbf{d}_1{:}\mathbf{d}_2, \mathbf{d}_1{:}\mathbf{d}_3, \ldots, \mathbf{d}_1{:}\mathbf{d}_2{:}\mathbf{d}_3{:}\ldots{:}\mathbf{d}_n\}$ and $|\mathbf{D'}| \leq (2^n - 1)$. Eventually, $\mathbf{D'}$ should only hold the combination of paths that are frequently consulted via the reporting system.

```
sorted_sets = {}
foreach path ∈ D' do
  | Insert path(time_slot) into sorted_sets
end
```

Algorithm 1: Initialization of sorted sets.

In the next step, one has to concentrate on the statistical values or parameters that should be measured, i.e., the so-called facts in OLAP. After identifying the facts we define sorted sets combining the time dimension (in the maximum resolution of interest) with the paths of $\mathbf{D'}$. Thus, the name of each sorted set can be built as specified on Algorithm 1 where *time_slot* represents one time slot in the maximum level of granularity for the reporting system. Every time there is an event related to one of the facts, the corresponding fact's score is incremented for the combination of all dimension ids related to the event. Hence, when

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-242-4

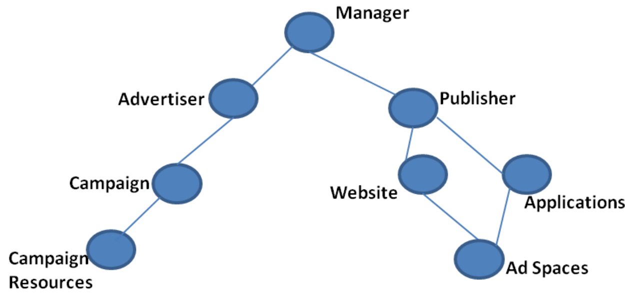

Figure 2. Example entity relationship diagram.

an event measured by a fact occurs, the score of a fact related to a particular combination of dimension ids, denoted as **dimension_ids**$(path)_{fact}$, in sorted set **path**$(time_slot)$ is incremented by one in case the fact corresponds to an aggregate statistic. If the fact is not measured by an aggregate statistic, the score is totally ignored for all set operations so the sorted set behaves just as a normal set.

For the sake of clarity, an example of the proposed data structure applied to a real advertisement server application is presented. Figure 2 shows the example data structure where the root node "Manager" corresponds to one advertising agency administrator. Each advertising agency has "Publishers" and each of those publishers has "Websites" and smartphone "Applications". In turn, each website and each smartphone application has one or many "Ad Spaces" which are sold to "Advertisers". On the other hand, each advertiser has "Campaigns" of products it wants to promote. Inside each campaign there are many advertising media assets like image banners, video banners, text banners, etc., which are called "Campaign Resources".

In the tree of the example shown in Figure 2, there exist three deepest paths given by the set **D** = { **M:A:C:R**[1], **M:P:W:S**[2], **M:P:Ap:S**[3]}. This set has to be united with the deepest path keys' combination set **E**, which corresponds to **E** = { **M:A:C:R:P:W:S**[4], **M:A:C:R:P:Ap:S**[5] } in our example. After the union operation has been performed, the resulting set contains five unique basic keys that can be derived in a straightforward manner from both sets above as:

$$\mathbf{D'} = \mathbf{D} \cup \mathbf{E} = \left\{ \begin{array}{c} \mathbf{M{:}A{:}C{:}R}, \mathbf{M{:}P{:}W{:}S}, \\ \mathbf{M{:}P{:}Ap{:}S}, \mathbf{M{:}A{:}C{:}R{:}P{:}W{:}S}, \\ \mathbf{M{:}A{:}C{:}R{:}P{:}Ap{:}S} \end{array} \right\} \quad (1)$$

In order to construct the full keys, we need to combine these five basic keys with each of the parameters or statistical measures we are interested in assessing, i.e. the facts in the OLAP jargon as well as the time expressed in the maximum resolution required by our reporting system. Let us say that in the example of Figure 2 we are interested in measuring the views and clicks as fact events. Thus, in the same way as in OLAP, our reporting system should be able to answer all types of multidimensional queries like: "what was the most clicked campaign resource from Advertiser **a** on ad spaces from Publisher **p** between 2011/07/19 and 2011/09/21?", or "what was the number of accumulated clicks and views on Campaigns from Advertiser **a** for all the Ad Spaces from Publisher **p**'s Web Sites between 2011/06/15 and 2011/12/21?", "what was the total number of accumulated clicks and views on campaign resource **r** when shown in Ad spaces from all applications of Publisher **p**?", etc. To build such a reporting system we make use of sorted sets. In order to name the sorted sets we need to follow the steps described below. For all deep-most branches in the tree and their combinations we name their associated sorted sets using the pattern **basic_key**(time_slot). If the time slot corresponds to one hour (i.e. the maximum resolution for accumulated statistics we are willing to tolerate is one hour) the sorted sets associated to the time period between 16:00 and 17:00 of 2011/07/09 can be named as **basic_key**(2011-07-09-16). Inside this sorted set we will increment the score of a particular combination of dimension ids based on the event-related fact. The pseudo-code below describes the basic logic to increment the score of the related basic-key elements in the sorted set.

```
if event = "view" then
    for basic_key ∈ D' do
        [dimension_ids(basic_key)_views].score++
    end
else if event = "click" then
    for basic_key ∈ BasicKeySet do
        [dimension_ids(basic_key)_clicks].score++
    end
end
```

Algorithm 2: Incrementing event-related facts.

