UCLouvain

École polytechnique de Louvain

Predicting records size with neural networks in key-value stores under heterogeneous multiget workloads

Author: **Laritza Cabrera Alba**
Supervisor: **Etienne Riviere**
Readers: **Paolo Laffranchini, Ramin Sadre**
Academic year 2019–2020
Master [60] in Computer Science

Acknowledgements

I would like to thank my supervisor, Etienne Riviere for his support during this thesis, and also to Ramin Sadre and Paolo Laffranchini for taking the time to read it, as well as being part of the jury for my defense.

I would like to thank immensely my lovely husband Mitchel, for being my pillar during all these challenging years of study: without your unconditional support and shoulder to lean on, this wouldn't have been possible.

Finally, to all my family and friends, especially to my beloved mother and wonderful sister for their endless love, and encouragement, and support.

Laritza Cabrera Alba.

Abstract

Key-value stores are the backbone of many interactive cloud services. Response times on these services, expected to be very efficient almost all the time, are largely tied to the latency response of their respective key-value database engines. Keeping such a latency low in key-value stores remains a challenging task, in particular for those queries with worst response times, typically located in the tail of the latency distribution. In this thesis, we consider exploiting records size prediction as complementary information to support scheduling algorithms during replica selection, so that they can make more informed decisions. We explored the different conditions under which this becomes feasible, and more importantly beneficial, in the context of multiget queries involving heterogeneous data. To this end, we propose a shallow neural network to predict whenever a queried record-column pair will be large, even before seeing this particular record (i.e. primary key) for the first time. Our network exhibits a limited computational overhead for inference, and unlike Bloom filters, can be updated "on the go" without the need for restart nor retrain from scratch. Experiments conducted on several synthetic datasets confirm that the proposed network attains a beneficial state, namely more True Positives identified than mistakes made, after seeing approximately 25% of all the records. Our results suggest that it is indeed possible to accurately predict if a key-column opset will be large, but only whenever large records are not evenly distributed among the different columns, or equivalently, whenever they are "rather grouped" into a set of a few columns.

Contents

Chapter 1

Introduction

Key-value stores, a type of NoSQL database engine, allows the storage and retrieval of massive amounts of data with seamless on-demand scalability. They constitute the backbone of many interactive services of different nature and needs, covering from lightweight text-based blogs content delivery, to social networks and e-commerce services, up to interactive multi-media platforms such as Netflix. Response time service-level objectives (SLO) in these systems are quite strict, meaning that the response time is expected to be very efficient almost all the time. Slow response time in web platforms does alienate active users, resulting in significant losses of revenues and growth, even if the service is only affected for a brief period of time [1, 2].

1.1 The heavy tail latency problem

Response time in interactive services is mostly driven by their associated key- values database engine latency. Maintaining a low latency in these engines is, therefore, an essential task required to deliver scalability while retaining a high quality of service (QoS) in compliance with targeted SLOs. Although mean or median latency values are indeed very relevant indicators of the overall performance in key-value store systems, they only provide a limited picture that is insufficient to assess whether or not the service is performing as it should.

Other fundamentals of the latency's distribution can be much more relevant. This is the reason why SLOs are defined upon the 95th, 99th and 99.9th percentiles of the latency's distribution, meaning that to evaluate the performance of certain service, they only care about the smaller group of queries that take longer than usual to be serviced. Take for example a service provider that claims to guarantee a 99% response time of 10ms after receiving a request. In line with the previous, this provider must spend the necessary efforts to ensure that less than 1% of the

serviced requests take longer than the aforementioned threshold. In other words, assuring a QoS under this SLO commitment requires that the provider executes a strategy, among several that will be discussed in further sections, in its database engine to keep the 1% worst service times bellow the desired committed threshold. This problem is typically known as reduction of long-tail latency, or taming heavy tail latencies, and is at the core of this thesis.

1.1.1 Why does latency occur in the first place ?

Long-tail latency can occur due to a large variety of reasons and involves a complex combination of interacting factors, including contention for shared resources and sub-optimal scheduling.

Contention for shared resources. High CPU loads and intensive memory accesses in multi-tenant virtualized environments, as well as traffic microbursts in shared networks are known causes leading to performance fluctuations that affect latency. Even in well-provisioned systems with nearly ideal conditions, maintenance and periodic tasks such as logs rotation are known to trigger latency spikes in individual hosts, that can eventually spill over other nodes within the same cluster [3]. As such, this type of latency is very difficult to predict, and many efforts to control it are focused on keeping an overall cluster-wide view on CPU loads and response times of individual nodes, and then exploiting this information during the replica selection to avoid overloading nodes with above-average CPU loads.

Heterogeneous records size. One of the most important aspects that contribute to latency is the differents size that different records have in a given database. If all the records were the same size, it would be more easier to plan the execution of the differents queries for optimal job scheduling. However, it has been shown that this rarely the case [4] and that neglecting the different sizes of the records can lead to significant performance deterioration. Overall, it is quite unlikely that all records in a key-value store are similar in size, and a more realistic scenario is to consider that their size is heterogeneous. Consequently, job planning should benefit from including this type of information.

Head-of-line blocking. Head-of-line blocking is another very common cause of latency in KVS. It mainly occurs as a result of sub-optimal schedules when planning the execution of queries. A simple example of head-of-line blocking is illustrated in Fig. (1.1), where the job is considered completed once all the records have been fetched, so they can all be returned together at once. The latency in the

head-of-line blocking mainly occurs due to the heterogeneous size of the different records.

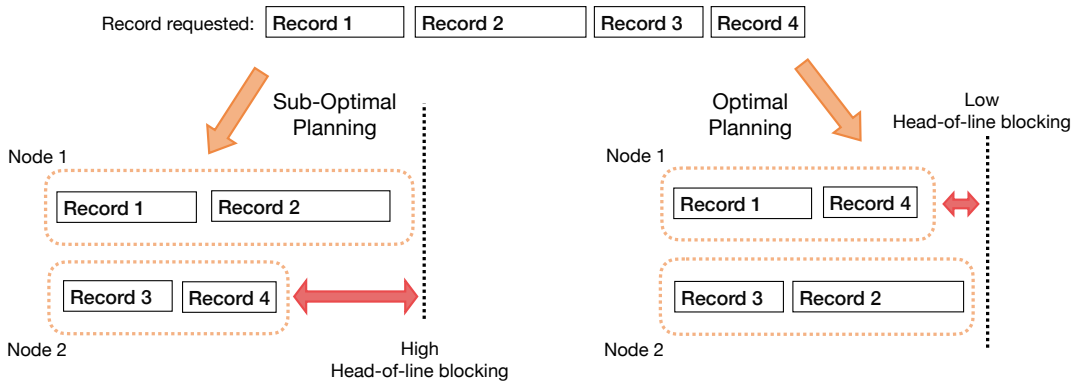

Figure 1.1: Head-of-line blocking, depicted as red arrows, involving 4 jobs and 2 servers. (left) Poorly planned jobs execution leads to higher head-of-line blocking. (right) Better execution planning leads to minimal head-of-line blocking.

As can be seen, poor execution planning of the jobs will increase the waiting time until all the records have been fetched. More specifically, even though one of the two nodes has already completed the retrieval of all its assigned records, the other node is blocking the final release of all records until it completes its jobs.

It is worth to notice that if all the records had the same "size", then there would be no head-of-line blocking issue since any execution order or planning would be equally efficient.

1.1.2 State-of-the-art on tail latency reduction

Most of the current works in tail latency reduction can be grouped into two main categories according to the use of size information and heterogeneous nature of the key-values pair. Information related to the query, including the number of records in multiget request [5], but under the assumption that all key-values are of similar or fixed-size (i.e. homogeneous). Authors in [6] keep indicators suggestive of high latency, including the recent history of reading latencies, to select the best replica. In the work of [7], the authors exploit information about the size of multiget requests, processing time, and fan-out, to identify opportunities for tail latency reduction. Yet, the size of individual key-value pairs is neglected. Head-of-blocking is avoided in Heron [2] by considering variable execution time of key-value pairs and using a Bloom filter to keep track of those pairs with the highest read latency.

