Row Key Designs of NoSQL Database Tables and Their Impact on Write Performance

Conference Paper · February 2016

DOI: 10.1109/PDP.2016.84

3 authors:

Eftim Zdravevski
Ss. Cyril and Methodius University in Skopje
157 PUBLICATIONS **1,430** CITATIONS
SEE PROFILE

Petre Lameski
Ss. Cyril and Methodius University in Skopje
102 PUBLICATIONS **928** CITATIONS
SEE PROFILE

Andrea Kulakov
Ss. Cyril and Methodius University in Skopje Macedonia
85 PUBLICATIONS **763** CITATIONS
SEE PROFILE

Row key designs of NoSQL database tables and their impact on write performance

Eftim Zdravevski*, Petre Lameski†, Andrea Kulakov‡
Faculty of Computer Science and Engineering
Ss.Cyril and Methodius University, Skopje, Macedonia
Email: *eftim.zdravevski@finki.ukim.mk, †petre.lameski@finki.ukim.mk, ‡andrea.kulakov@finki.ukim.mk

***Abstract*—In several NoSQL database systems, among which is HBase, only one index is available for the tables, which is also the row key and the clustered index. Using other indexes does not come out of the box. As a result, the row key design is the most important thing when designing tables, because an inappropriate design can lead to detrimental consequences on performances and costs. Particular row key designs are suitable for different problems, and in this paper we analyze the performance, characteristics and applicability of each of them. In particular we investigate the effect of using various techniques for modeling row keys: sequences, salting, padding, hashing, and modulo operations. We propose four different designs based on these techniques and we analyze their performance on different HBase clusters when loading HDFS files with various sizes. The experiments show that particular designs consistently outperform others on differently sized clusters in both execution time and even load distribution across nodes.**

***Keywords*—NoSQL, HBase, Hadoop, table design, row key, primary key, clustered index**

I. Introduction

We live in a time in which people are connected on the Internet and the expectation is to find useful and personalized results instantaneously. To address this demand, the focus on many companies has become to deliver targeted information (e.g. recommendations or online ads), and their ability to achieve this directly influences their success. In the past, companies had the liberty to decide which data to store, keeping in mind the associated resources for it. Now there is an increasing need to store and analyze all data that can be collected, so the company can be competitive on the market. The results of such analysis when integrated in other parts of the systems can boost the data generation further. It is important to realize that the need for more data continues to increase with the development of new machine learning algorithms. Different companies have followed various ideas to solve the Big Data challenge. One traditional approach is to increase the processing power and it is effective up to a certain point. Nevertheless, for web-scale applications, to which we are accustomed nowadays, hardware scaling is not as effective. After publishing the paper describing Google's file system [1] in 2003, their approach to MapReduce [2] in 2004, and the concept of Big Table [3] in 2006, the paradigm of parallelization and distribution of computation has become popular. Since then, many open-source projects and companies have followed similar approaches. Systems like Hadoop [4, 5] now enable to gather and process petabytes of data. It is a framework that is used by industry leaders like Yahoo, Facebook, Ebay, Adobe, StumbleUpon, etc [6].

Be that as it may, the scalable performance of Hadoop does not come out of the box, per se. In order to use the facilities of Hadoop and its subsystems like HBase and HDFS to store and analyze huge volumes of data, an appropriate design is needed. A bad design can produce devastating performance of a system. In contrast, a good design can result in reduction of the cluster size and thus the cost of the system, while boosting the performance. This is especially true for the write performance of HBase tables, which depends mostly on the design of their row key. The importance of this topic is highlighted in all HBase books. Also some papers like [7] address this issue by employing a hash as part of the row key.

In this paper we analyze the performance, characteristics and applicability of different row key designs. Specifically, we employ various techniques for modeling row keys: sequences, salting, padding, hashing, and modulo operations. We propose four different designs based on these techniques and we analyze their performance on different HBase clusters when loading HDFS files with various sizes.

The rest of this paper is organized as follows. In section II we provide an overview of HBase as a representative of NoSQL databases. Then, in section III we describe four different row key designs. Afterwards, in section IV we present the comparative results of the different row keys designs and we discuss the advantages and drawbacks of each of them. Finally, this paper ends with section V where we make some conclusions and point some directions for future work.

II. HBase: a NoSQL database

HBase is an open source, non-relational, distributed database modeled after Google's BigTable. It runs on top of HDFS (Hadoop Distributed Filesystem), providing BigTable-like capabilities for Hadoop [4]. It is a platform for storing and retrieving data with random access. HBase is a NoSQL (Not Only SQL) database that can store structured and semistructured data [8]. It is designed to work on a cluster of computers, where each node in the cluster adds up to the computational, memory and storage capacity of the whole system. To facilitate this HBase splits tables in regions, which are hosted by servers called Region Servers.

Generally, given that Google's Distributed File System, Big Table and Map Reduce were the source of inspiration for Hadoop, HDFS and HBase, there are equivalent concepts in both ecosystems. Therefore, the ideas presented in this paper are applicable to other Big Table-like systems such as Amazon's DynamoDB [9] and Cassandra [10].

