Pred-Cache: A Predictive Caching Method in Database Systems

Omar El Zarif, Safwat Hassan
(oelzarif,shassan)@cs.queensu.ca
Queen's University
School Of Computing
Kingston, Ontario, Canada

Ying Zou
ying.zou@queensu.ca
Queen's University
Department of Electrical and Computer Engineering
Kingston, Ontario, Canada

Calisto Zuzarte, Vincent Corvinelli, Mohammed Al Hamid
(calisto,vcorvine)@ca.ibm.com
mohammed.alhamid@ibm.com
IBM Canada Ltd

ABSTRACT

The performance of large-scale systems (LSS) depends heavily on the time consumed in retrieving users' data from the databases. The database management system (DBMS) is essential to handle the storage and retrieval of users' data. Recent studies show that performance degradation in retrieving users' data can cause a severe revenue loss. Hence, improving the performance of the DBMS is essential for maintaining and enhancing user experience.

Query caching is a technique employed by the DBMS that presents immense improvements to the overall performance of the system. Prior work improves query caching techniques by maximizing the reuse of the cached queries (e.g., deciding on the beneficial queries to cache and deciding on the cache eviction and replacement policies). However, the existing work is tailored to specific server query languages and lacks in the adaptation to the different changing factors in the system, such as the occurrences of queries, the time of their occurrence, and query coupling.

In this work, we propose a predictive database caching framework, which can be deployed as a middleware layer independently from the database system. Our framework uses deep learning models to predict expensiveness (in terms of execution time) and the occurrences of queries to guide the caching process. We evaluate our framework using the TPC Benchmark DS (TPC-DS) database where we generate a 50GB database with 100,000 queries. Remarkably, our framework improves the cache hit ratio by 6% to 29% over the existing query caching mechanisms in the different benchmark scenarios that simulate the different types of query histories.

KEYWORDS

database systems, deep learning, neural networks, query caching

ACM Reference Format:
Omar El Zarif, Safwat Hassan, Ying Zou, and Calisto Zuzarte, Vincent Corvinelli, Mohammed Al Hamid. 2020. Pred-Cache: A Predictive Caching Method in Database Systems. In *CASCON 2020, November 2020, Toronto, Ontario, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

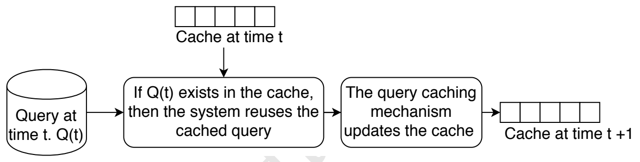
Cache at time t
Query at time t. Q(t)
If Q(t) exists in the cache, then the system reuses the cached query
The query caching mechanism updates the cache
Cache at time t +1

Figure 1: An overview of the query caching mechanism.

1 INTRODUCTION

The composition of modern software systems, particularly large-scale software systems, relies heavily on the interaction of the software with the database system to process the imposed huge amounts of data. *Query caching* is considered an essential technique to improve the performance of the software systems by improving the execution times in the database [8, 16, 17, 55].

Query caching increases the performance of a system as it eliminates the re-execution of queries. Figure 1 shows an overview of the query caching mechanism. As shown in Figure 1, the new coming query at time t Q(t) is matched against the cache to eliminate the re-execution of queries in the system [38, 59]. Then, the query caching mechanisms decide whether to update the content of the cached queries if Q(t) is not already cached. The decision to cache a query should aim to maximize the reuse of the query, and thereby increase the system performance while adhering to the memory constraints of the system [36, 44]. Query caching is an extensively studied topic in database systems [4, 20, 38, 40, 42, 43, 59]. It has been approached by different aspects, such as identifying beneficial queries to cache in a system and deciding the optimum query eviction or replacement policies [4, 5, 40, 42].

Prior work proposes **reactive caching** mechanisms that maintain the cache content by controlling the cache eviction mechanisms, such as Least Recently Used (LRU) [36], Least Frequently Used (LFU) [44], and Least Recently Frequently Used (LRFU) [37] mechanisms. The content cached by applying reactive mechanisms is driven by the current access patterns in the system (e.g., caching the most frequently used queries). The reactive caching mechanisms are easy to implement. They proved their efficiency in the cases of repetitive data in the system. However, under constrained cache sizes these strategies lead to thrashing in the system [14]. The mechanisms are passive and slow as they start by caching a huge amount of insignificant data until popular data patterns start to emerge [10, 12, 61].