More recently, authors of TailX [4] proposed exploiting the size of individual records for optimal scheduling of multiget queries. A Bloom filter is employed as well to keep a list of the largest records. In both works [2, 4], an efficient gossip protocol is used to share either the records that took the most to be fetched [2], or the largest records in size [4].

However, Bloom filters are known to require large amounts of memory and cannot cope with frequent updates, meaning that they need to be restarted (or cleared) and retrain from scratch again. Since it is difficult to know what the number of records in a KVS would be in the future, the memory of these filters needs to be set to arbitrarily large numbers in advance during capacity planning. The aforementioned issues limit severely the applicability of Bloom filters in circumstances where the hardware setting is not ideal and the databases are large, as well as in key-value stores handling frequent write (i.e. update, delete, etc.) operations.

Furthermore, most of the previous works consider a coarse and rather limited view of a record size as uniquely small or big, even though the size of a record is the result of the composition of its associated key-value pairs. As a consequence, no discrimination is made among records that are big, albeit a record of 100kb can be big but certainly not as big as a record of 1mb. Another important aspect often neglected is that many key-value stores, such as Cassandra currently allows requesting a sub-set of columns - or cells/key-value pairs - from one or multiple records, but not necessarily all columns. Making a distinction not only between records (i.e. primary keys) but also between the different key-value pairs associated with one record would require even bigger filters.

Recent works have considered different strategies to obtain direct or indirect information about individual records size, and have shown remarkable success in exploiting this information toward reducing tail latency. However, less attention has been paid to the data-driven prediction of records size. There is currently a need to incorporate fast predictive systems of records size that could underpin scheduling algorithms during replica selection, in particular for multiget requests in heterogeneous workloads involving a variable number of columns.

1.1.3 Fetching multiple rows but only some cols in Cassandra ≥ 3.4

Most of the current works consider that fetching a record necessarily implies retrieving all the key-value pairs, regardless of the columns requested in the query. However, Cassandra-10567 [8] and newer versions explicitly address skipping the reading of columns not requested in queries. It is unclear if this feature always leads to speedups and performance improvements. Yet, we know that this improvement is indeed obtained (17% of speed-up in [8]) in cases where the columns requested

are smaller than the rest of the columns within the same record.

From the previous, we conclude that requesting a list of records with columns $(c1, c2, c3)$ can be slower than requesting the very same list of records but only with columns $(c1, c2)$. This highlights the importance of not only differentiating between the size of the different records, but also between the size of the cells within individual records.

Under this view, we cannot categorically say if a record is big or small unless we also consider which columns or key-value pairs our query is including; consequently:

- certain record A can be small whenever it includes the columns $c1, c2$
- the same record A can be considered big whenever it includes the column $c3$

Popular APIs such as DataStax [9] and Hector [10] already provide the necessary mechanisms to perform multiget requests and while specifying which columns (or key-values) will be fetched.

1.2 Records size as a consequence of the size of its key-value pairs

1.2.1 Records size vs size of its key-value pairs

It seems intuitive to consider that the overall size of an individual record is just the composition of the size of the different key-value pairs that this record contains, plus some negligible meta information such as the record TTL. This is true in the specific case where all the key-value pairs of a record are relevant for a query, but more importantly, it also holds for the more general case when only a few key-value pairs are involved in a query.

Interestingly, approaches such as Bloom filters are applicable in this context as well. However, since we are no longer differentiating between single records (i.e. primkey) but also between the different columns within a single record (i.e. primkey plus colName), the memory requirements of these Bloom filters could be much higher, depending on the number of different columns and the frequency on which a larger key-value pair within a single record is requested. It could be that some records have zero or one big cell within, and other records could have several big cells.

1.2.2 Challenges in preventing heavy tail latency in this context

Predicting records size, either in a coarse or fine-scale, remains a difficult task since this task requires identifying recurrent patterns present in the data, provided that these patterns exist - which is not necessarily always the case. From the practical side, its application context set many constraints on the performance of the predictive model, especially with regards to the computational overhead during both inference and updates.

1.2.3 Research questions

In this thesis, we consider the more general and realistic case of multiget queries with heterogeneous data sizes, in order to address the following research questions:

- could it be possible to predict individual records size accurately?
- under which conditions this might be possible and beneficial?

1.3 Rationale

1.3.1 Global organization of key-value stores

Due to the nature of key-value stores, the data is distributed and redundantly stored in multiple nodes within the cluster. Differently from other data models, the Cassandra data model illustrated in Fig.(1.2) keeps a copy of a group of records, defined by the partition key, in several replicas. It includes key namespaces as a kind of differentiation of databases, as well as column families which are effectively a kind of table. Each record is composed of a list of key-value pairs known as cells. A table or column family can include multiple columns, and a record can contain key-value pairs involving one or several columns but not necessarily all of them.

1.3.2 Uneven per-column distributions of big cells

Let's consider that, regardless of their nature (i.e. numbers, text, images, videos, etc.), all the cells in a key-value store are organized in a single column. Suppose that we would like to separate them into two different columns, without necessarily following any particular order. The Fig. (1.2) illustrates how a list of cells, identified as "red" (denoting large cells) or "green" (denoting small cells) types, can be organized into two columns in three different ways.

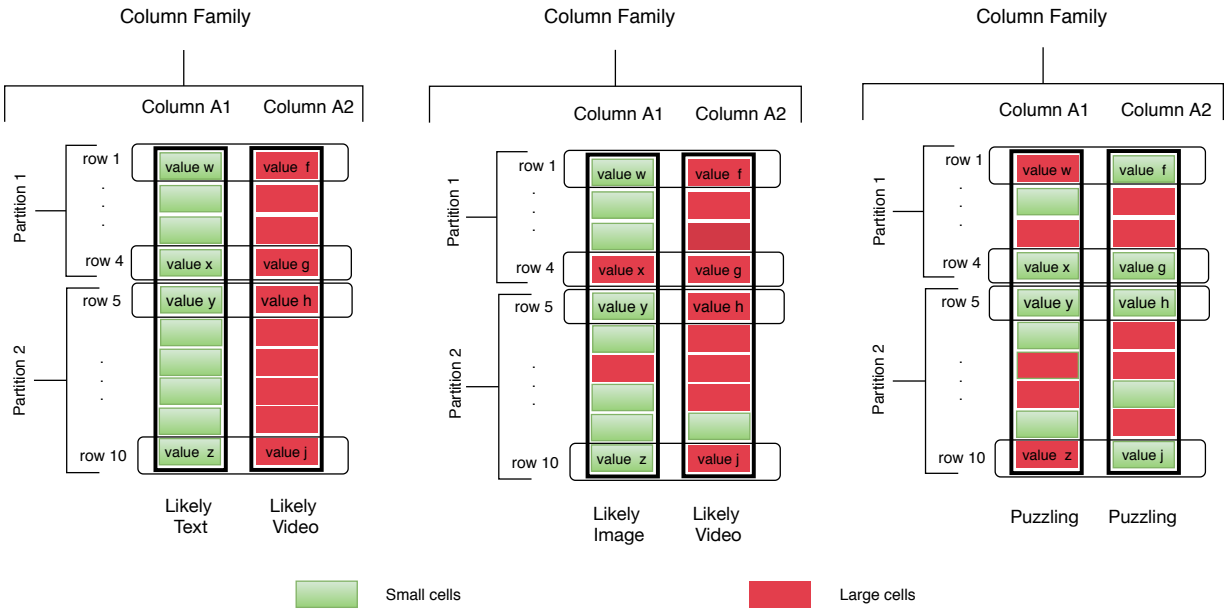

Figure 1.2: Different organization of records into two columns. (left) Clear organization suggesting that each column store records with information of different nature, such as Text and Videos in columns A1 and A2, respectively. (middle) Less clear organization, such as what one could expect when storing Images and Videos in columns A1 and A2, respectively. (right) No organization at all makes it difficult to judge what kind of information could be stored in the two columns.