The design and the use cases of HBase significantly differ from the Relational Database Management Systems (RDBMS), and consequently the approach for implementing applications that use HBase is also different. In short, before creating a table one has to think about its potential usage patterns, the expected volumes of data, and even the data distribution of the row key. The row keys of HBase tables determine the performance when interacting with them. According to [11, 8], when designing HBase tables, the row key is the single most important thing and should reflect the expected access pattern. The reasons for that are: the fact that regions serve a range of rows based on the row keys and are responsible for every row which is in that range, and the fact that HFiles store the rows sorted by row key on disk. In relational databases, whenever the usage pattern of a table is changed, we can add additional index to address this, or we can drop obsolete indexes. All of this can be done without any downtime. On the contrary, HBase indexes only on the row key, so the only way to access data is by using it. If we want to access a row by some information contained in other columns, then we will need to scan a range of rows, or even the entire table.

There are various techniques to design row keys that are optimized for different access patterns, as explained in [11, 8]. In these books there are plenty of useful examples and use-cases that illustrate the importance of the row key design. Nevertheless, they lack an extensive analysis of row keys that can offer balanced load on the cluster, while being generic and applicable for a variety of scenarios. In [7] the authors propose the usage of the MD5 hash function when generating row keys. Nevertheless, the row key design is specific to a particular application. Moreover, there is no analysis of the distribution of row keys, nor of the execution times compared to other row key designs.

In the following section we describe four different row key designs and afterwards in section IV we analyze their effect on communication costs and uniform load across nodes in the HBase clusters.

III. Generic row key designs

As pointed out previously, the row key design of HBase tables has the greatest impact on the performance of the system. Monotonically increasing row keys (e.g. when using a timestamp) cause only one of the table regions to be pounded when inserting data, while others to be idle. In [5, 11] are described scenarios where an import process can have dreadful performance caused by monotonically increasing keys. The authors of these books recommend avoiding such keys by adding either randomized prefixes or some sort of pseudo-random prefixes in the row key. In this section we describe four different approaches to achieve this. For each of them we point out their limitations, drawbacks and advantages. Some of their properties are validated by the experiments and performance evaluations described later in section IV.

Aiming to generate row keys that are lexicographically sortable, we use a User Defined Function (UDF) for padding numbers with zeros. Being able to have an expected ordering of the row keys aids the design of HBase tables because it can enable choosing good split points for pre-splitting, or identify regions that might become "hot". Hot regions have significantly more load than others in an extended period of time. By using this UDF we are able to combine multiple integers in a slightly more complex structures (i.e. tuples) that are lexicographically sortable in binary or string representation. Also in three of the row key designs we use the MD5 hashing function, therefore we describe it in subsection III-A. Then, in the following subsections we describe each of the proposed row key designs.

A. MD5 hashing function

The MD5 message-digest algorithm was proposed in [12]. The algorithm takes as input a message of arbitrary length and produces as output a 128-bit "fingerprint" of the input. It is conjectured that it is computationally infeasible to produce two messages having the same fingerprint, or to produce any message having a given prespecified target fingerprint. The MD5 algorithm is intended for digital signature applications, where a large file must be "compressed" in a secure manner before being encrypted with a private (secret) key under a public-key cryptosystem.

Aside for cryptographic applications, MD5 hashing functions can be used for other purposes as well. One of them is to generate an integer hash value from an arbitrary argument. In the context of designing row keys, our hypothesis is that MD5 hash values can be used as pseudo-random numbers. Nevertheless, the range of values generated by the MD5 hash function is quite wide, and is impractical for direct usage. More memory is needed for holding the values and the range is not as descriptive. We can amend this by calculating the modulo of the MD5 hash value and the range size. In Python this can be done by the code shown at listing III-A. The input argument s is of arbitrary data type, which is converted into a string with the function $force\_string$. The argument max_value is used as a divisor in the modulo operation, thus limiting the range of integers that can be returned by this function to $0..max\_value - 1$.

```
1 def MD5(s, max_value):
2     md5 = hashlib.md5()
3     md5.update(force_string(s))
4     digest = md5.hexdigest()
5     number = int(digest, 16)
6     return number % max_value
```

Listing 1. Python UDF for generating MD5 hash values in the range 0..max_value-1

In order to confirm that the distribution of values in that range is pseudo-random, we have performed extensive experiments. We have used different values for the parameter max_value, while providing sequential integers and arbitrary strings with varying length to the MD5 function. We have provided 4 million different arguments to the MD5 functions while using the values 1000, 10000, 100000 and 1000000 for the max_value parameter. The results showed that indeed the distribution of returned values by the MD5 function was uniform. This experiment confirmed that the MD5 function can be used for generating pseudo-random integers. Fig. 1 shows the histogram of MD5 modulo 1000 for 4 million sequential integers. Given that we have generated 4 million numbers in the range $0..999$, ideally each of the 1000 distinct numbers would be generated 4000 times. Nonetheless, in our experiment each distinct number was generated from 3810 to

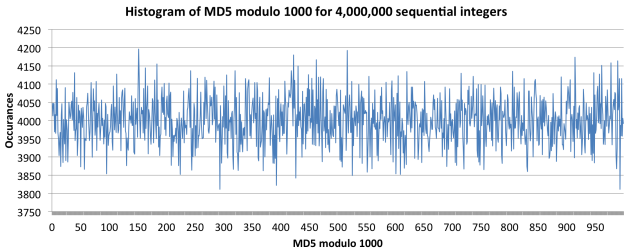

Fig. 1. Histogram of MD5 modulo 1000 for 4 million sequential integers

4199 times with standard deviation of 61.89. It is evident that the generated numbers with the MD5 function have similar properties as random numbers that would be generated in the same range.