In contrast, **proactive caching** mechanisms are introduced to solve the slow responsiveness of the reactive caching mechanisms [40, 43, 62]. Proactive caching tends to evaluate the cost of queries (in terms of execution time) before deciding on the caching process.

For example, prior work relies on the execution time estimation using query execution plans (QEPs) to cache the queries with a high-execution time [26, 43]. However, query execution plans are system-oriented and require manual work and tuning of the database system for better cost estimation [13, 31, 52].

To introduce a cache between the software's query request and its execution in the database system, we present a framework for **proactive query caching** in database systems as follows. First, the framework **predicts the upcoming queries** using a recurrent neural network (RNN) [47]. Second, the framework **predicts the cost** (as the execution time and memory requirements) of the upcoming queries using a feed-forward neural network (FFNN) [65]. Finally, our framework caches the upcoming queries with long-execution time and low-memory requirements predictions (i.e., **prefetch and cache the cost-efficient queries**).

We assess the impact of using our framework on the performance of the system by calculating the cache hit ratio [71]. In particular, we compare the performance of our framework with the traditional reactive caching mechanisms (i.e., LRU, LFU, and LRFU mechanisms). We benchmark our work following various generated scenarios of query executions (e.g., running queries in sequential order) in the system. Each scenario was produced from the TPC-DS workload benchmark [49, 56] where we generated 100,000 queries on a 50GB database to simulate our study. In particular, we analyze the following research questions (RQs) to evaluate our framework:

RQ1: How accurate are the cost estimation and the prefetching functions?

The feed-forward neural network exhibits high accuracy in the prediction of memory and runtime with AUC above 0.9. In addition, the recurrent neural network exhibits high accuracy in predicting the next queries with a perplexity score of 25.

RQ2: What mechanisms exceed in terms of the cache hit ratio in the different benchmark scenarios?

The prefetching of the predicted queries (i.e., using RNN) and the prefetching of the cost-efficient queries (i.e., using both RNN and FFNN) outperforms the traditional reactive caching mechanisms in all of our benchmark scenarios.

RQ3: What is the percentage of improvement of our framework over the traditional mechanisms?

Using our proactive caching framework, the percentage of improvement in the cache hit ratio ranges from 6% to 29%, on average, over the traditional reactive caching mechanisms (i.e., LRU, LFU, and LRFU).

Our main contributions can be described as follows:

(1) We present a proactive caching framework that combines the query cost estimation and the prefetching of future cached queries in the system.
(2) Our framework can be deployed as an independent in-memory middleware layer between any software and database as the queries are abstracted. Hence, it does not require any configuration or modification in the database system.
(3) The benchmark results exhibit improvements of our proactive caching mechanisms over the work of the traditional reactive caching mechanisms.

Paper Organization: The rest of the paper is organized as follows, we provide a background on FFNN and RNN in Section 2. We describe the data collection process in Section 3. We describe our framework in Section 4. We showcase the results in Section 5. We discuss the threats to validity in Section 6. We present the related work in Section 7. Finally, we conclude the paper in Section 8.

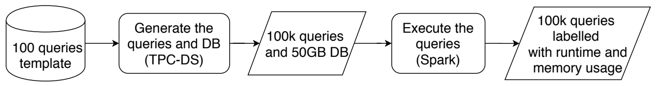
100 queries template
Generate the queries and DB (TPC-DS)
100k queries and 50GB DB
Execute the queries (Spark)
100k queries labelled with runtime and memory usage

Figure 2: The data labeling process.

2 FFNN AND RNN BACKGROUND

We use the RNN and the FFNN to predict the upcoming queries and prefetch the cost-efficient queries.