As can be seen in Algorithm 2, after naming the sorted sets, we increment by one the score for the combination of ids associated with the basic_key and the related event. For example, let us suppose there is a click at 4:45 p.m. on Campaign Resource **r** from campaign **c** ran by Advertiser **a** (belongs to manager **m**) when shown at an ad space **s** from website **w** of publisher **p**. Algorithm 3 shows how each of the five **basic_key** $\in$ **D'** would be modified.

The above sets are suitable for the most granular questions over multiple dimensions for our reporting system. But what happens if we need to know, for example, the total accumulated clicks on website **w** of Publisher **p** during the

[1]Manager, Advertiser, Campaign, Campaign Resource
[2]Manager, Publisher, Web Sites, Ad Spaces
[3]Manager, Publisher, Applications, Ad Spaces
[4]Manager, Advertiser, Campaign, Campaign Resource, Publisher, Web Site, Ad Space
[5]Manager, Advertiser, Campaign, Campaign Resource, Publisher, Application, Ad Space

Copyright (c) IARIA, 2012.

$TimeSlot = \{$2011-07-09-16, 2011-07-09, 2011-07, 2011$\}$
foreach *t in TimeSlot* **do**
  In set **M:A:C:R**(t)
    [**m:a:c:r**$_{clicks}$].score++
  In set **M:P:W:S**(t)
    [**m:p:w:s**$_{clicks}$].score++
  In set **M:P:Ap:S**(t)
    nothing is modified
  In set **M:A:C:R:P:W:S**(t)
    [**m:a:c:r:p:w:s**$_{clicks}$.].score++
  In set **M:A:C:R:P:Ap:S**(t)
    nothing is modified
end

Algorithm 3: Modification of scores in basic key sets.

month of February 2011. In that case, we can obtain the requested measure by performing union operations among the sorted sets between $date_1 =$ 2011-02-01-00 and $date_2 =$ 2011-02-28-23. In this paper, we present three different approaches to deal with this type of queries. By default, we assume the scores of the same element are added during a union operation of sorted sets, however, the mathematical operation performed on the values during a union can be extended to more complex operations.

A. Approach 1

This approach consists of performing union operations only on the deep-most branch keys of the basic key set **D'**. For our example, the set

$$\mathbf{Q} = \bigcup_{d=date_1}^{date_2} \textbf{M:P:W:S}(d) \tag{2}$$

contains the total number of accumulated clicks for all ad spaces throughout all days within the specified period for all combinations of publishers, websites and ad spaces. Thus, in order to answer the query stated above, we just have to add the scores of values **m:p:w:***$_{clicks} \in \mathbf{Q}$, where $*$ symbolizes all ad spaces that belong to web site **w** of publisher **p** with manager **m**.

The disadvantage of the method above is that we need to search for all elements with pattern **m:p:w:***$_{clicks} \in \mathbf{Q}$ using regular expressions or other parsing techniques, and then add all their associated scores. This is time-consuming and expensive in terms of resources as we need to load all elements in **Q** to memory and perform the search operation for the pattern in a separate application. In order to address this problem, we store a **M:P:W:S**$(\mathbf{m}, \mathbf{p}, \mathbf{w}, *)$ set, which contains all combinations of the requested dimension ids in the form of **m:p:w:***$_{clicks}$. Then, we can simply perform an intersection between **M:P:W:S**$(\mathbf{m}, \mathbf{p}, \mathbf{w}, *)$ and **Q**, and sum the scores of the members in the resulting set

$$\mathbf{Q'} = \textbf{M:P:W:S}(\mathbf{m}, \mathbf{p}, \mathbf{w}, *) \cap \mathbf{Q} \tag{3}$$