B. Pre-splitting HBase tables

If no split points are provided at table creation time, HBase creates only one region for the table. The reason for this is because it cannot possibly know how to create the split points within the row key space before having data in the table. Making decisions about good split points should be based on the distribution of the keys in the data. So when a table that has one region gets large enough, it will be automatically split. Nevertheless, this is an expensive operation and has several drawbacks. First, the automatically chosen split point might be optimal at the time of the split, but not be the best choice in general. This is due to the fact that the expected table growth or the distribution of row keys in the future are not taken into consideration. Second, when a region is split into two, both of them will be assigned to the same Region Server. Moving one of the regions to another Region Server that is less loaded needs to be done manually. Third, there is an overhead of the entire operation and the time when the split is going to occur cannot be precisely predicted, which can slow down applications that use the table at the time of the split.

HBase provides tools for a process called pre-splitting, which enables to create a table with a particular number of regions by supplying the split points at the table creation time. This is a convenient way to make sure that the table has the desired number of regions even before it has any data. Moreover, HBase will try to distribute the table regions uniformly, so each Region Server is responsible for approximately the same number of regions. For instance, if a table is split into R regions and we have RS region servers, then each Region Server will get R/RS regions when $R \geq RS$. When $R < RS$ then R of the region servers will be responsible for one region, thus some of the region servers would not be used for this table. The commands for creating tables with split points or for manually splitting an existing HBase tables are explained and illustrated with examples in [8].

Pre-splitting ensures that the initial load is more evenly distributed throughout the cluster, and it should be always considered when the row key distribution is known upfront. If it is not known, then there is a risk of creating regions that will not be loaded as expected due to data skew. If the initial set of region split points is chosen poorly, the load will be heterogeneous, which will in turn limit the cluster's performance. This stresses another aspect of the importance of the row key design, namely having predictable distribution, which is the focal point of this paper.

During the next subsections we assume that each row has a unique integer identifier, id. Later on, in subsection III-G, we discuss how the proposed row key designs can be generalized for other cases when a unique identifier is not available. Additionally, we choose to add a prefix in the row key design that is also an integer, so we can easily estimate the difference between prefixes. The proposed row keys have the following format: $(prefix, padded\_id)$. We have chosen this design because it allows the row keys to be lexicographically sorted. In order to achieve that, the id and the $prefix$ are padded with zeros to a predefined number of digits. Another convenient property of this design is that the row key can be automatically formatted as a tuple by the Pig Latin scripting language [13], which we use for conducting our experiments. If we want to save space we could omit the parenthesis and the coma from the row key. Keeping in mind that the number of digits for the $prefix$ and the id is fixed, the row key integral parts can still be decoded. For a given maximum value of the prefix MP, and a given number of regions R, where $MP > R$, the number of regions can be calculated as MP/R. In the interest of having R regions, we need to specify $R-1$ split points when creating the table. For the proposed row key format the split points can be: $MP/R, 2\times MP/R, 3\times MP/R, ..., (R-1)\times MP/R$. With these split points the i-th region, $i = 1..R$, will handle the row keys that have a $prefix$ in the range $[(i-1) \times MP/R, i \times MP/R)$. Given that each region will handle exactly MP/R prefixes, it would be ideal if all regions are equally loaded. Aiming to achieve even load on each region, we propose different designs of the $prefix$ part of the row key in the following subsections.

When choosing the number of regions, it is usually good idea to take a multiple of the number of region servers in the HBase cluster. The logic for having more regions than actual nodes is because the nodes are multi-core machines, so different threads on the same node can service different regions. Also when choosing the maximum value of the prefix MP it is good for it to be a multiple of the number of regions R, so the MP/R divisions mentioned above to be an integer, which in turn makes all nodes to have equal number of prefixes. Additionally it is recommended that $MP >> R$ because the cluster can get larger over time. Choosing MP in that way will provide flexibility to make additional splits of the existing table regions based only on the $prefix$. Moreover, it will enable to distribute the data on all regions more evenly. To illustrate this, let us consider the completely opposite example in which $MP = R$. This in turn means that every $prefix$ was already used as an original split point. If we add one node to the existing cluster with R regions, we would like to give a portion of the table to the new node gets so it can do useful work. But because all MP prefixes were already used for split points, we are not able to specify a split point based on the prefix. This will force us to make a split point based on the second part of the row key, namely the id in this case. On the other hand, for the distribution of the values of the id in general we cannot make any guarantees. It means that the chosen split point that includes both the $prefix$ and the id is not optimal. However, if MP was significantly greater than R, we would be able to make additional split points that will split the table in a predictable manner.