Exploiting grouping information to predict big records. As can be seen, information with regards to the amount as well as the frequency of "red" cells present within a column could provide a hint on the type of information that could be found within a certain column. This is just an example, and albeit it would be difficult to guess exactly what's stored in the different columns, this example highlights how the column itself can be exploited as an organizational unit. This becomes evident in cases where "big" cells are rather organized or grouped.

However, relying on column information is not enough to predict whether a cell will be big or not, since it's very unlikely that any column family will always be organized like that. Instead, a combination of primary keys and column names could be useful to identify grouping patterns across the different columns and column families.

1.3.3 Neural network for records size classification using col names

In this work, we propose a neural network model, illustrated in Fig. (1.3) to classify a cell into small or big from by predicting the size of each of its $(primkey, column)$ tuples present multiget queries involving heterogeneous data. Such a neural network tries to exploit the possible relationship between individual cells and the column-wise organization of its contained values.

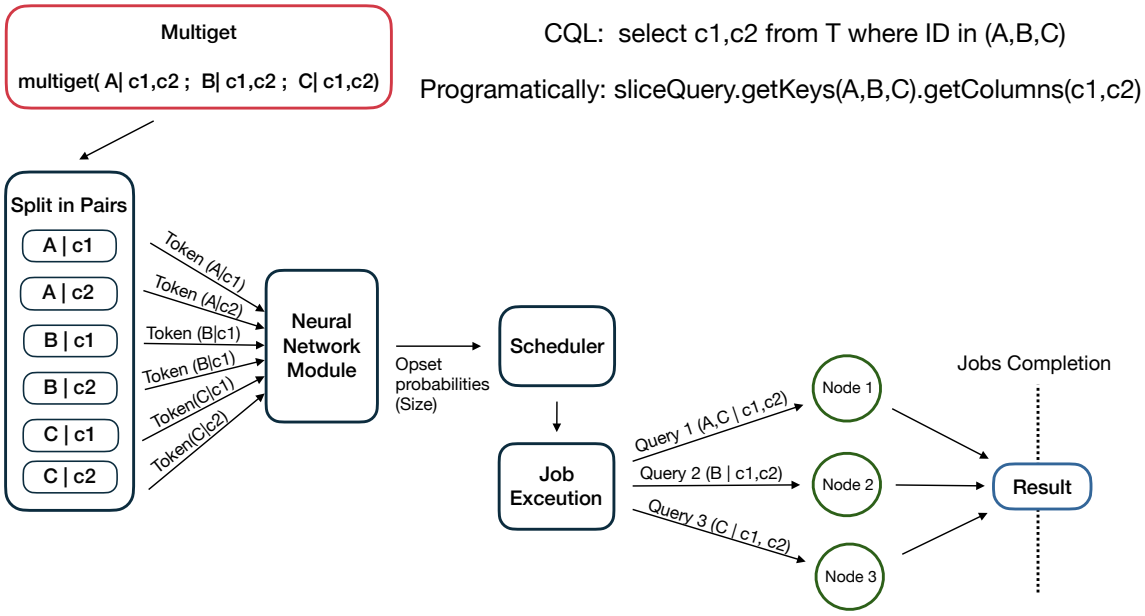

Figure 1.3: Scheduler takes more informed desicion.

Our network is trained online as the different records are read from or written into the KVS. It exhibits a (limited) computational overhead of $O(c/Q)$ and $O(c)$ in predictive and update operations, respectively, for a multi-query of c columns involving Q parallel threads. This makes possible the integration and exploitation in modern key-value store engines. The main contributions of this thesis are the following:

- we propose a neural network able accurately predict a cells size class under relatively mild though realistic assumptions:

  (i) the size of cells in the KVS follows a log-uniform distribution

  (ii) big cells are relatively well grouped column-wise

- we analyze the conditions under which our neural network might be effective, and more importantly beneficial
- we evaluated the accuracy of our model in numerous experiments involving synthetic datasets as in [11]

Neural networks models have been successful in tackling many challenging tasks that require identifying patterns in large amounts of data. However, these models require extensive training routines, significant amounts of data and their ability to learn patterns is related to the network size (or capacity). Bigger networks might have a better prediction performance but will also involve a larger number of multiplications during prediction, which increases the computational costs of inference and therefore makes then unsuitable for our task.

We studied how the different degrees of column-wise records grouping impacts the ability of a shallow neural network to identify large records. We generated 30 different databases, each one containing 1 million key-value pairs with different per-column organizations or grouping degrees. These databases were used to compare the predictive performance, in independent experiments, of the proposed model vs Bloom filters and a baseline model, as well as to measure the latency of our neural network during records size estimation.

The remainder of this thesis is organized as follows. In the next chapter, we provide a brief description of concepts related to our proposition, while the proposition itself is detailed in Chapter 3. Our model is evaluated in Chapter 4, including an assessment and discussion of its most important aspects. Chapter 5 concludes our findings, identify shortcomings, and provide a perspective on its application and potential improvements.

Chapter 2

Preliminaries

As discussed in the previous chapter, our approach relies on the assumption that cell's sizes are log-normally distributed but also that the big cells are rather grouped together column-wise. The size of all cells within all records in a key-value store seems to follow a log-normal distribution [12] when taken all together. However, as shown in Fig.(1.2), big cells can be grouped together in many different ways.

Generating a list of cells whose size follows a log-normal distribution is quite straightforward. Yet, organizing these cells in a way that they exhibit certain degrees or grouping is far from straightforward. For instance, could simply generate a 26 million of cells and assign them a size value obtained by sampling from a log-normal distribution (described by certain parameters). Then, we can randomly organize these cells in 26 columns and 1 million of records, which is also quite simple.

What is not trivial is how to measure the organizational degree or level that big cells are exhibiting in a given database. More importantly, beyond measuring the degree of grouping that big cells have in a given database, we are interested in generating a database with specific and varying degrees of grouping for all big cells, so we can assess the performance of our proposed solution (and others as well) under different degrees of groupings.

In this section, we discuss how it could be possible to generate a database where all the cells and records taken together to follow a log-normal distribution, but more importantly, what procedure to follow in order to guarantee different degrees or per-column grouping. A specific metric (to be defined next) is then used to measure the resulting degree, in the following intervals $\{0.0, 0.25, 0.50, 0.75, 1.0\}$, of column-wise organization of big cells in our randomly generated databases. The details of data generation will be provided in Chapter (4)

2.1 Dispersion values between groups in log-normal distributions

A log-normal distribution is essentially a normal distribution whose values are transformed into a logarithmic scale, as follows:

$$y = \mathrm{e}^{\mathrm{x}} \tag{2.1}$$

and therefore the parameters (i.e. mean and var) of the log-normal distribution Y is related to the mean of its corresponding normal distribution X as follows:

$$\mu_Y = \mathrm{e}^{\left(\mu_\mathrm{X}+1/2\sigma^2_\mathrm{X}\right)} \tag{2.2}$$

$$\sigma^2_\mathrm{Y} = \mathrm{e}^{\left(2\mu_\mathrm{X}+\sigma^2_\mathrm{X}\right)}(\mathrm{e}^{\mu^2_\mathrm{X}} - 1) \tag{2.3}$$

Suppose that we divide a list (i.e. one column) of samples related the distribution Y into C groups (or C columns), each group $Y_1, \cdots Y_C$ also being a log-normal distribution, with their corresponding normal counterparts denoted as $X_1, \cdots, X_C$.

As consequence of (2.2 and 2.3), the statistical parameters of these normal counterparts and the log-normal distribution Y are related as follows:

$$\mu_Y = \mathrm{e}^{(\sum \mu_{\mathrm{X_i}}+1/2\sigma^2_{\mathrm{X_i}})} \tag{2.4}$$

$$\sigma^2_Y = \mathrm{e}^{(2\sum \mu_{\mathrm{X_i}}+\sigma^2_{\mathrm{X_i}})} - \mathrm{e}^{(\sum 2\mu_{\mathrm{X_i}}+\sigma^2_{\mathrm{X_i}})} \tag{2.5}$$

The implication of the previous is the following: we can generate a list of C populations (with parameters μ_i, σ_i), and we know that when they are combined or put together, they will conform a log-normal distribution with parameter μ_Y, σ_Y when transformed as Eq.(2.1).