In practice, we map the function **M:P:W:S(m**,**p**,**w**,*) to a set in Redis whose key follows the pattern **M:m:P:p:W:w:S**. Please note that **M:P:W:S(m**,**p**,**w**,*) is a regular set, while **Q** and **Q'** are sorted sets. In our context, the intersection between a sorted set and a regular set results in a sorted set whose members are present in both sets, while retaining the score they had on the sorted set. For example, let us assume that the clicks registered between $date_1$ and $date_2$ are contained in the sorted set

$$\mathbf{Q} = \left\{ \begin{array}{l} \mathbf{m}_1\textbf{:}\mathbf{p}_1\textbf{:}\mathbf{w}_1\textbf{:}\mathbf{s}_{1clicks}\textbf{=}2, \mathbf{m}_1\textbf{:}\mathbf{p}_1\textbf{:}\mathbf{w}_1\textbf{:}\mathbf{s}_{2clicks}\textbf{=}7, \\ \mathbf{m}_1\textbf{:}\mathbf{p}_2\textbf{:}\mathbf{w}_2\textbf{:}\mathbf{s}_{3clicks}\textbf{=}4, \mathbf{m}_1\textbf{:}\mathbf{p}_2\textbf{:}\mathbf{w}_2\textbf{:}\mathbf{s}_{5clicks}\textbf{=}5, \\ \mathbf{m}_2\textbf{:}\mathbf{p}_3\textbf{:}\mathbf{w}_3\textbf{:}\mathbf{s}_{7clicks}\textbf{=}3, \mathbf{m}_2\textbf{:}\mathbf{p}_3\textbf{:}\mathbf{w}_4\textbf{:}\mathbf{s}_{8clicks}\textbf{=}6 \end{array} \right\} \tag{4}$$

and that all combinations of ids for web site $\mathbf{w}_2$ of publisher $\mathbf{p}_2$ with manager $\mathbf{m}_1$ that have so far experienced events such as clicks and views during the entire history of the system are given by

$$\textbf{M:P:W:S}(\mathbf{m}_1, \mathbf{p}_2, \mathbf{w}_2, *) = \left\{ \begin{array}{l} \mathbf{m}_1\textbf{:}\mathbf{p}_2\textbf{:}\mathbf{w}_2\textbf{:}\mathbf{s}_{3clicks}, \\ \mathbf{m}_1\textbf{:}\mathbf{p}_2\textbf{:}\mathbf{w}_2\textbf{:}\mathbf{s}_{4clicks}, \\ \mathbf{m}_1\textbf{:}\mathbf{p}_2\textbf{:}\mathbf{w}_2\textbf{:}\mathbf{s}_{5clicks}, \\ \mathbf{m}_1\textbf{:}\mathbf{p}_2\textbf{:}\mathbf{w}_2\textbf{:}\mathbf{s}_{6clicks} \end{array} \right\} \tag{5}$$

Then, the resulting set **Q'** is denoted by

$$\mathbf{Q'} = \left\{ \mathbf{m}_1\textbf{:}\mathbf{p}_2\textbf{:}\mathbf{w}_2\textbf{:}\mathbf{s}_{3clicks}\textbf{=}4, \mathbf{m}_1\textbf{:}\mathbf{p}_2\textbf{:}\mathbf{w}_2\textbf{:}\mathbf{s}_{5clicks}\textbf{=}5 \right\} \tag{6}$$

The amount of clicks for this particular combination of manager, publisher and website between $date_1$ and $date_2$ is 9. For this example, the combinations $\mathbf{m}_1\textbf{:}\mathbf{p}_2\textbf{:}\mathbf{w}_2\textbf{:}\mathbf{s}_{4clicks}$ and $\mathbf{m}_1\textbf{:}\mathbf{p}_2\textbf{:}\mathbf{w}_2\textbf{:}\mathbf{s}_{6clicks}$ did not occur during the specified period of time and thus, were not included in either **Q** or **Q'**. However, they might have appeared if a larger time range had been specified. In our implementation, we initialize **M:P:W:S(m**,**p**,**w**,*) as an empty set and update it each time an event occurs in the system. Since adding or checking whether a member exists within a set is a constant time operation, this does not degrade the performance of the system. In this way, **M:P:W:S(m**,**p**,**w**,*) only contains combinations for which data is available, which in turn allows us to efficiently perform the intersection operation.

B. Approach 2

A simpler and faster alternative to the previous approach would be storing the desired combination of keys to answer the report query in a unique sorted set as well. In that case we would not need to perform any intersection operation over ad spaces but only set unions over time. Thus, for the previous example, if we stored a-priori the sorted set "**M:P:W**$(time)$" with the corresponding values "**m:p:w**$_{clicks}$" and their scores inside, we could answer the

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-242-4

previous question by simply performing a union operation over the time period as follows:

$$\textbf{Q'} = \bigcup_{d=date_1}^{date_2} \textbf{M:P:W}(d) \quad (7)$$

As the union operation automatically performs an addition of the scores, after it finishes we just need to look at the score associated to the "$\textbf{m:p:w}_{clicks}$" combination we are interested in. This approach is considerably faster and simpler, but it is more expensive in terms of resources as more sorted sets need to be created.