C. Random prefix

In general, random prefixes provide best distribution of write load across region servers and consequently across nodes in the cluster [11]. Nevertheless, this approach generally provides bad performance for random reads. Let us consider a simple example where each row has monotonically increasing id. Furthermore, let the row key be designed in a way that a random number prefixed the id. The design of the row key would be as explained in the previous subsection in the format $(random\_prefix, id)$, and the table would be appropriately pre-split. This row key design is independent of the data type of the id of the row. In fact, the id can be a composed of multiple columns because the random prefix does not depend on them.

D. ID modulo prefix

The UDF shown at III-D generates a tuple which represents the row key using a modulo of the id as prefix. Here id denotes the unique identifier of the row, id_pad_digits is the number of padding digits for the id, and mod_number is the divisor argument of the modulo operation. It returns a tuple consisted of two parts: the padded modulo of the id and the padded id.

Obviously this design is only suitable for integer ids of rows and this is its main drawback. Other drawbacks are discussed in subsection III-G.

```
def rowkey_mod_prefix(id, id_pad_digits,
    mod_number):
    mod_number_digits = num_digits(mod_number)
    prefix = int(id) % int(mod_number)
    return (pad_number(prefix,
        mod_number_digits), pad_number(int(id),
        id_pad_digits))
```

Listing 2. Python UDF for generation of row key based on modulo of the ID

E. ID MD5 modulo prefix

The UDF shown at III-E generates a tuple which represents the row key using a modulo of the MD5 hash of the id as prefix. Here id stands for the unique identifier of the row, id_pad_digits is the number of padding digits for the id, and mod_number is the divisor argument of the modulo operation. It returns a tuple consisted of two parts: the padded modulo of the MD5 of the id and the padded id.

This row key is applicable for arbitrary types of the id. In fact, even the function presented in listing III-E does not specify the type of the id, while the MD5 function accepts arbitrary data types that are converted to string before calculating the MD5 hash and the modulo. As it turns out, demonstrated by the experiments described in section IV, this row key is the best option for both evenly distributed load during writes across the cluster, while also being suitable for lookups during reads of random rows.

```
def rowkey_md5mod_prefix(id, id_pad_digits,
    mod_number):
    mod_number_digits = num_digits(mod_number)
    prefix = MD5(id, mod_number)
    return (pad_number(prefix,
        mod_number_digits), pad_number(int(id),
        id_pad_digits))
```

Listing 3. Python UDF for generation of row key based on modulo of the MD5 hash of the ID

F. Line MD5 modulo prefix

This UDF is exactly as the one described in the previous subsection, but instead it calculates the MD5 hash of the whole row as string, denoted as $line$ in the code. The code for this UDF identical to the one shown in listing III-E, but in to the MD5 function we pass the whole row (line) and not only the Id.

This row key design has an intrinsic anomaly that it cannot be used for random reads. Still, we have decided to test it for two purposes. First, we wanted to test if the execution time of the MD5 function degrades for quite larger arguments. Second, we wanted to test if the MD5 modulo prefix distribution is still random for arguments comprised of mixed data types. The experiments confirmed that the MD5 function is scalable to large arguments while returning numbers with random properties.

G. Generalization of the primary key

So far, we have assumed that each row has an integer id that can uniquely distinguish it. In fact, many applications have composite keys comprised of multiple columns. Obviously a good generic row key design should also facilitate such scenarios, as well. Let us consider a composite primary key (i.e. id) that has the following format (A, B, C, D) where A,B,C and D are the columns of arbitrary data types that comprise it. For such key, obviously the prefix described in subsection III-D is not applicable because it cannot calculate the modulo of a non-integer id like this one. All other designs are still applicable. First, the random prefix, described in subsection III-C, only contains random numbers in a given range that does not depend of the type of the id. Then, the design based on the modulo of MD5 of the id, described in subsection III-E, will treat the key (A, B, C, D) as a string, calculate its MD5 hash, and consequently calculate a modulo of it. Finally, the design described in subsection III-F does not depend on the id at all, it rather uses the whole row as an argument to the MD5 function. So all advantages and drawbacks of these types of prefixes also generalize to arbitrary types of the primary key of the row.

Another key point is that in some cases it is not suitable to build the prefix based on the whole composite key. Namely, instead of passing the composite id as an argument to the MD5 function for generation, it might be more suitable to use only a part of the id as basis for the prefix. To illustrate this, let us consider the case when we need to store some actions for each user on a website, which is very common use case in many business applications. In this case, the id of an action (i.e. event) might be structured as $(userID, timestamp)$. We can still use this composite key as an argument to the proposed row keys. If we do that, then the events of the same user will be scattered on potentially all regions in the table. However, this might be very inconvenient for various reasons. We might want to do some analysis or perform some machine learning based on all events of a particular user. Gathering all events of a user entails communication and computation costs. This

can be alleviated by using only the userID and not the whole composite id for generation of the prefix. The benefit of doing this is that all events of the user will be lexicographically sorted. On the other hand, this design endows unbalanced load if the distribution of events per userID is skewed.