A second takeaway is that: for the same μ_Y, σ_Y, we can have populations where $\mu_1 = \mu_2 = \cdots = \mu_C$ or just different $\mu_1 \neq \mu_2 \neq \cdots \neq \mu_C$; this will be discussed in detail next. Yet, what is relevant is that this opens a path organize the very same population (μ_Y, σ_Y) of log-normal distributed values into C groups, in different ways with different grouping degrees, and measure how distinct or similar these C groups are among them.

2.1.1 Per-column organization of values and dispersion.

It is possible to divide Y such that all the resultant distributions $Y_1, \cdots Y_C$ are exactly the same, and hence $\mu_{X1} = \mu_{X2} = \cdots = \mu_{XC}$. In other cases, the same Y

distribution can be divided in a way that they differ. The variance of their mean can indicate how dispersed or grouped the values are among the different groups:

$$\text{Var}(\mu_{X1}, \cdots, \mu_{XC}) \tag{2.6}$$

It becomes clear that whenever the values in those groups are roughly the same, then $\text{Var}(.) = 0$; on the contrary, any value of $\text{Var}(.) > 0$ will indicate that the values are rather grouped. In our experiments, we calculate Eq.(2.6) per-database to confirm the different degrees of organization.

2.1.2 Uneven distribution of values/cell size among different columns

It is worth to notice that the variance in Eq. (2.6) is the variance of the means of differents distributions, and therefore the variance is equal to zero whenever the variance is roughly the same as shown in figure Fig. (1.2 right). In the opposite case, whenever the variance is higher than zero, then the means will be different and consequently, the columns will be different as shown in Fig. (1.2 left).

From the previous, it can be seen that the dispersion is zero whenever big cells are unevenly organized and it is higher than zero whenever big cells are rather organized into a few columns. The higher the variance between the different columns, the higher the cells are organized within a sub-set of those columns.

2.1.3 Threshold to define a large record

There is a clear relationship between the threshold set to determine whenever a record size is considered large, and the number of such records available in the database. To a lesser extent, this threshold will also have an impact on the degree of dispersion of large cells between the different columns with regards to their size. For example, if one column contains small and moderate size images, then a high threshold will set all the records of this column as small records. Meanwhile, a low threshold will set all these records as large records. It can be seen that at a specific threshold value, half of these records will be set as small and the other half as large records.

Chapter 3

Methodology

Predicting the size of records as a whole, or even each of its key- value pair or cells requires learning or discovering patterns that allow to discriminate between them, provided that such patterns exist. In its simplest form, our problem consists in learning the following functional:

$$f(x) \quad = \quad y \tag{3.1}$$

that, from a given description (i.e. tokens) of an element x, predicts its size y. The size y can be a dichotomic value representing two classes, like 0 or 1; can be a real number indicating the probability of x being a certain class; or it can even be a list of values. The learning problem therefore consist in finding the functional $f(.)$ whose prediction is the same as the real size y:

$$(f(x) - y) = 0 \tag{3.2}$$

where the difference, which actually quantifies the prediction error, will be exactly zero whenever the prediction $f(x)$ matches the real size y. In the case that the is not perfect, we would like that the prediction error is as small as possible.

3.1 Deep neural networks

Neural networks (NN) are known to be universal function approximators [13], thus we can represent f by a neural network, and train the neural network to minimize (3.3). The proposed neural network is detailed in Fig. (3.1).

As can be seen, for each $(primkey, col)$, the neural network receives a list of 10 tokens and provides a real number between indicating the probability of the cell corresponding to the $(primkey, col)$ key-value pair to be bigger than a previously specified threshold.

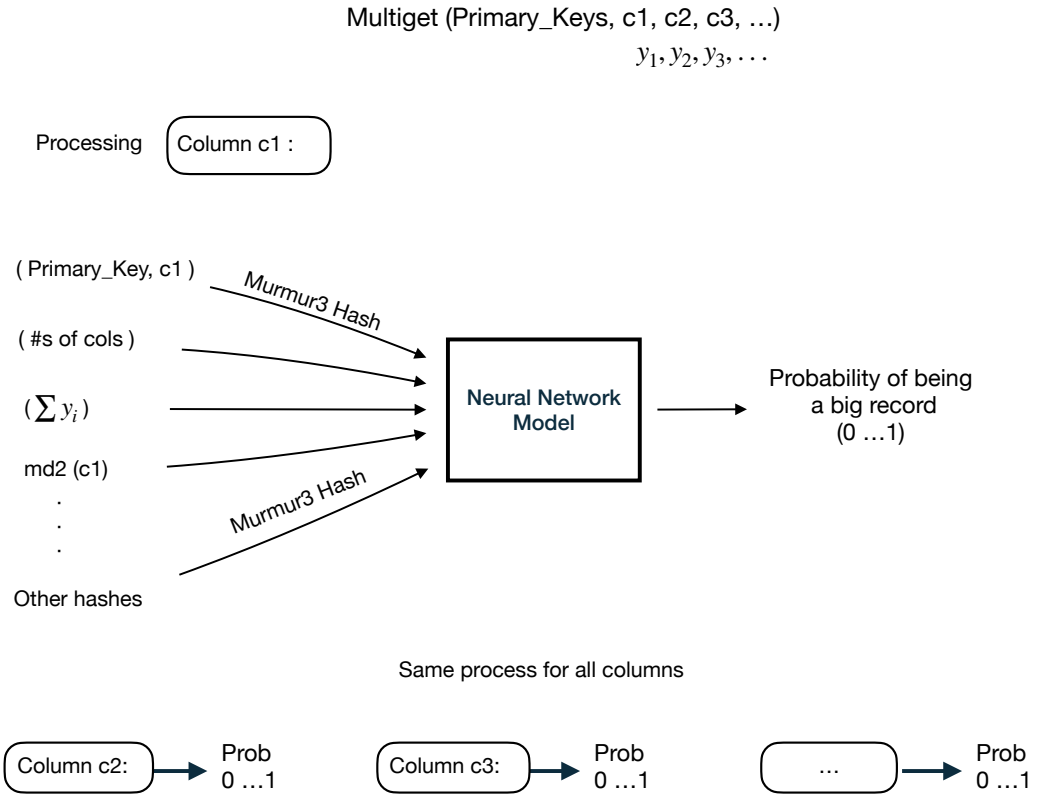

Figure 3.1: Design of the proposed neural network to predict records class.

3.1.1 Input layer and tokens generation

The input to our neural network consists of a list of tokens, with values $0, \cdots, 1$. These tokens are generated using certain information that is hashed using the Murmu3 algorithm implemented in Java [1] that provides a hash with values between $-1, \cdots, 1$ that is multiplied by 0.5 and added $+1$ so they finally are in between 0 and 1.

From token to element in a set. These tokens can be multiplied by the number of elements N in the set, and ceiled (i.e. round by excess to the next larger integer), to obtain a value between 1 to N.

[1] Code available at `https://github.com/laritzalba/neural-network-predictor`. Credits for murmur3 implementation go to `http://murmurhash.googlepages.com/` and its authors.