C. Approach 3

A third approach to deal with this type of queries is simply naming the sorted sets, not by using generic dimension names as done by the first two approaches, but instead by naming them using the individual ids of the dimensions of interest and storing the facts in them. Hence, in the case of the above example, a sorted set could be named as $\textbf{M:P:W}(\mathbf{m}, \mathbf{p}, \mathbf{w}, time_slot)$, and in it we can store "clicks (with score), views (with score)", where the score in each value represents the total number of clicks and views, respectively, accumulated during the period of time defined by time_slot for that unique combination of dimension ids (**m** and **p**). Furthermore, in this case it makes even more sense to keep only the keys of the deep-most branches and their combinations, as the less granular queries can be answered by performing simple union operations over their child sorted sets:

$$\bigcup_{d=date_1}^{date_2} \textbf{M:P:W}(\mathbf{m}, \mathbf{p}, \mathbf{w}, d) \quad (8)$$

The main difference with the first approach is that in this case it is not necessary to look for the specific combination of keys inside the resulting set but this output set will readily provide the value of interest. The disadvantage is that it creates considerably more sorted sets than in the first two approaches.

D. Non-aggregate events

In all of the above examples, sorted sets and union operations work well since clicks and views correspond to events that are aggregate statistics. But, this is not true for all the cases, like unique users for instance. So, how can we answer a query like: "what is the number of unique users that clicked on campaign resource **r** from campaign **c** ran by advertiser **a** between $date_1$ and $date_2$?". We cannot simply add the number of daily unique users throughout the whole comprehended period, because a given unique user could access the system at two or more different time slots. For this case, we do not actually need neither sorted sets nor scores but normal sets defined, for instance, as in Approach 3, with the only difference that instead of the basic keys, we can

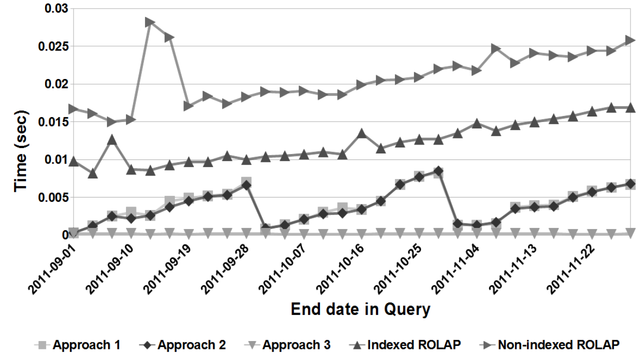

Figure 3. Performance comparison for Query #1.

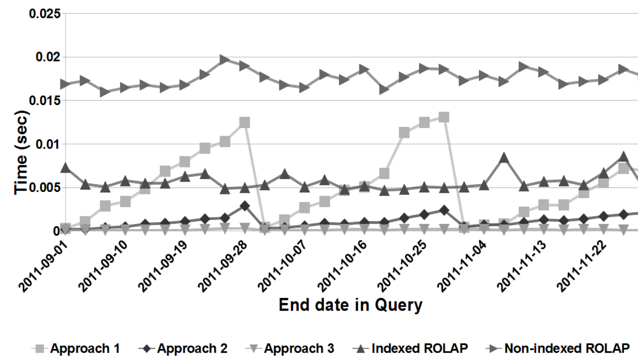

Figure 4. Performance comparison for Query #2.

directly store the non-aggregate statistic of interest (user_id in this case) inside sets like the one described below.

$$\textbf{Users:A:C:R}(\mathbf{a}, \mathbf{c}, \mathbf{r}, time_slot) = \{user_1, \ldots, user_n\} \quad (9)$$

The example set above stores the unique users detected per time_slot (1 hour in this case) for the combination campaign resource **r**, campaign **c** and advertiser **a**. Thus, whenever there is a new view for that combination of dimension ids, we simply store the user_id into the set. If the user_id is already an element in the set then no change takes place. If the user_id is not an element in the set, it is added to it. To answer the example query stated above we only need to get the cardinality of the resulting set **U** from the union operation over the whole time period as stated below:

$$|\mathbf{U}| = \left| \bigcup_{d=date_1}^{date_2} \textbf{Users:A:C:R}(\mathbf{a}, \mathbf{c}, \mathbf{r}, d) \right| \quad (10)$$

V. Tests and Results

The proposed system was implemented on Redis 2.4.2 for a real ad server application running on Ruby on Rails 2.3.10 and MySQL [3]. The models previously used as example in this paper come from the same platform but have been pruned and much simplified for the sake of clarity.

A. Experimental settings

To assess the performance of our three proposed approaches, we present a set of seven sample queries for our ad server application, which cover many relevant cases involving multidimensional data so that other specific queries can be easily derived from them.