The FFNN is the basis of supervised deep learning problems. FFNN is a form of the basic neural networks where the information flow from one layer to the next layer in a unidirectional manner, unlike recurrent neural networks where the information flow bidirectionally from next and previous layers [65]. The FFNN works by estimating a function f^*. The problem is defined as a classifier $y = f^*(x)$ that maps an input x to a class y. The FFNN defines a function $y = f(x, \theta)$, where it learns the parameter θ to approximate the value of $f^*(x)$. The input x defines the first layer, the function f represents the intermediate layers, and the output y defines the last layer. The information flows from the first to the last layer, where the parameter θ is evaluated at the output layer, and recalculated for the next $f^*(x)$ estimation [23].

The RNN is a descendant of the FFNN. The work of the RNN suits the problem of classifying a sequence of inputs to a sequence of outputs. The RNN maps a sequence of inputs $x ... x_n$ to outputs $y ... y_n$. Where the function $y = f^*(x)$ maps each x to y over the time-steps of $t_1 ... t_n$. To approximate the function f^*, the RNN defines the functions $y_t = f(x, \theta_t + \theta_{t-1})$, where the parameter θ is reevaluated at each time step to approximate the value of $f^*(x)$ [23, 47].

3 DATA COLLECTION

Our study is based on the TPC-DS database [49]. The database represents a data warehouse that revolves around online analytical processing tasks. TPC-DS database emulates a decision-support system of a retail product supplier with 100 defined queries that represent reporting jobs, which covers all the database tables. The database schema constitutes of 24 tables with an average of 18 columns per table, and 108 foreign keys, which signifies the complexity of the system.

The TPC-DS database employs benchmarking capabilities where it allows scaling and augmenting the database size and queries [56, 57]. Neural networks require data samples in the order of thousands to be trained [6, 67]. Hence, we augmented the number of original queries in the TPC-DS database to attain a sufficient number for the training and validation of our neural networks.

The augmentation of queries uses each of the 100 most used queries as *templates*. We augment to 1,000 queries from each template query by replacing the conditional operations that follow the WHERE clause in the query with random values. In the end, we generate a 50 GB SQL database, with 100 thousand queries augmented from the 100 most used queries in TPC-DS.

Table 1: The data set description.

Data Set	# of queries	# of unique queries	Description
Full data	100,000	29,000	The whole number of generated queries.
Training data	70,000	23,000	Data used to train and validate the FFNN.
Benchmarking data	30,000	6,000	Data used to train the RNN and benchmark our approach.

We showcase the labeling process in Figure 2. The queries need to be labeled by their execution time and memory consumption to train our FFNN model, as the FFNN serves as a cost estimator. The FFNN model predicts the runtime and memory consumption of a new incoming query to guide the caching decision. We labeled the queries by loading the TPC-DS dataset to Spark [74]. Then, we execute the 100 thousand queries using Spark framework. Spark is an open-source parallel computing framework that allows importing large-scale databases and the parallelization of the execution of multiple queries in the system. This process allows an accurate memory and runtime recording in terms of milliseconds and bytes.

The initial data collection process allows us to establish a benchmark of 100 thousand queries that serve as the basis of our approach for training, testing and benchmarking our two neural networks in different scenarios. As shown in Table 1, the 100 thousand queries were cut into 70% to train and validate the FFNN and 30% to train the RNN. The same 30% of the queries were also used to benchmark our approach.

4 OVERVIEW OF OUR FRAMEWORK

Figure 3 shows an overview of our approach. For each incoming query in the system, first, our framework *generates the query embeddings* that converts the input query text to an embedding vector. Second, our framework *predicts the next upcoming five queries* using the RNN. We chose the number five since the RNN is able to correctly predict the next five queries with high accuracy of 95%, on average, in our hyperparameter tuning experiments. The accuracy in predicting more than five upcoming queries drops by 15% when that number is incremented by five progressively (i.e., next 10, next 15, etc.). We discuss that process further in Section 6. Third, the FFNN *predicts the cost* (i.e., the runtime and memory consumption) of the upcoming queries. Finally, the framework *prefetches the cost-efficient queries* among the upcoming ones. Our framework evicts queries based on any reactive caching algorithm (e.g., LRU, LFU, and LRFU) when the cache size would not fit the upcoming queries. In the next sections, we describe the steps for generating query embeddings, the architecture of the used neural networks (i.e., RNN and FFNN), and the benchmark process.