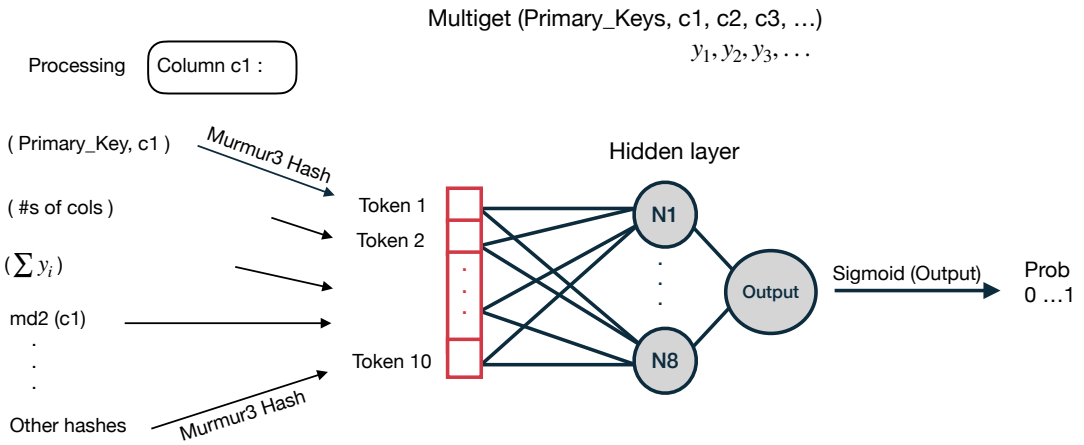

Figure 3.2: Neural Network Model

Information as tokens. As input, one typically uses the record's primary key in order to generate a token identifying this record in the set of 1 to N. However, this does only work assuming that the requested queries involve all the columns. Since we are identifying which columns are small or big for a single primary key, instead we use a combination of:

- $(primkey, colName)$ preceded by the keynamespace and table (i.e. column family), example: 'laritzadb.tab.221955.A'
- number or position of the column from 1 to 26 (i.e. from A to Z) identified as y in Fig. (3.1)
- summation of all the positions of the columns involved in the request
- independently generated hashes of the column name (i.e. from A to Z), including: md2(.), md5(.), sha(.), sha-224(.), sha-256(.),sha-384(.),sha-512(.)

and the Murmur3 hash of each of the previous information is what finally gives the input to the neural network, as a list of 10 tokens.

As discussed in YCSB [11], we made sure that the tokens generated are not located in the same real values range, namely that they are spread all over the values $0, \cdots, 1$. Likewise YCSB, we generated unique ids (i.e. primkeys) as integer values between $1, \cdots, 20 \times N$ instead of $1, \cdots, N$ using Zipfian. During the experiments (Sec 4), we confirmed that this is the case by showing the grouping of tokens associated with big records (see Fig. 4.2), by displaying these tokens generated for $10,000$ records.

Input layer. For each primary key and column pair, we generate a total of 10 tokes as previously described. Given that their values are between 0 and 1, these tokes are used as is to define the input layer of our neural network as can be observed in Fig. (3.2).

3.1.2 Learning the Neural Network

The learning problem can be specified as finding an $f(.)$ where:

$$\min_{w} \mathcal{L}(x) = \left(f(x; w) - y\right)^2 \tag{3.3}$$

where the function $f(.)$ is represented by a Neural Network with unknown weights w that must be learned using backpropagation [14] . From optimization, we know that the loss $\mathcal{L}(.)$ can be minimized using sequential updates, and that the solution w_* lays in the direction of the gradient. Thus, for the updates of w can be defined across time as:

$$w_t \quad = \quad w_{t-1} - \eta_t \nabla_w \mathcal{L}(x) \tag{3.4}$$

during execution of the t-th query at time step t. The parameters η_t are a learning rate, which regulate how much we allow the neural network to be updated every time, and essentially re-scale the magnitude of the gradient at the point x while keeping its direction; typically $\eta_t = cte \quad \forall t$ where the cte is initialized at between 0.5 to 0.005.

3.1.3 Algorithm

Our algorithm starts by accessing a queue of pending queries to be processed. For a given number of threads Q and learning rate η, the algorithm to train the network while it is being used to predict is the following:

As can be seen, for each query, the list of keys and columns involved are separated in pairs, and the neural network is then used to predict their sizes (steps 4-6). This information is then exploited by the scheduler to plan the execution of the queries (step 8). The queries are then executed according to the planning decided (step 9), and once the execution is completed, the real size of each key-value pair is used to correct the network (step 10). The records fetched are finally returned to the client.

Algorithm 1: Neural Network for Size Prediction

Input: Learning rates η_t
Initialize: w randomly
waitfor next query to arrive **do**
    foreach $(primey, col)$ pair **do** in paralell with Q threads
        $x_i = \text{GenTokens}(primkey, col)$ for the i-th pair
        $s_i = f(x_i; w)$ for the i-th pair
    endfor
    Scheduler plans execution with all $\{s_1, s_2, \cdots\}$
    Execute queries and reveal real per-record cells sizes
    foreach Incorrectly Predicted pair **do**
        Update w as in Eq.(3.4) with prob. 1 (or $1/Q$)
    endfor
    Return fetched records and process next query

3.2 Neural network design and trade-offs

In our solution, the neural network is composed of 1 hidden layer containing 8 units, which is a very light neural network. The output of the network is also corrected by a sigmoid function to ensure that all the predicted values are in the desired range of 0 to 1 so that it indicates probabilities. This network design provides several benefits as well as drawbacks.

3.2.1 Network capacity

The most tangible benefit is that each prediction requires 88 multiplication operations so it can be very fast during inference. Notice that having additional layers or a larger number of nodes (or both) is possible and configurable in our design. Yet, this would make the inference process slower by increasing the number of multiplication operations required for each prediction, which increases the computational overhead and could make the network unsuitable for our problem where fast inference is a must.

On the negative side, the limited number of layers and nodes affects the accuracy of the network. In other words, networks with a larger number of hidden layers and nodes can potentially be much more accurate. However, as discussed before, a bigger network will incur higher computational overheads for each prediction. Furthermore, a bigger network could also need more time to learn different patterns.

3.2.2 Memory required

Another important benefit is that the network occupies a negligible amount of memory in the server. In our case, this amount is $88 * 8 = 704$ bytes, which is less than 1kb of memory. Even very large neural networks are unlikely to consume beyond a few dozen MB of memory, which is an advantage compared to other models.

3.3 Online training

The training process should happen as the data related to the different queries requested to the key-value stores flows in. That implies correcting the neural network while the records are being processed. As the primary keys and columns of the records arrive in as a query, the neural network is used to predict the size of each of the cells associated with all the records. Regardless of the correctness of the prediction, the record will be scheduled for fetching, and the true size of the record and its key-values will be known. At this point, if the prediction was incorrect, then the network should be updated so that $f(.)$ becomes a better predictor for future records involved in future queries. This process is repeated continuously to keep improving the network as more data arrives.

3.3.1 Convergence across time

The parameter η regulates how much the neural network is corrected at every update. To guarantee that the neural network prediction becomes stable and does not largely change, we decrease the value of η after every certain amount of processed read (i.e. SELECT) queries, down to a certain minimum value. We use several learning rates η_t, defined as:

$$\eta_t \quad = \quad \max\left(\eta \cdot 1/t^{0.5}, ctemin\right) \tag{3.5}$$

On the contrary, η_t is increased every time a write (i.e. UPDATE) query is executed. The aim of the previous is to ensure that the values of the neural network converge after a while. As the number of iterations t increases, the value of $\eta_t \to ctemin$ which essentially stops the learning process for a low $ctemin \to 0^+$.

Chapter 4

Experimental Results

The assessment of the proposed solution requires the execution of several multiget queries in sequential order. In this work, we consider an experimental setting where a database has already been populated in a key-value store and that does not change. This setting mainly aims at evaluating the predicting performance when fetching records. Although limited, this setting is standard practice in database performance benchmarks. Nonetheless, our solution can be applied to other settings without additional requirements. Differently from Bloom filters, adding and removing records from the database should not have a significant impact on our solution.

Notice that as long as the distribution of big records -whichever it might be- in the different columns do not change substantially, our solution will still be able to predict them by exploiting the very same information. On the other hand, we expect that removing and adding columns relatively often in such a database would impact negatively our solution. We compare the proposed solution against a random decision, which is a model that randomly guesses if a record is small or big, and against a model that always tells that a record is small, which is essentially ignoring the value of predicting the record's size.