IV. Results and discussion

In this section we present the results from our experimental evaluation of the proposed row keys. This section is structured in several subsections, which describe and discuss specific aspects of the experimental results. In the following subsections IV-A and IV-B, we describe the cluster configuration and the dataset used for the experiments. Thereupon, in the next subsections IV-C, IV-D and and IV-E, we describe and discuss the effects of the cluster size, row key designs and number of files and their sizes.

A. Environment and cluster configurations

In the interest of evaluating the various row key designs, we have used a cluster that was deployed on-premises at the Faculty of Computer Science and Engineering (FCSE) at the Ss.Cyril and Methodius University, Skopje, Macedonia. It had a total of 65 nodes, each of them an Intel Xeon Processor E5640 with 12M Cache, 2.66 GHz, 24 GB RAM, 4 cores and 8 threads. From them 55 were configured to run the following services: HBase Region Servers, HDFS DataNodes and YARN MapReduce NodeManagers. The remaining nodes were used for other Hadoop and Cloudera management services. We have started experimenting using a HBase cluster with 55 nodes (i.e. region servers) and have gradually reduced the number of active nodes. For all experiments the number of HDFS and YARN nodes was fixed to 55. Because data is read from HDFS files, the number of HDFS nodes affects the number of map tasks. HDFS source files, depending on their size and the configured block size of HDFS, are automatically partitioned on one or multiple blocks. In our experiments we have used the default block size of 128 MB. The number of YARN nodes determines how many nodes can run MapReduce applications (i.e. jobs) [14]. In our experiments the number of map tasks was always lower than the number of YARN nodes, so the cluster limit regarding the number of active applications or map tasks was never reached. We had exclusive access to the cluster and we have executed all experiments sequentially, therefore only one job was active at any time.

We have used the scripting language Pig Latin [13] for writing the MapReduce jobs that parse the flat text files and store them in HBase tables. The main reasons we prefer it over manually writing MapReduce jobs are its simplicity and speed of development. Moreover, it enables combining more complex data flows, which we have applied for other types of analysis of the same dataset that are not covered in this paper. Pig determines the number of map tasks depending on the number of partitions of the data source. If the data source is a HBase table then the number of table regions elicits the number of map tasks. When the data source is a HDFS file, the number of blocks that comprise the file and the number of HDFS nodes determines the number of map tasks. If we use more than one HDFS file, then blocks from multiple files that are on the same node might be combined in one map task. To summarize, when using HDFS files as data source in Pig Latin scripts the number of map tasks cannot be determined upfront and depends on how the files are stored in HDFS and the HDFS cluster configuration. Nevertheless, for a particular set of files on a given cluster configuration, the number of map tasks is constant.

In order to examine the effect of the number of table regions on the parallelism, we have pre-split the destination HBase tables to have 1, 2, 4, 8, 16, 32, 55, 110, 165 and 220 regions. HBase automatically distributed the table regions in such way that each HBase Region Server is servicing an equal number of regions per table. Note that for the current experiments the number of table regions limits the parallelism during the writes only, while the read parallelism and the number of map tasks are determined by the HDFS files. As for the modulo number used in all MD5-based row key designs, we have decided to use 1000000, which results in generated prefixes in the range from 0 to 999999. Likewise, for the random prefix row key, the generated numbers were in the same range. Based on the approach discussed in subsection III-B the tables were pre-split accordingly.

B. Dataset for experiments

Aiming to test the various row key designs we used the AAIA'14 [15] dataset. This dataset is a sparse matrix of 50000 rows and about 12000 columns with 0.9 % non-zero elements. Instead of using its original form, we modified the dataset files that were stored as flat text files, so each row contains all non-zero values as pairs of column id and value. Additionally, we have multiplied the dataset horizontally 80 times, after which its size became 4 million rows. We have performed other experiments on Hadoop clusters with this dataset, as discussed in [16] and [17]. This representation of the dataset in flat text file needed about 3 GB of space. We have stored the enlarged dataset in different ways. We have tried storing it in one large file, but also splitting the large file in many smaller files, as well. We have repeated the experiments for all different file sizes intending to determine the effect of the file size on the performance. Table I shows all partitioning schemes that were used. It was evident when the dataset is stored in 10, 20, 40 and 80 files the blocks of these files are not fully filled in. More importantly, for those cases even though the total number of blocks increases, the number of map tasks does not significantly increase. Interestingly, the maximum number of map tasks was 40 and was obtained when using 40 files. Somewhat counterintuitive, when using 80 files the number of map tasks reduced to 32. One logical explanation for this phenomenon is that Pig Latin combines the processing of multiple blocks of files that reside in the same HDFS DataNode in one map task. This also explains why the number of map tasks is lower than the total number of blocks when using 10, 20, 40 and 80 files.