4.1 Generate Word Embeddings for Queries

The FFNN processes the *text embeddings* of queries as an input. The FFNN is trained to estimate the memory consumption and runtime of the queries. Transforming the query text to query embeddings inherently guards the meaning of the operations of the query which makes the cost estimation achievable.

To extract embeddings for each query, we employ Word2Vec [21] to transform each word in the query text to a 64 bits array. The Word2Vec algorithm works by reducing each word in the input (i.e., corpus) into a unique vector. The vectors are positioned in a hyperdimensional space where words that share common contexts are positioned close to each other [60]. To represent the overall embedding of a query, the vectors for each word in a query text are summed and normalized using L2 normalization [70, 72]. To validate the correctness of the final query embeddings, we use the cosine similarity metric [53]. The cosine similarity measures the angle between two vectors projected in a hyperdimensional plane where a value of 1 signifies a total overlay of the two vectors meaning that the vectors are identical [69].

To verify the contextual similarity of the generated embeddings that are derived from the same template, we measure the cosine similarity among the generated 1,000 queries of each template of the 100 TPC-DS query templates. The mean cosine similarity varies from 0.85 to 0.92 in the 100 query templates, which proves the correctness of our embedding extraction.

The RNN extracts the patterns from a sequence of executed queries. Hence, RNN needs to represent every query as a unique value in the input sequence of queries [7]. To represent the queries as unique values, we hashed the whole query text into 256 bits hash using SHA256 [58]. The algorithm secures collision resistance giving a unique hash for each text [33], thus securing unique words for each unique query.

4.2 Architecture of Neural Networks

The Feed-Forward Neural Network. The FFNN [65] is used to estimate the cost of executing a query. It consists of two hidden layers with rectified linear units (ReLU) [48]. The input layer of the neural network is of size 64 bits and accepts the aforementioned extracted embeddings of the queries. The neural network has two outputs layers: the first layer is used for predicting the memory consumption of the query, and the second layer is used to predict the execution time of the query. The two outputs employ a Softmax activation function. Softmax serves a multi-class probability prediction at the last layer of the neural network by normalizing the outputs by the sum of their exponents to represent them as a probability distribution [22].

To adhere to the usage of Softmax as output layers, we quantized the runtime and memory consumption. Quantization serves in increasing the accuracy of predictions in neural networks by predicting the quantiles (i.e., the classes) instead of predicting a distribution of real numbers [18]. We quantize the runtime and memory distributions to three classes (e.g., low, memory, and high). The first class represents the quantile from 0% to 33% that is the low-level memory consumption or runtime, the second class describes the 34% to 66% quantile that is the medium-level memory consumption or runtime, and the last class from 67% to 100% quantile represents the high-level runtime or memory consumption. This process avoids predicting the exact runtimes or memory consumption to a classification task that predicts ranges of runtime or memory consumption described as low, medium, and high.

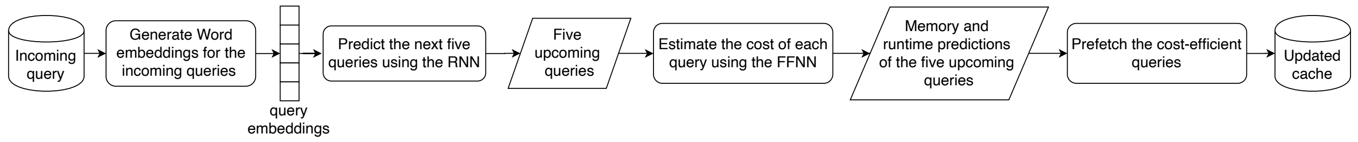
Incoming query
Generate Word embeddings for the incoming queries
query embeddings
Predict the next five queries using the RNN
Five upcoming queries
Estimate the cost of each query using the FFNN
Memory and runtime predictions of the five upcoming queries
Prefetch the cost-efficient queries
Updated cache

Figure 3: An overview of our query caching framework.

The FFNN is trained on 70 thousand of the 100 thousand generated queries (i.e., training and validation dataset) as shown in Table 1. We guarantee the uniqueness in the training dataset to eliminate overfitting in the neural network that may occur if the same data appears in the training and the validation process. Hence, we train and validate the FFNN on the 23 thousand unique queries of the training dataset after filtering the 70 thousand queries from duplicates.