Update and Delete operations. UPDATE queries or writes operations are treated likewise any read operation, but without actually fetching the records. This means that for an update operation, the neural network predicts the size of the record, and if the prediction is not correct then it is corrected with the true size, which is available by definition in the update query. The cost of an update is $O(c)$ for c key-value pairs, and cannot be performed in parallel. If necessary, the neural network correction can be done (step 7 of the algorithm) with probability $1/Q$ which will lead to $O(c/Q)$.

For our purpose, a DELETE query can be treated as an UPDATE query where the size of the record and all its key-value pairs is set to zero. Arguably, once a record has been deleted, it should have no effect in the performance since it should

not be requested in any subsequent query [1].

4.1 Data preparation

In our experiments, we generated a total of 30 synthetic databases, corresponding to 5 different types of dispersion/grouping and 6 experiments per type, so that each experiment can be independently repeated "in a different database" each time. We followed a generation the same strategy (or as close as possible) detailed in [11]. Each database contained 1 million of records that were stored in a single keyspace within the same column family composed of 26 columns. For each database, we simulated the execution of a list of queries, that were generated as follows:

1. pick randomly one or more records using a Zipfian distribution
2. pick randomly a subset of columns from this record using a Zipfian distribution
3. SELECT {subset of colums} FROM tab WHERE id in ({list of keys})

and this process was repeated until the number of key-values involved in the queries all together reached 1 million. In total, around $110,000$ of queries were generated, each query involving one record and approximately 7 ± 5 columns requested. Multiget can then be defined by grouping 2 or more consecutive queries into one. Also, for each of the 30 databases, we share [15] the list of records and columns per query, as well as the real size of each key-value associated with the queried record.

In the context of query execution, it is relevant to know which columns from which records will be the biggest, hence we treat each query as a multiget composed of several pairs in the form of (primkeys,colName). Providing more granular information about the size of individual key-value pairs allows to decide if a record is big or not.

Three distributions for three data aspects. These three aspects of the data are modeled thru three different distributions.

- Popularity of a record (Zipfian): some records (primkeys) are more often requested than others
- Popularity of a column (Zipfian): for a given record (primkey), some columns are more often requested in a query involving that specific record

[1] It is not clear to use if a KVS will schedule the execution of a query involving a record that does not exist in its tables/indexes.

- Size of a cell (log-normal): the size of a cell, which is a key-value pair contained within a record/row
- Size of a record (log-normal): a record is a composition of cells and therefore it follows a log-normal distribution likewise cells

Next, we will detail how these distributions are used to generate data, and which parameters are used for the generation in our experiments.

4.1.1 Single-record popularity as a Zipfian distribution.

In any key-value store, some specific records (i.e. primary keys) are more popular than others. This is in line with the fact that some websites or smartphones applications are indeed more popular than others, and thus the records associated with them must be more frequently fetched from the database. To model this popularity, we used a Zipfian distribution [16] with a rho parameter value of $\rho = 0.99$ in a records space of 20million records.

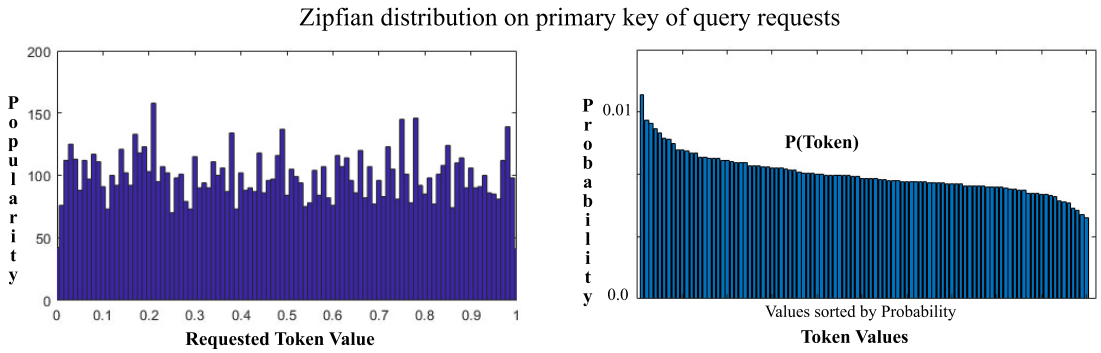

Figure 4.1: Popularity of the queries is independent of the record type. (a) Primary keys frequency. (b) Distribution of primary keys frequency.

The resultant records generated with this distribution parameters are shown in Fig. (4.1) for a total of $10,000$ records, where the records are represented by their corresponding tokens. As can be seen, different records have different popularity and some are more often requested than others.

4.1.2 Columns access frequency as Zipfian distribution.

For each data generation, we started by generating a total of 1 million of key-pair values. These were organized into approximately 100,000 records, each one containing a variable number of key-pair values. This allowed to model queries where the same primary key was requested using a different set of columns.

Likewise, in the case of the record's popularity, it seemed more realistic to consider that certain columns were more popular than others. For example, we could see the price and picture of a product in an e-commerce platform, but only a few times we would click to see more details related to it. Similar to an image on Instagram, where one sees the image, and only sometimes go for more details about one particular image. Therefore, we considered that the popularity of each column included in multiget requests followed a Zipfian distribution with a parameter $\rho = 0.99/2$.

To select the columns, we first permuted the column names $\{A, \cdots, Z\}$ to get a randomized (uniform) order, like $D, A, C, \cdots, B$. Next, for each multiget, we sampled 26 numbers at random with replacement following a Zipfian distribution, and discarded those columns repeated in the list. These numbers indicated the name of the columns to be included in that particular multiget. The process was repeated until all the million records were organized into different multigets.

As summary, this process provided an average number of requested columns per multiget of about 7 ± 5, where the column popularity followed a Zipfian distribution.

4.1.3 Log-normal distribution of the size of key-value pairs.

We modeled the overall size of the individual cells (i.e. key-value pairs) as a log-normal distribution, that was earlier discussed in Sec.2, with distribution parameters $\mu = -0.3$ and $\sigma^2 = \frac{1}{6}$. These values were scaled and truncated so that the final key-values sizes were in between 1byte and 1mb.

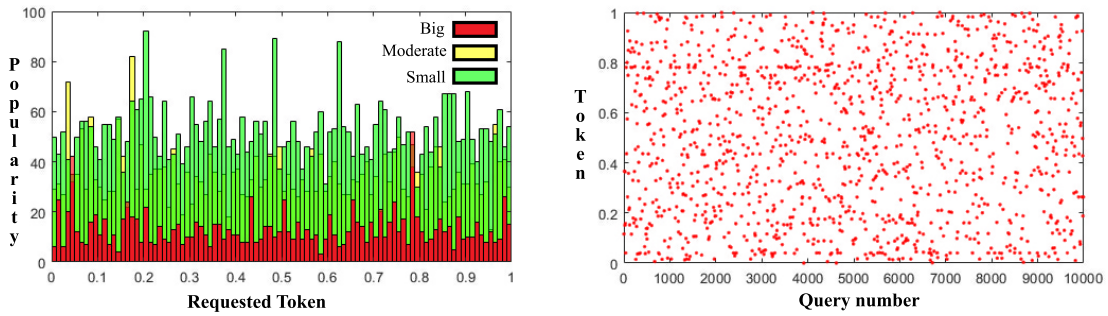

Figure 4.2: Popularity of the individual cells according to their different size. (a) Primary key according to the record size. (b) Large records sequential query order vs token.

The Fig. (4.2) shows the distribution of big, moderate size and small records. As can be seen, Our generation guarantees a proper spread of big records across the different token values, as well as their per-token distribution compared to small records.

Column-wise grouping of records. During the generation, big records were organized considering five different degrees of per-column dispersion with values of $\{0, 0.25, 0.50, 0.75, 1.00\}$. The process to generate the records according to different columns has been described earlier in Chapter (2).

Threshold setting. As detailed above, we consider as big records those with cells that have at least one cell with a size bigger than the previously defined threshold. For the rest, we considered as small all those with all cells less or equal to 2kb, and of moderate size those records whose cells were all between 2kb and the threshold.