- **Query 1:** What is the total number of clicks and views between any two dates $date_1$ and $date_2$ for campaign resource **r** from campaign **c** owned by Advertiser **a** and Manager **m**?
- **Query 2:** What is the total number of clicks and views between any two dates $date_1$ and $date_2$ on the advertising space **s** owned by Publisher **p** and Manager **m** when shown on applications?
- **Query 3:** What is the number of clicks and views between any two dates $date_1$ and $date_2$ for campaign resource **r** from campaign **c** owned by Advertiser **a** when shown in web sites on the ad space **s** owned by Publisher **p** and Manager **m**?
- **Query 4:** What is the number of clicks and views between any two dates $date_1$ and $date_2$ for campaign resource **r** from campaign **c** owned by Advertiser **a** when shown on all ad spaces of all websites owned by Publisher **p** and Manager **m**?
- **Query 5:** What is the number of clicks and views between any two dates $date_1$ and $date_2$ for the whole campaign **c** owned by Advertiser **a** when shown in applications on the ad space **s** owned by Publisher **p** and manager **m**?
- **Query 6:** What is the number of unique users that clicked on any campaign resource from campaign **c** owned by Advertiser **a** when shown on applications in ad space **s** owned by Publisher **p** and Manager **m**?
- **Query 7:** What is the number of unique users that clicked on all campaign resources owned by Advertiser **a** when shown on all ad spaces owned by publisher **p** and manager **m**?

The data used for all queries correspond to real production data taken from September 1st, 2011 through November 30th, 2011, i.e. exactly three months.

In a first performance measurement attempt, an open-source ROLAP database based on a *Pentaho's Mondrian* server [4] running on top of the MySQL database holding the ad server application's data was set up and tested. Its performance, however, was rather slow when answering the seven example queries taking in average between one and three minutes. In view of the situation, an optimized ROLAP database was setup by providing the underlying MySQL database with pre-computed tables holding all the measures of interest (clicks and views) per hour and, furthermore, either provide or not the MySQL data with tailored indexes to answer each of the seven queries. This optimized MySQL database provided to Mondrian is denoted as *optimized ROLAP* and it is the benchmark against which our approach is compared.

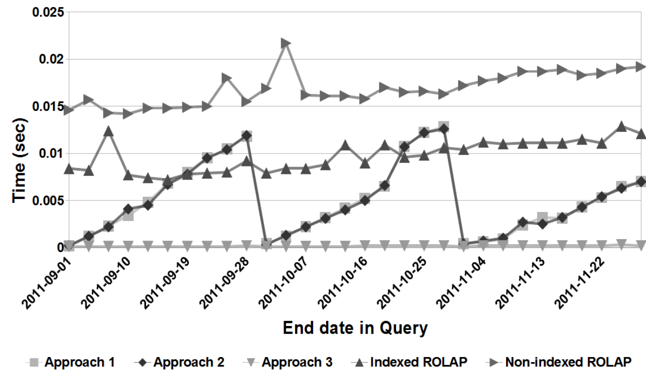

Figure 5. Performance comparison for Query #3.

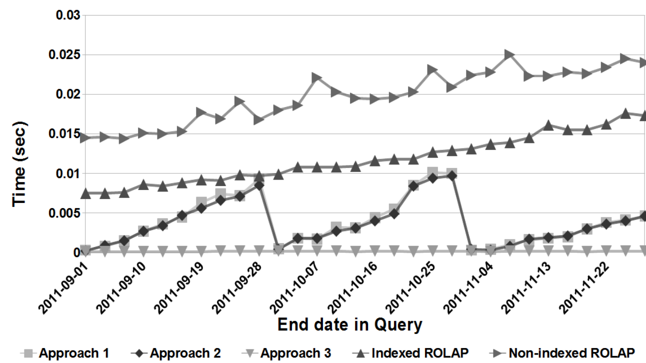

Figure 6. Performance comparison for Query #4.

One of the most relevant Redis functionalities is the policy for automatic deletion of keys in the database. Our system should only keep sorted sets that are frequently consulted or queried, and at the same time automatically remove those that are seldom seen or used. Fortunately, Redis provides a set of policy options to automatically remove keys so that the amount of memory allocated to the databases is never exceeded. In our implementation we have set the *least used resource* policy so that when the system reaches its maximum allocated memory, it can start removing the least used sorted sets from the database.