This data (i.e., the 23,000 queries) is split into 80% training and 20% validation split using 10 folds cross-validation. Each fold is split into the same percentage (80%-20%) as we trained and validated the FFNN.

The Recurrent Neural Network. The neural network consists of two long short term memory (LSTM) layers [19] with 10 units per layer. The LSTM layer serves the objective of predicting future queries at each time step. The time steps are the index of the query in the sentence of queries. The LSTM is widely used to process a sequence of data as it solves the vanishing gradient problem when the sequence of data lingers in its length [64].

As shown in Table 1, we train the RNN on the 30 thousand queries that form the benchmark dataset. The vocabulary of the dataset consists of 6 thousand unique queries after filtering the 30 thousand queries from duplicates. Hence, the output layer of the RNN consists of a Softmax layer with 6 thousand classes that represents the vocabulary of our corpus (i.e., the number of unique queries in the benchmark dataset). The Softmax layer serves as a probability prediction of the most suitable word (i.e., query) to occur at each time step. We take the highest five probabilities to predict the upcoming five queries from the previous queries.

4.3 Generating The Different Datasets

The datasets for training the RNN is formed by hashing the query texts and generating the order of queries under three different scenarios. Each scenario represents a different plausible real-life occurrence of queries. The RNNs identify the occurrences of queries to form patterns that lead to the prediction of upcoming queries. The occurrences are represented by the repetition of the same hash.

Sequence Dataset. The sequence dataset represents the queries that occur sequentially in real-life scenarios. For example, some queries might be part of a task where they are always executed consecutively. This could be part of a reporting job where different queries on different tables are executed in sequence to extract the data. To generate the sequence dataset, we choose a random query from each of the 100 templates in our benchmark dataset sequentially until we reach all the queries in the benchmark dataset.

Batch Dataset. The batch dataset represents a scenario where the queries might co-occur as a batch of jobs. Similarly to the sequential scenario, the batch represents jobs where a group of queries is repeated in a sequence. The generation of the batch dataset is similar to the sequence dataset where the only difference is instead of choosing one query of each template to run sequentially, we chose n numbers of queries of each template from the benchmark dataset randomly to run sequentially.

Random Dataset. The random dataset is formed by shuffling the whole benchmark dataset. We chose random scenario as a stress test for our framework. The sequential and batch histories will guarantee patterns that will enhance the work of the prefetching mechanism (i.e., the RNN). In contrast, we test our prefetching mechanism when the scenario is formed by a random occurrence of queries. An effective prefetching mechanism should not suffer from a major degradation in performance in this scenario.

4.4 The Benchmark Process

The benchmark process relies on three different workloads that are generated similarly to the datasets for training the RNN. To simulate the work of the RNN and the FFNN in practice, we generate from the benchmark dataset another sequential, batch, and random datasets. The benchmark workloads were also generated from the same data that was used to train the RNN to guarantee that the same vocabulary (i.e., corpus for the RNN) is consistent. But the generation guarantees a different order of occurrences of queries. Hence, the training and the benchmark data for the RNN are different.

We evaluate the results on each generated benchmark dataset on its own comparing the cache hit ratio of the proactive caching mechanisms (e.g., the RNN, and the combination of the RNN with the FFNN) and the reactive caching mechanisms (e.g., LRU, LFU, and LRFU). The execution time in the system should be reduced with the presence of the same incoming query in the cache, while the cache hit ratio increases when the same incoming query is present in the cache. Thus, the cost estimation and the prefetching mechanisms are compared against the traditional mechanisms LRU, LFU, and LRFU. The cache hit ratio is recorded gradually while we process the queries from each generated benchmark dataset into the system and either execute them or reuse the queries from the cache if they are present as shown in Figure 4.

4.5 Caching Decision for the Different Mechanisms

As shown in Figure 4, we compared each of our mechanisms (i.e., the RNN plus the combination of the RNN and the FFNN) against the traditional reactive caching mechanisms (i.e., LRU, LFU, and LRFU) to validate the performance of each our mechanisms.