C. Effect of cluster size on performance

Fig. 2 displays the run times for loading the dataset depending on the number of source files used, the number of regions per destination HBase table when the number of Region Servers is 55 for the different row key designs. It can be noticed that for each row key design the best execution times were when using table with 165 regions. The explanation for this is the following. Each node in the cluster is a 4 core

TABLE I. DIFFERENT PARTITIONING SCHEMES OF A 3 GB DATASET

Files	Size (MB)	Blocks	Blocks Filled (%)	Total Blocks	Total Maps
1	3070	24	99.9%	24	24
2	1535	12	99.9%	24	24
4	768	6	100.0%	24	24
8	384	3	100.0%	24	24
10	307	3	79.9%	30	25
20	154	2	60.2%	40	26
40	77	1	60.2%	40	40
80	38	1	29.7%	80	32

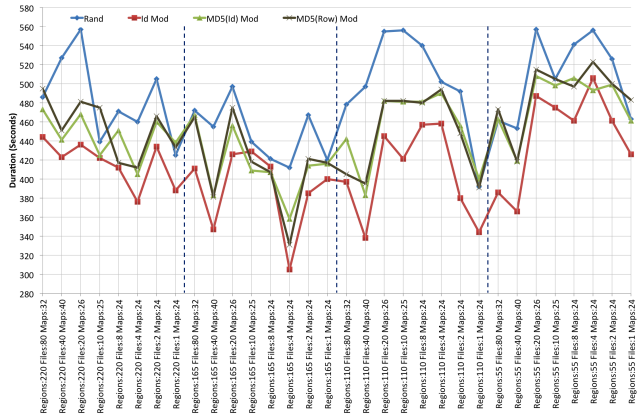

Fig. 2. Write performance for the different row key designs on a HBase cluster with 55 Region Servers, different number of regions per table and different number of source HDFS files.

machine. The 165 regions were uniformly distributed to the 55 region servers, so each of them services 3 regions of the destination table. One of the 4 cores of each node remains free to service other operating system and Hadoop processes, while the other 3 are available for servicing the write requests to the 3 regions. When using a table with 110 and 55 regions, only 2 and 1 of the cores, respectively, are utilized when servicing write requests to the 2 and 1 regions of the table that are on that node. This is less than optimal and the full resources of the cluster are utilized. On the other hand, when the table has 220 regions, all 4 cores of each node are being utilized during the write requests, but there is no core left to service the operating system and Hadoop. As a result, the operating system spends more time for task switching, which in turn slows down the write process.

D. Effect of row key designs

Regarding the execution times from the perspective of row key designs, from the chart on Fig. 2 it can be noticed that the random prefix provided worst results, especially when a particular number of source HDFS files were used. Also it is evident that the MD5 modulo prefixes always performed similar, meaning that the size of the argument of the MD5 function does not significantly increase the execution time. This fact verifies the claim that the MD5 function can be used for complex keys comprised of multiple columns with different data types. The ID modulo prefix obviously performs best on this cluster configuration.

The charts presented on Fig. 3 show the performance of the cluster during the load of the dataset stored as one file of 3 GB. There are presented 3 charts: one showing the HBase

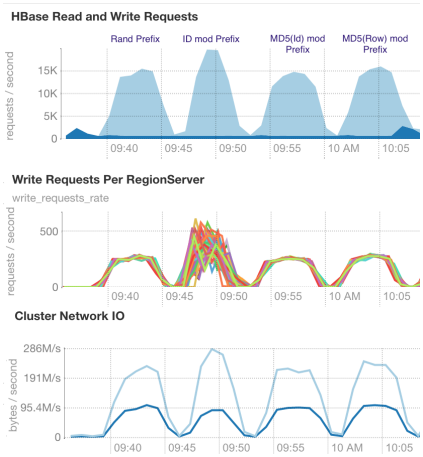

Fig. 3. Write performance for one 3GB file when storing it in HBase tables with 220 regions distributed on 55 region servers.

Read and Write Requests; one displaying the Write Requests per Region Server; and one showing the Cluster Network I/O. On each of them are shown the performances of each of the row key designs in the following order: Random prefix, described in subsection III-C, ID modulo prefix, described in subsection III-D, the ID MD5 modulo prefix, described in subsection III-E, and last the Line MD5 modulo prefix, as described in subsection III-F. Generally, it can be noted that the execution times were similar (392, 344, 400, and 391 seconds in the respective order). However, it can be noted that the ID modulo prefix produces more Read and Write requests at a given time and more network I/O. Moreover, from the second chart, where each line represents the requests per particular Region Server, it is evident that not all region servers are equally active at the same time. For all other designs all region servers are equally loaded with requests all of the time, causing the activity lines on the chart to overlap. A similar behavior can be noticed in the other configurations with 55 region servers, as discussed previously. Important to realize is that for all cases the ID modulo prefix consistently generates thee to four times more requests on region servers than the other row key designs. Even though its run times are somewhat shorter than for other designs, this behavior can be devastating and cause region servers to crash if more applications run at once for an extended period of time.

Another evidence to support the claim that the ID modulo prefix can perform inconsistently is shown on Fig. 4. The only similar execution times to the other row key designs are when only one large file is used. The reason for this is because the smaller files are actually splits of the large file, so the IDs in the rows in them are monotonically increasing in a small range. As a result, the prefix, which is actually the remainder returned by the modulo operation, is also a monotonically increasing number causing only some of the table regions to be active at a given time.