B. Experimental Results

Briefly, the first five queries (Figures 3 through 7) correspond to aggregate statistics (clicks and views) while the last two (Figures 8 and 9) are associated with non-aggregate statistics (unique users). For the former five the performance of the three approaches described in Sections IV-A, IV-B and IV-C is measured and compared with the *optimized ROLAP* (with indexed and non-indexed MySQL data) with pre-computed tables holding all the measures of interest per hour. For the last two queries, the performance of the non-aggregate statistics method described in Section IV-D is also

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-242-4

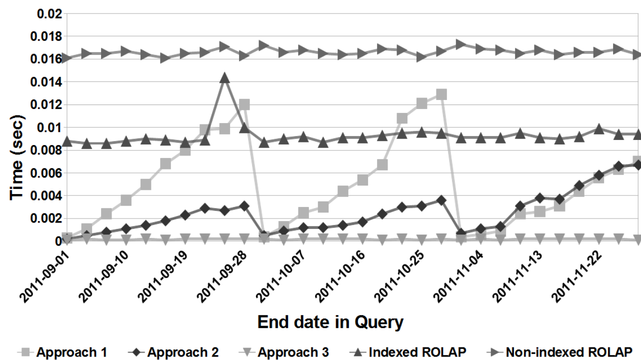

Figure 7. Performance comparison for Query #5.

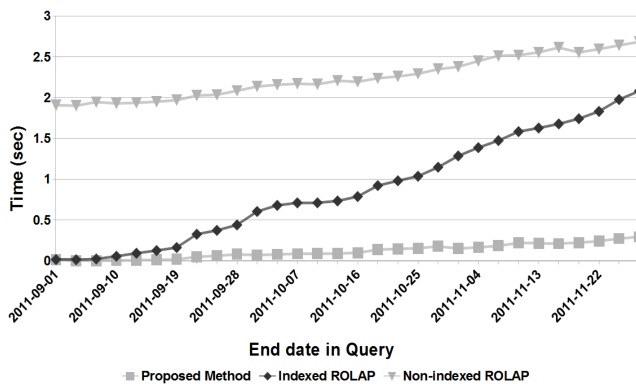

Figure 9. Performance comparison for Query #7.

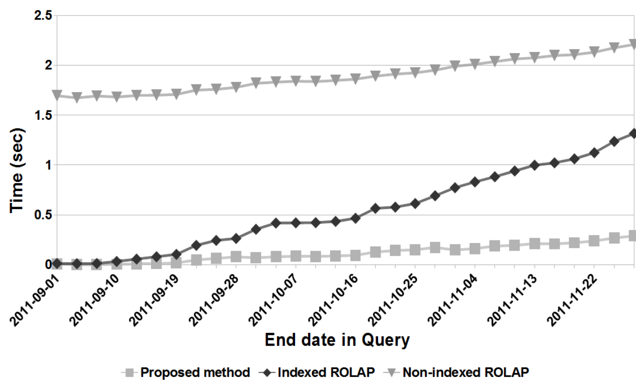

Figure 8. Performance comparison for Query #6.

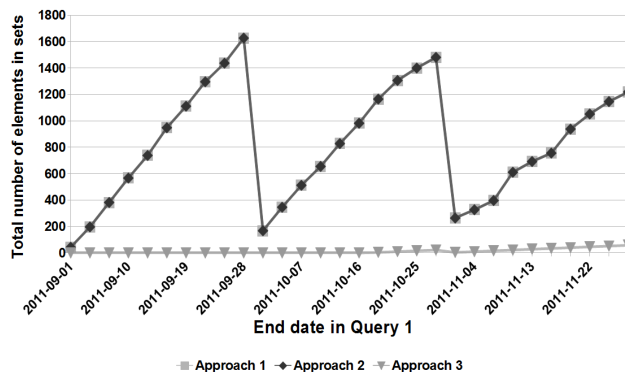

Figure 10. Number of elements in sets for Query 1.

benchmarked against the same *optimized ROLAP*.

The maximum resolution chosen for our application is one hour. Thus, the sorted sets of the three proposed approaches are named using the time dimension elements hour, day, month and year. That means we can always access pre-computed aggregate data in those time scales for any combination of dimension ids used in our sorted sets. From Figures 12 and 13, one can see that the ratio among the three proposed approaches in terms of both memory-usage and number of created sorted sets in Redis is roughly 1:3:6, for approaches 1, 2 and 3, respectively. Regarding the execution time for the seven sample queries, first of all it is important to remark that the abrupt falls in the execution time every 30 days are due to the use of our pre-computed aggregate monthly data in our system. It can be observed in the results that our proposed approaches 2 and 3 are faster than appropriately indexed MySQL in all cases except in the case of Query 3. This is because Query 3 corresponds to the most granular query in our system as it involves all single nodes in the ORM from the root to the bottom leafs in all branches. As such, it is required to perform a union operation on large sets of per-day aggregate data with hundreds of elements each. Despite this, we can see that, even in case of Query 3, our system outperforms *properly indexed optimized ROLAP* most of the times. In the context of this paper, *properly indexed* means with indexes tailored for the incoming queries.