The **RNN** predicts for each incoming query the next five incoming queries and caches them. The prefetching mechanism relies on

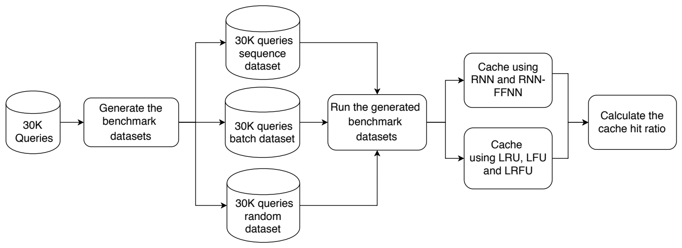
30K Queries
Generate the benchmark datasets
30K queries sequence dataset
30K queries batch dataset
30K queries random dataset
Run the generated benchmark datasets
Cache using RNN and RNN-FFNN
Cache using LRU, LFU and LRFU
Calculate the cache hit ratio

Figure 4: The benchmark process.

the patterns of occurrences of queries learned from the training process.

We combined the recurrent **neural network and the FFNN (RNN-FFNN)** to prefetch and cache expensive queries only. After predicting the next five upcoming queries, each query would fall under the prediction of the FFNN to estimate the runtime and memory consumption. The mechanism prefetches and caches the queries with medium or high runtime and low memory consumption when the cache is 70% full. The condition is relaxed when the cache does not reach that threshold to cache any incoming query. We have tested this mechanism under different thresholds and found that 70% is the best suitable threshold under different cache sizes. The low, medium, and high predictions are the results of the quantization of the real values as described in the data processing to three classes or quantiles.

The two caching mechanisms that serve the prefetching (RNN), and the cost estimation and prefetching combined (RNN-FFNN) would guide the decision for caching the queries, but their work is incomplete with the absence of caching eviction mechanisms. Hence, we implemented the work of LRU, LFU, and LRFU to be employed as cache eviction policies, either on their own or with the combination of our proactive caching mechanisms (RNN, and RNN-FFNN).

We benchmark our results and compare the usage of each traditional mechanism with its modified version respectively (e.g., comparing LRU with the RNN that uses LRU as an eviction mechanism).

5 RESULTS

In this section, we present the motivation, the approach, and the results of the studied research questions.

5.1 RQ1: How accurate are the cost estimation and the prefetching functions?

Motivation: Before employing our two neural networks (FFNN and RNN), we need to evaluate the accuracy of our work in cost estimation and the prediction of the next queries. Hence, we can ensure the correctness of the predictions of our two neural networks to employ them in prefetching the cost-efficient queries.

Approach: To evaluate the FFNN we rely on the area under the curve (AUC) metric [24, 25]. Our model outputs probability predictions for runtime and memory consumption. AUC tests the fit of our model in probability predictions with different thresholds. A model with an AUC value of 0.5 signifies a random prediction model. A model with an AUC value of 1 signifies a model with perfect true positives and true negatives predictions, while a model with an AUC value of 0 signifies a model with perfect inverse probability predictions [51].

We train our model on 10 folds cross-validation, thus the AUC is calculated as the average AUC of the 10 folds on the validation sets [45]. The 10 fold cross-validation reshuffles the data and retrains the network from scratch on each fold. Hence, the average AUC demonstrates the effectiveness of our model in different data distributions.

To ensure that the model is not showing bias in the result we enforced weight class distribution in the training sets. The weights added to the cross-entropy loss function (i.e., the neural network loss function) ensure that the loss function is weighted to overcome the biases in predicting one certain class since it is more dominant [35, 73]. The average class weight across the 10 sets for the runtime is 1.03 for class 0, 1.01 for class 1, and 0.95 of class 2. (The classes 0 to 2 represent the low, medium, and high classes respectively). The average class weight for the memory is 0.80 for class 0, 1.92 for class 1, and 0.81 for class 2.