The cut-off value to determine if a key-value pair was big or small was set to 38kb, since this threshold would guarantee that in no dataset, the number of big key-value pairs was never higher than 20%, with a typical percentages between 1% and 7% for most of the 30 databases finally generated.

4.2 Experimental setup

4.3 Simulations

Each experiment consisted of sequentially querying a list of multiget requests, one after the other, and predict each column's size present within the multiget by using our model for inference. This prediction would then be passed to the scheduler for the planning of the different keys and columns in the different replicas. After the records have been retrieved, then the real size is used to update our model.

To evaluate the prediction performance, we didn't have to run any scheduler or planning, and just assessed how well the big key-value pairs were identified by the different models under assessment.

4.3.1 Evaluation Metrics

During the queries execution, we keep statistics with relation to the prediction accuracy, by keeping in particular whenever:

- True Positive (TP): a record is predicted as big and it's big
- False Positive (FP): a record is small and it's predicted as big
- False Negative (FN): a record is predicted as big and it's small
- True Negative (TN): a record is small and it's predicted small

In addition, we are interested in knowing if the overall mistakes surpass any correct prediction that our model provides. To this end, we define the following metric:

$$Gain = \frac{TP - FP - FN}{TP + FN} \tag{4.1}$$

which clearly shows that the gain is positive whenever $TP > FP + FN$, namely, the number of correctly predicted big records is higher than the number of errors. The previous can be used as an indicator of when the model becomes more valuable than one of the baselines.

Gain and comparison with baselines. We consider as baseline those scheduling strategies that simply assume that all records sizes are small.

In this strategy, the accuracy value can still be high in the order of 95% or more. Other metrics such as Precision and Recall can also be applied and will give a better idea of their performance compared to accuracy.

However, from the definition of gain, we can see that this strategy will always lead to a gain of -1, which provides a baseline reference for comparison. In addition, the gain will be > 0 only whenever the number of correct predictions surpasses the number of mistakes makes. The brake-even value of the gain is 0 whereas in True positive rates or Precision is higher than zero, thus we used this gain metric for better interpretability in our problem context, but any other metric could be used instead since all of them are defined from the TP, TN, FP, FN.

Together with the accuracy, which also includes True Negatives, the gain tells the overall benefits of employing records prediction in certain database. True positive rates or Precision could also be alternatives and are more commonly used.

4.4 Results

The assessment of our model was done for 5 different degrees of dispersion or grouping, in independent experiments that were repeated 6 times, from where we reported the mean along with the standard deviation of the gain and the accuracy.

In all the experiments, the weights of our neural network model were initialized at random using a gaussian distribution with a mean zero and variance of 0.1. The learning rate for the weights update was always initialized at 0.3. As detailed in Chapter (3), the input to the neural network consisted of 10 tokens, and the output was a value between 0 to 1 denoting probabilities; values > 0.5 were indicative of a big cell.

4.4.1 Comparison to a Bloom filter and Random desicion

In order to assess the performance of the proposed model, we compare it concerning a Bloom filter. In both cases, the same list of synthetic databases was used, as well as the same queries in the same execution order. As input to the Bloom filter, we used the same first 2 tokens available for the neural networks. We configured the filter likewise recommended [2] with a 0.1 false-positive chance for a hardware configuration with limited RAM. The size of the filter's memory was calculated according to the Eq.(6.2) derived in the Appendix, in a way that was enough to store n elements (n is the total number of big records) with two independently generated hashes ($k = 2$) corresponding to the same first two tokens given to the neural network, using a false positive chance of 0.1

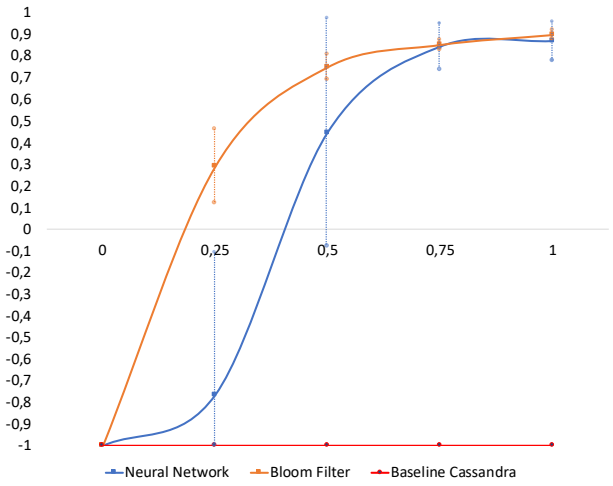

Figure 4.3: Prediction gain of neural network vs Bloom filter. Distinguishing small from large cells/values becomes more beneficial (i.e. positive gain) in databases where big records are more organized in a few columns.

The results are shown in Fig. (4.3) and detailed in Tab. (4.1). As can be seen, the performance of the Bloom filter was similar or higher compared to the performance of the neural network. In terms of variability of the results, the Bloom filter was more consistent, whereas the neural network exhibited a higher variance. Both model's gains and accuracies dropped for datasets with high dispersion of big records across the different columns.

[2] https://cassandra.apache.org/doc/latest/operating/bloom_filters.html

Model	Grouping ≡ 1-Dispertion				
	1.00	0.75	0.50	0.25	0.00
Gain (percent)					
Neural Net	86.82 ± 9.06	84.23 ± 10.62	44.70 ± 52.73	< 0	< 0
Bloom Filter	89.82 ± 1.98	85.20 ± 02.23	74.90 ± 05.86	29.02 ± 17.17	< 0
BaseLine	< 0	< 0	< 0	< 0	< 0
Accuracy (percent)					
Neural Net	99.24 ± 0.44	99.18 ± 0.68	98.83 ± 0.88	96.81 ± 1.27	71.87 ± 10.66
Bloom Filter	99.21 ± 0.31	99.24 ± 0.43	99.29 ± 0.40	98.22 ± 1.37	84.11 ± 0.06
BaseLine	93.38 ± 4.87	96.22 ± 2.41	97.53 ± 2.05	98.13 ± 1.59	94.04 ± 0.24

Table 4.1: Results of big cells prediction with thresholds fixed at 38kb.

4.4.2 Prediction latency and complexity

From the design of our neural network, it is clear that the computational cost of one prediction is $O(c/Q)$, and thus linear in the number of primkeys and column pairs c, when executed in parallel using Q threads. To measure the computational overhead of our neural network prediction, we executed multiple times our prediction routine for a set of queries and calculated the overall elapsed time. Dividing the elapsed time by the number of queries gave us the expected overhead time for a single prediction.

The previous measurements indicate that our shallow neural network takes ≈ 0.3 milliseconds (i.e. 3×10^{-4} secs) to predict a single primkey/column pair.

4.5 Discussion

From the results, it becomes that the Bloom filter performed best in comparison with the neural network. Not only in terms of the mean of gain and accuracy but also its standard deviation was smaller compared to that of the neural network. Even more important, we used a false positive rate for the filter with a value of 0.1, and decreasing this value to 0.01 or 0.001 would yield even much better results.

The neural network performance quite well in the databases where the big cells/values where rather grouped into a few columns, with gains and accuracies similar to the Bloom filter. The variance of the neural network was unexpectedly high, meaning that some results were quite accurate and others the contrary, This high variance could be attributed to a combination of the random initialization

of the neural network weights (Algorithm 1, step 2), the way that the learning rate was decreased (Eq. 3.5) and the number of queries processed during training. Additional experiments could be useful to confirm if this is the case.