It is interesting to observe that in the case of queries 6 and 7 for non-aggregate statistics, the proposed approach is more than 10 times faster than non-indexed *optimized ROLAP*, and 5 times faster than indexed *optimized ROLAP* for three months of data. For this approach, however, the system needs to create a large amount of sets, making the system inefficient from a memory usage point of view. Figures 10 and 11 show the total number of set elements for approaches 1, 2 and 3 when answering the first and second queries. As it can be seen in Figure 10, the number of elements in approaches 1 and 2 is exactly the same for Query 1, but substantially differ in Query 2, as shown in Figure 11. This is because Query 1 involves a combination of dimensions directly included in the basic set of keys, while for answering Query 2 new sets must be created in Approach 1. These new sets are created by union operations over the basic-key sets, as the combination of dimensions requested in the query involves only partial branches from the relational tree. This also explains the execution time of Approach 1 being considerably larger than Approach 2 in queries 2 and 5 (Figures 4 and 7, respectively).

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-242-4

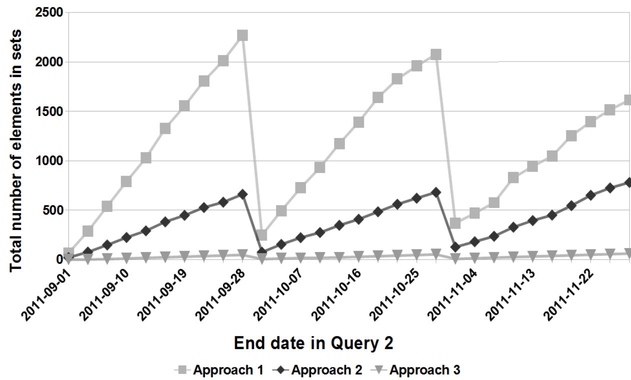

Figure 11. Number of elements in sets for Query 2.

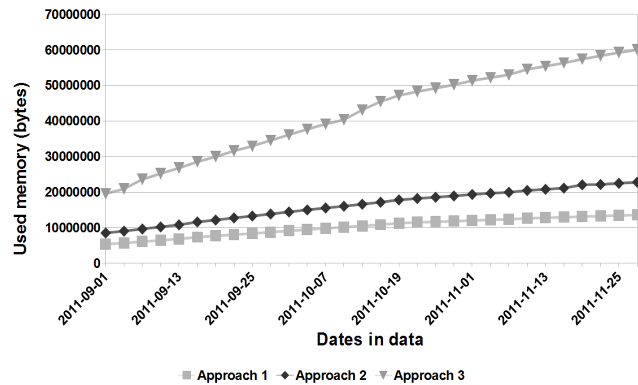

Figure 12. Memory usage.

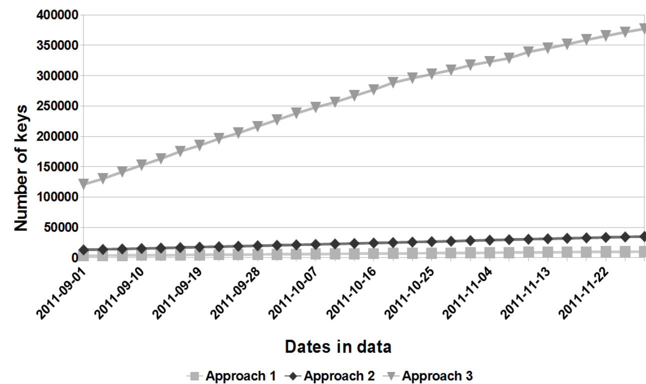

Figure 13. Number of keys.

C. *Discussion of Results*

As expected, the results show that Approach 3 is the fastest among all three (in all cases its execution time falls between 100 and 200 microseconds), but at the same time it is very inefficient in terms of memory: it consumes thrice as much as Approach 2 and six times as Approach 1. On the other hand, Approach 2 uses only twice the memory of Approach 1 but significantly outperforms it in terms of query execution time for all sample queries. Hence, Approach 2 is a much better performance compromise than Approach 1 and a very good alternative in comparison with Approach 3 when the memory usage is a big issue. That does not mean, however, it is impractical to use Approach 3 or Approach 1 in all kind of scenarios since as it can be observed in Fig. 12 the consumption of memory increases linearly with the amount of stored data for the three approaches. Thus, Approach 3 can be used in applications that feature a small number of RDB tables - i.e. a small number of sorted sets in Redis - with a lot of data inside while Approach 1 becomes attractive in applications where very granular queries are common since that means full-path queries in the proposed Redis data structure would be used more often.

Our results show that the speed of MySQL considerably improves after appropriately indexing the database on different dimensions to match those of the concerning query. However, it is well known that it is inefficient to keep many composite indexes especially on databases subject to frequent INSERTs, UPDATEs and DELETEs (as in our case) and furthermore, it is impractical to keep composite indexes for all possible multidimensional queries. Problems related to multi-column indexes in relational databases are well documented in the literature [11], but these topics are beyond the scope of this paper. A big advantage of the proposed system is that it does not have to pre-compute any sorted set but it creates them on the fly as the queries arrive following one of the described approaches. That means no further delay is introduced by our system in order to build the sorted sets as it works on the caching layer from the application's viewpoint and no data has to be transferred from the RDB to the proposed system in a cold start.