The RNN is evaluated based on the perplexity score of the predictions. Given a sequence of input queries, the RNN predicts the next query to occur. The RNN model uses a sequence loss function (e.g., cross-entropy function) that predicts the most suitable query from the vocabulary of queries at each time step. The model can then be evaluated as a sequence to one prediction error by using the perplexity metric. The perplexity measures the fit of the query distributions on unseen data. The measured value represents the inverse of the prediction probability of what is the next query to occur. The perplexity is an unbounded function that can span to infinity [30].

Results: We recorded the AUC for the runtime classification and the AUC for the memory consumption classification. The runtime AUC is 0.94 and the memory consumption AUC is 0.92. The AUC function is resilient for varying class distributions, so the result does not show biases [29]. The AUC was recorded as the average of the 10 folds cross-validation on the validation sets. The training and testing of the FFNN are produced on the training dataset, as described in Section 3.

The RNN achieves an average perplexity score of 25 on the three benchmark datasets. The lowest perplexity measure is 1, it means that the fit is perfect, while a good language model should have a perplexity lower than 200 [9].

Summary of RQ1

The FFNN exhibits high accuracy in the prediction of memory and runtime with AUC above 0.9. In addition, the RNN exhibits high accuracy in predicting the next queries with a perplexity score of 25.

5.2 RQ2: What mechanisms exceed in terms of the cache hit ratio in the different benchmark scenarios?

Motivation: In the previous RQ, we observe that our approach can accurately predict the upcoming queries. In this RQ, we aim to understand whether our proactive caching mechanism outperforms the existing reactive caching mechanisms (i.e., LRU, LFU, and LRFU). Understanding the optimum caching mechanism can help system owners better select suitable caching mechanisms to improve the performance of the software systems.

Approach: The performance of query caching mechanisms can be impacted by the cache size. Hence, we evaluate the performance of the studied caching mechanisms using different cache sizes (i.e., small, medium, large, and the normal cache size). We design the different cache sizes as follows. First, we set up the *cache limit*. The cache limit represents a relative portion of the workload data that can be stored in the memory. Caching the full data of all users in a large-scale system can be very expensive and practically infeasible [54]. Hence, we choose 30% (i.e., 2,000 queries) of the number of unique queries in our benchmark dataset as the cache limit.

Then, given a certain cache limit, we define the used cache sizes as follows. The *small cache size* is calculated by multiplying the cache limit (i.e., 2,000 queries) with the minimum memory size of a query. The *medium cache size* is calculated by multiplying the cache limit with the median memory size of a query. The *large cache size* is calculated by multiplying the cache limit with the maximum memory size of a query. Finally, the *normal cache size* is calculated by multiplying the cache limit with the average memory size of a query. For every cache size, we recorded the cache hit ratio using three different benchmark datasets (i.e., sequence, batch, and random).

We benchmark two proactive mechanisms (1) fetching the upcoming queries (i.e., using RNN) and (2) fetching the cost-efficient queries (i.e., using RNN and FFNN named as *RNN-FFNN*). In addition, we benchmark three reactive approaches LRU, LFU, and LRFU. As described in Section 4, proactive caching can use different eviction mechanisms. Hence, we benchmark every proactive caching mechanism (i.e., RNN and RNN-FFNN) with the three reactive approaches LRU, LFU, and LRFU. We represent the RNN that uses the LFU cache eviction mechanism as *"RNN + LFU"*, the RNN that uses LRU as *"RNN + LRU"*, and the RNN that uses LRFU as *"RNN + LRFU"*. Similarly, we represent the RNN-FFNN that uses the LRU cache eviction mechanism as *"RNN-FFNN + LRU"*, the RNN-FFNN that uses LFU as *"RNN-FFNN + LFU"*, and the RNN-FFNN that uses LRFU as *"RNN-FFNN + LRFU"*.

Results: Benchmarking the Sequence Scenario. As shown in Table 2, the RNN-FFNN and the RNN exhibit the best results with the normal cache size. The cache hit ratio ranges from 0.50 to 0.51 when using RNN or RNN-FFNN combined with any reactive caching mechanism including LRU, LFU, and LRFU.

As shown in Table 2, in the small cache sizes, the RNN + LRU and the RNN + LRFU perform the best. In the medium and large cache sizes, adding the notion of cost estimation ameliorates the performance of prefetching. RNN-FFNN on top of the two reactive caching mechanisms LRU and LFU exhibits the best cache hit ratios of 0.54 and 0.69 for medium and large cache sizes respectively.