4.5.1 Limitations of both models

Neural networks. Given that the prediction latency of the neural network is not high, its main limitations seem to be related to the gain and accuracy. It fails much more often than the Bloom filter in predicting the size of a cell and is also less consistent which can be seen in the variance of the results. On the positive side, the neural network size is fixed regardless of the amount of data

Bloom filter. The main limitation of the Bloom filter is known to be its memory requirements. Yet, it did not manifest in our databases nor our simulations. Nonetheless, we performed a more in-depth analysis (see details in Sec.6.1.2 of the Appendix) to analyze why this was the case. Our analysis suggests that, to reach the limits in our per-column prediction approach, we need a database with around 10 million records and 26 columns for a Bloom filter with a false positive chance of 0.01, or a database of around 80 million records and 26 columns with a false positive chance of 0.1. The higher the number of columns and the number of different columns combinations used in the different queries, the smaller the DB needed for the Bloom filter to reach its limit in this context.

Chapter 5

Conclusions

In this thesis, we showed that a neural network can predict whether a key-value will be large quite accurately, but only when certain conditions are met: large records are mostly grouped into a few columns rather than dispersed everywhere. This is very intuitive, such as in the simplest case where all big records are stored in one specific column, then any query involving that column will provide a big record.

Main conclusion. Bloom filters are great, not only to identify big records but also to identify big key-value pairs comprised within records. Nonetheless, we discussed how the detection of big records at key-value pairs levels challenge the feasibility Bloom filters, specifically with memory requirements that scale linearly with the number of different columns combinations involved in the different queries. This brings the limit of a Bloom filter to a few dozen million records when working with two dozen columns in a column family.

What is more important, the previous position neural networks as an attractive alternative for multiget queries involving different sub-set of columns, especially in the presence of medium-to-large databases that are deployed in nodes with small-to-medium memory size, which likely quickly pushes the limits of Bloom filters and solutions alike that require large amounts of RAM.

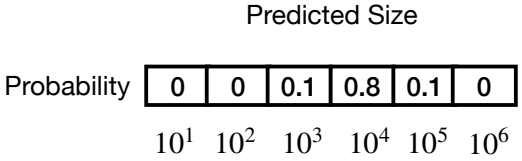

Figure 5.1: Unlike other models, neural networks allow predicting a finer information about the record/cells sizes, not only if its small or big.

Furthermore, unlike Bloom filters, neural networks can not only predict if the

record or its cells are small or big, but also provide a finer type of prediction, as shown in Fig.(5.1), which can provide an even more detailed information for the scheduler to exploit during jobs planning and replica selection.

Shortcomings. Two main shortcomings can be identified in this work. Firstly, it is unclear whether real-world databases, if some or all or how often, have some degree of organization of big cells within its columns, although we believe that columns names provide a natural organization or grouping for values (ex: a column with name "video"). Hence it would be necessary to assess our model in real-world databases, preferably in those in the range of 50 to 100million records.

Secondly, it is still unclear if or how, having more granular information about the record size at the column-wise level, will materialize into a practical speed-up and tail latency reduction. For this, the experiments should be conducted involving YSCB and a real Cassandra cluster.

Future works. In future works, the aforementioned shortcomings could be addressed. Besides, as improvements, we could consider improving the neural network design in terms of the number of hidden units and layers to see how much it can improve while keeping a low inference time latency. A practical extension could be to consider different settings where the same neural network is shared across multiple master nodes or is trained used common information shared via gossip protocols.

Chapter 6

Appendix

6.1 Bloom filters

Bloom filters can be considered as a data structure that relies on a kind of data compression to store elements from the same set. A special property of Bloom filters is that if an element has been already added, it will always be found on this set. As a consequence, the False Negative rate of a Bloom filter is always zero. A negative aspect of such a filter is that no element can be removed from this set once it has been added unless one reset the filter completely and start all over again.

In its most simplistic form, a Bloom filter is just an array of n booleans that constitute the filter's memory. Given an element, represented as a list of k independently generated tokens with values restricted from 1 to n, the filter can add such an element into the set by setting to True each position of its memory according to each of the k tokens.

In order to determine certain element is within the set or not, also from a given representation of this element as a list k tokens, the filter just check each of these k positions within its memory to confirm that they are all set to True.

6.1.1 False Positives rate in Bloom filters

The probability of getting a false positive from such a filter can be derived from the filter's memory size n, the number of k tokens, and the maximum number of elements potentially to be included in the set m.

The performance of a Bloom filter depends on a trade-off between the memory size n and the number of elements m expected in the set, with k being kind of "assurance" factor.

Filter memory size indexing primkey or (primkey, column) tuples The upper bound on False positive (FP) error rate of a bloom filter can be established from these parameters as follows:

$$\text{FP rate} \quad \leq \quad \left(1 - \mathrm{e}^{\frac{-\mathrm{km}}{\mathrm{n}}}\right)^{k} \tag{6.1}$$

where, from a desired FP rate pre-set or fixed beforehand and $k \geq 1$, we can solve the ratio between the filter size and the number of elements:

$$\frac{m}{n} \quad = \quad \frac{-\log(1 - \text{FP rate})}{k^2} \tag{6.2}$$

6.1.2 Practical limits of Bloom filters for detection of big cells

In order to analyze the limits of Bloom filters when tracking big cells instead of big records, we considered a practical scenario including a hardware setup for the master nodes.

We followed the hardware recommendations for Cassandra provided by the Apache Foundation [17] and DataStax[18], and selected an Amazon EC2 instance of "i2.large" type that comes with 30.5GB of RAM. In practice, Cassandra takes around 12GB of RAM when running for its filters and tables and its also desired to keep 6GB for the OS to fulfill monitoring and maintenance tasks, which leaves in total 12GB free for our Bloom filter.

Notice also that in our experimental setup, we have that $k = 2$ and the big records threshold set in a way that in each experiment, no more than 20% of all cells are considered big. The number of columns per family column was set to 26 whereas the average number of columns requested in a query was 7 ± 5.

False positive chance. The previous experimental setting and hardware configuration, together with Eq.(6.2), effectively set the limits of our bloom filter in practice. After this limit, the filter will start to fail and give many more false positives than expected.

As can be seen, for a false positive chance of 0.1, this kind of filter will reach its limit at 72.5 million of records (or rows). Conversely, for a false positive chance of 0.01, the limit will be reached at 6.8 million of records (or rows).

Bibliography

[1] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[2] Vikas Jaiman, Sonia Ben Mokhtar, Vivien Quéma, Lydia Y Chen, and Etienne Rivière. Héron: Taming tail latencies in key-value stores under heterogeneous workloads. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 191–200. IEEE, 2018.

[3] Shuang Chen, Shay GalOn, Christina Delimitrou, Srilatha Manne, and Jose F Martinez. Workload characterization of interactive cloud services on big and small server platforms. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 125–134. IEEE, 2017.

[4] Vikas Jaiman, Sonia Ben Mokhtar, and Etienne Rivière. Tailx: Scheduling heterogeneous multiget queries to improve tail latencies in key-value stores. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 73–92. Springer, 2020.

[5] Diego Didona and Willy Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 79–94, 2019.

[6] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[7] Waleed Reda, Marco Canini, Lalith Suresh, Dejan Kostić, and Sean Braithwaite. Rein: Taming tail latency in key-value stores via multiget scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 95–110, 2017.

[8] Cassandra version 3.4. `https://issues.apache.org/jira/browse/CASSANDRA-10657`.

[9] Datastax. `https://www.datastax.com`.

[10] Hector. http://hector-client.github.io/hector/build/html/index.html.

[11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[12] Andrew S Tanenbaum, Jorrit N Herder, and Herbert Bos. File size distribution on unix systems: then and now. *ACM SIGOPS Operating Systems Review*, 40(1):100–104, 2006.

[13] Simon Haykin and Neural Network. A comprehensive foundation. *Neural networks*, 2(2004):41, 2004.

[14] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[15] Databases used in experiments. https://github.com/laritzalba/neural-network-predictor.

[16] Sumita Barahmand and Shahram Ghandeharizadeh. D-zipfian: a decentralized implementation of zipfian. In *Proceedings of the Sixth International Workshop on Testing Database Systems*, pages 1–6, 2013.

[17] Cassandra hardware choices. https://cassandra.apache.org/doc/latest/operating/hardware.html.

[18] Datastax documentation. https://docs.datastax.com/en/dse-planning/doc/planning/capacityPlanning.html.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**