VI. Conclusions and Future Work

This paper has proposed and analyzed a new method to map a relational database into sorted sets on a key-value store database in order to create OLAP-like reporting systems. The four approaches described to map relational databases and their OLAP facts into sorted sets in key-value store databases for both aggregate and non-aggregate statistics are easily applicable to any conventional RDB structure. Our implementation in Redis shows that the proposed system is fast, surpassing properly indexed MySQL, to answer complex multidimensional queries for both aggregate and non-aggregate data. Furthermore, the flexibility of the database schema and the presence of automatic key deletion policies make the implementation very efficient. In our view the proposed scheme opens an innovative way to deal with complex data visualization and reporting systems based on multidimensional data using key-value store databases. Future work will aim at better algorithms to tune the system, studying alternatives for distributed systems based on other key-value data stores and reduce the amount of memory used by sets related to non-aggregate data.

References

[1] Druid. [Accessed Jul 13, 2012] http://www.metamarkets.com.

[2] Java programming language. [Accessed Jul 13, 2012] http://www.java.com.

[3] Mysql. [Accessed Jul 13, 2012] http://www.mysql.com.

[4] Pentaho mondrian project. [Accessed Jul 13, 2012] http://mondrian.pentaho.com.

[5] Python programming language. [Accessed Jul 13, 2012] http://www.python.org/.

[6] Qlikview. [Accessed Jul 13, 2012] http://www.qlikview.com.

[7] Redis. [Accessed Jul 13, 2012] http://www.redis.io.

[8] Ruby programming language. [Accessed Jul 13, 2012] http://www.ruby-lang.org.

[9] What is object/relational mapping? [Accessed Jul 13, 2012] http://www.hibernate.org/about/orm.

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

[11] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD ’08, pages 967 – 980, New York, NY, USA, 2008. ACM.

[12] A. Abelló, J. Ferrarons, and O. Romero. Building cubes with mapreduce. In *Proceedings of the ACM 14th international workshop on Data Warehousing and OLAP*, DOLAP ’11, pages 17 – 24, New York, NY, USA, 2011. ACM.

[13] G. Alonso, D. Kossmann, and T. Roscoe. Swissbox: An architecture for data processing appliances. In *CIDR*, pages 32 – 37. www.crdrdb.org, 2011.

[14] L. Bonnet, A. Laurent, M. Sala, B. Laurent, and N. Sicard. Reduce, you say: What nosql can do for data aggregation and bi in large repositories. In *Database and Expert Systems Applications (DEXA), 2011 22nd International Workshop on*, pages 483 – 488, September 2011.

[15] P. Brezany, Y. Zhang, I. Janciak, P. Chen, and S. Ye. An elastic olap cloud platform. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 356 – 363, December 2011.

[16] Y. Cao, C. Chen, F. Guo, D. Jiang, Y. Lin, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. Es2: A cloud data storage system for supporting both oltp and olap. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 291 – 302, April 2011.

[17] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39:12 – 27, May 2011.

[18] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1 – 4:26, June 2008.

[19] C. Chen, G. Chen, D. Jiang, B. C. Ooi, H. T. Vo, S. Wu, and Q. Xu. Providing scalable database services on the cloud, 2010.

[20] E. F. Codd, S. B. Codd, and C. T. Salley. Providing olap to user-analysts: An it mandate. *Ann ArborMichigan*, page 24, 1993.

[21] A. Cuzzocrea, I.-Y. Song, and K. C. Davis. Analytics over large-scale multidimensional data: the big data revolution! In *Proceedings of the ACM 14th international workshop on Data Warehousing and OLAP*, DOLAP ’11, pages 101 – 104, New York, NY, USA, 2011. ACM.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205 – 220, October 2007.

[23] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37:29 – 43, October 2003.

[24] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1:1496 – 1499, August 2008.

[25] D. Kossmann, T. Kraska, S. Loesing, S. Merkli, R. Mittal, and F. Pfaffhauser. Cloudy: a modular cloud storage system. *Proc. VLDB Endow.*, 3:1533 – 1536, September 2010.

[26] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44:35 – 40, April 2010.

[27] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD ’09, pages 165 – 178, New York, NY, USA, 2009. ACM.

[28] L. Qin, B. Wu, Q. Ke, and Y. Dong. Saku: A distributed system for data analysis in large-scale dataset based on cloud computing. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, volume 2, pages 1257 – 1261, July 2011.

[29] Y. Wang, A. Song, and J. Luo. A mapreduce merge-based data cube construction method. In *Grid and Cooperative Computing (GCC), 2010 9th International Conference on*, pages 1 – 6, November 2010.