Next, on Fig. 5 is displayed the performance of another cluster configuration. We use a destination HBase table with 4 regions that are distributed on different region servers. The spike in the first block of the second chart on Fig. 5 reveals that when a random prefix is used and the number of regions is small, some region might get significantly larger load than the

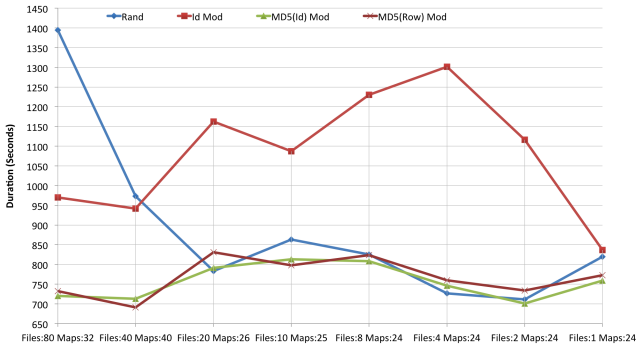

Fig. 4. Write performance for the different row key designs on a HBase cluster with 8 Region Servers and 8 regions per table and different number of source HDFS files.

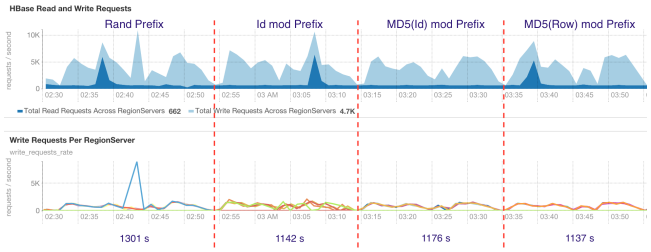

Fig. 5. Write performance for one 3GB file when storing it in HBase tables with 4 regions distributed on 4 region servers.

others. In fact, for the random prefix we have noticed similar cases while using other configurations with small number of regions. Nevertheless, given the random nature of the MD5 modulo prefixes, we acknowledge that the same situation can happen for them, as well. Such thing can pose a problem and cause a Region Server to die if it is "hot" for an extended period of time, which can happen if multiple applications run at the same time. Be that as it may, this is not a problem when only one application runs on the cluster at one point because of the limited duration of the increased load. Additionally, when using clusters with larger number of nodes the probability of one Region Server being hot for an extended period of time diminishes.

E. Effects of HDFS file sizes and modulo number

Interestingly, from Fig. 2 it can be also noticed that having more map tasks is not always better. Namely, when the dataset was split into 80 and 40 files, there were 32 and 40 map tasks, respectively. Nevertheless, when comparing the performance of the same row key and same cluster, it was evident that when using more map tasks (i.e. having smaller files) was the same or worse than when having less map tasks (i.e. when using one large HDFS file). We explain this behavior by the fact that HDFS partitions large files systematically while trying to distribute the load on the cluster uniformly. On the other hand, when using many small files their blocks are not fully filled in, so resources are wasted when storing and reading them. This is especially evident when the dataset is split into 80 small files. As it can be seen from table I, in that case the blocks are only about 30% full. In fact, from Fig. 2 it is evident that using one large file provided similar performance for all row key designs and almost always one of the best results.

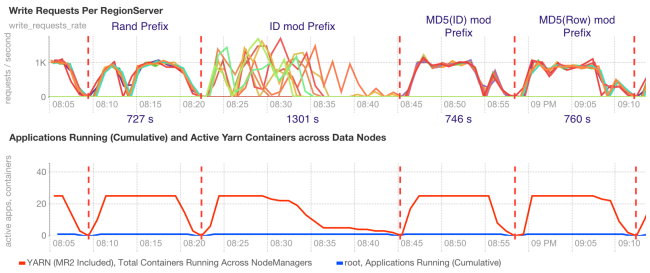

Fig. 6. Write performance for the different row key designs on a HBase cluster with 8 Region Servers and 8 regions per table and 4 source HDFS files.

For the row key designs based on the MD5 hash and the random prefix, the divisor argument of the modulo operation, which determines the range of values of the prefix, does not make any significant difference, Nevertheless, this is not the case for the row key based on the ID modulo prefix. If this value is in the same order of magnitude as the number of rows in the dataset, the prefixes generated by this row key design will not be in the full range of possible prefixes. Another key issue with this type of value of the divisor argument is that for consecutive integer ids, the prefixes are also consecutive numbers. As a result, only some regions are active at a given time. This behavior can be noticed on the charts presented on Fig. 6. Here is displayed the performance of the different row key designs on a HBase cluster with 8 Region Servers, 8 regions per table and 4 source HDFS files. The first chart shows the number of write requests per Region Server, where each line is the activity of a particular Region Server. On the second chart is displayed the number of active MapReduce jobs. All other row keys elicited an uniform load on the cluster. However, the ID modulo prefix produced bad distribution of the load throughout the lifetime of the MapReduce job. It can be noticed that in the second half of the lifetime of the MapReduce job, only a few of the Region Servers were active while servicing couple of map tasks. Therefore, the duration of the MapReduce job was almost twice as large compared to the other row key designs. The situation was very similar when using more files on the same cluster. Although this issue can be alleviated by reducing the value of the divisor in the modulo operation, we do not spending time on optimizing it. Rather, we suggest using the row key design based on the MD5 function, proposed in subsection III-E, as it overcomes these issues naturally, while being also applicable to ids of arbitrary data types.

V. Conclusion

In this paper we have discussed the importance of the row key design of HBase tables. We have pointed out the benefits of pre-split tables in order to distribute the load evenly throughout the nodes of the cluster. In order to achieve that, the row keys distribution needs to be known beforehand, which is a very strict constraint. To address this problem we have analyzed different techniques for adding prefixes in the row keys. Our hypothesis was that the prefix would have predictable distribution even if the distribution of the remaining part of the row key (i.e. the id) was skewed. For the prefix we have analyzed four different designs: random prefix, modulo of the ID, modulo of the MD5 hash of the ID and modulo of the

MD5 hash of the whole row. We have performed extensive experiments in which we loaded data in HBase tables on clusters with varying number of nodes and different HDFS file sizes. The experiments showed that the prefix that is modulo of the MD5 hash of the ID has equally well write performance as the random prefix. Furthermore, unlike the random prefix, it offers random lookups in the table because for a given ID the prefix is always the same, which keeps the row key consistent. Dissimilarly, the row keys with random prefix do not offer any facilities for random reads. When using such keys, aiming to retrieve a particular row a full table scan must be performed. The same disadvantage has the row key with prefix that is the modulo of the MD5 hash of the whole row, which is also somewhat more time-consuming to compute. Finally, the prefix that is modulo of the ID facilitates random row lookups without needing a full table scan. This design distributes the load on all nodes generally well. Nevertheless, in that regard it is worse than the other designs. Even though the total number requests per region are equal, some of the regions are hot in one point in time, and then they are idle for some period. This behavior contributes to bad parallelization and region hot-spotting because not all regions are equally active all of the time.

A general drawback of all proposed prefixes is that they lack the ability to perform scans of particular ranges of rows. For instance, if the ID is an integer, one may want to scan all rows that have an ID in a particular range. In this scenario, the random prefix and the prefix that has MD5 of the whole row would require a full table scan, regardless of how small the range is. Likewise, the other two designs do not allow using the built-in facilities of HBase to do the row scans. Nevertheless, they can enable the client application to perform as many lookups as the number of ids in the required range and then merge the results. However, HBase itself cannot perform this operation. If the client application is poorly designed, then a full table might be needed to obtain the required range of rows.

The main contribution of this paper is the detailed analysis of the different options for adding prefixes to the row keys of tables in NoSQL databases like HBase. We have recommended a generic design that facilitates uniform load on the cluster during writes and also allows random lookups. Moreover, the design based on the modulo of the MD5 hash values of the ID, can be used for a variety of applications and therefore we recommend it.

ACKNOWLEDGMENT

This work was partially financed by the Faculty of Computer Science and Engineering at the Ss.Cyril and Methodius University, Skopje, Macedonia.

REFERENCES

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[3] 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," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 15–15.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, May 2010, pp. 1–10.

[5] T. White, *Hadoop: the definitive guide*, 3rd ed. Beijing: O'Reilly, 2012.

[6] "Hadoop wiki: List of institutions that are using hadoop for educational or production uses," https://wiki.apache.org/hadoop/PoweredBy, accessed: 2015-01-29.

[7] H. Ochiai, H. Ikegami, Y. Teranishi, and H. Esaki, "Facility information management on hbase: Large-scale storage for time-series data," in *Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International*, July 2014, pp. 306–311.

[8] L. George, *HBase the definitive guide*. Sebastopol, CA: O'Reilly, 2011.

[9] 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.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

[10] E. Hewitt, *Cassandra: the definitive guide*, 1st ed. Beijing: O'Reilly, 2011.

[11] N. Dimiduk and A. Khurana, *HBase in action*. Shelter Island, NY: Manning, 2013.

[12] R. Rivest, "The MD5 message-digest algorithm," 1992. [Online]. Available: http://tools.ietf.org/html/rfc1321?ref=driverlayer.com

[13] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: The pig experience," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1414–1425, Aug. 2009.

[14] D. Miner, *MapReduce design patterns*. Sebastopol, CA: O'Reilly, 2013.

[15] A. Janusz, A. Krasuski, S. Stawicki, M. Rosiak, D. Slezak, and H. S. Nguyen, "Key risk factors for polish state fire service: A data mining competition at knowledge pit," in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, Sept 2014, pp. 345–354.

[16] E. Zdravevski, P. Lameski, A. Kulakov, B. Jakimovski, S. Filiposka, and D. Trajanov, "Feature ranking based on information gain for large classification problems with mapreduce," in *Proceedings of the 9th IEEE International Conference on Big Data Science and Engineering*. IEEE Computer Society Conference Publishing, August 2015, pp. 186–191.

[17] E. Zdravevski, P. Lameski, A. Kulakov, S. Filiposka, D. Trajanov, and B. Jakimovski, "Parallel computation of information gain using hadoop and mapreduce," in *Proceedings of the 2015 Federated Conference on Computer Science and Information Systems*, M. P. M. Ganzha, L. Maciaszek, Ed., vol. 5. IEEE, 2015, pp. 181–192.