Table 2: The cache hit ratio of the sequence scenario with different caching mechanisms and varying cache sizes.

Mechanisms	Small cache	Medium cache	Large cache	Normal cache
LRU	0.12	0.41	0.62	0.37
LFU	0.17	0.48	0.66	0.43
LRFU	0.12	0.41	0.62	0.38
RNN + LRU	**0.34**	**0.54**	0.68	**0.51**
RNN + LFU	0.29	**0.54**	0.68	0.50
RNN + LRFU	**0.34**	**0.54**	0.68	**0.51**
RNN-FFNN + LRU	0.32	**0.54**	**0.69**	**0.51**
RNN-FFNN + LFU	0.29	**0.54**	**0.69**	**0.51**
RNN-FFNN + LRFU	0.32	0.53	0.68	**0.51**

Table 3: The cache hit ratio of the batch scenario with different caching mechanisms and varying cache sizes.

Mechanisms	Small cache	Medium cache	Large cache	Normal cache
LRU	0.14	0.41	0.62	0.39
LFU	0.23	0.49	0.66	0.46
LRFU	0.14	0.41	0.63	0.39
RNN + LRU	0.27	0.49	0.69	0.48
RNN + LFU	**0.37**	**0.57**	0.69	**0.54**
RNN + LRFU	0.27	0.50	0.69	0.48
RNN-FFNN + LRU	0.17	0.39	0.67	0.41
RNN-FFNN + LFU	0.28	0.51	**0.70**	0.52
RNN-FFNN + LRFU	0.18	0.40	0.68	0.41

In summary, the RNN and the RNN-FFNN perform the best. In the small cache sizes, the RNN performs better than the combination of RNN and FFNN. The RNN-FFNN performs the best on medium and large cache sizes.

Prefetching all the co-occurrent queries would save time more than prefetching the expensive ones only. This is due to some queries that might occur frequently but are not expensive. In large cache sizes, prefetching more co-occurrent queries would cause thrashing, thus leading to caching unnecessary queries. The cost estimation function (i.e., using FFNN) would tune the work of the prefetching mechanism as it combines the expensive and the co-occurrent queries.

Summary of benchmarking the sequence scenario

The prefetching function solely (i.e., RNN) or combined with the cost estimation function (i.e., RNN-FFNN) exhibits the best results in terms of the cache hit ratio in the sequence scenario.

Results: Benchmarking the Batch Scenario. The cache hit ratio results summarized in Table 3 indicates that the prefetching mechanism using the RNN + LFU performs the best with a 0.54 cache hit ratio on normal cache size. The RNN-FFNN + LFU comes in second place with a cache hit ratio of 0.52 on normal cache size. The cache hit ratio for the RNN + LFU outperforms all the other mechanisms on all the cache sizes except on the large cache size, where RNN-FFNN + LFU performs slightly better.

Table 4: The cache hit ratio of the random scenario with different caching mechanisms and varying cache sizes.

Mechanisms	Small cache	Medium cache	Large cache	Normal cache
LRU	0.13	0.41	0.62	0.38
LFU	0.18	0.46	0.66	0.43
LRFU	0.13	0.41	0.63	0.38
RNN + LRU	0.15	0.46	0.66	0.41
RNN + LFU	**0.19**	**0.51**	0.67	**0.45**
RNN + LRFU	0.15	0.46	0.66	0.42
RNN-FFNN + LRU	0.14	0.45	0.66	0.41
RNN-FFNN + LFU	**0.19**	**0.51**	**0.68**	**0.45**
RNN-FFNN + LRFU	0.18	0.45	0.66	0.41

In summary, for the batch scenario, the RNN + LFU performs the best. In smaller and medium cache sizes, the RNN solely performs better than the combination of RNN and FFNN. This is due to the similar reasons in the sequence scenario that with bigger cache capacities caching more content can cause thrashing. Hence, the usage of FFNN to estimate the expensiveness and cache the expensive queries would tune the performance of the system on larger cache sizes.

Summary of benchmarking the batch scenario

The prefetching function using RNN solely exhibits the best results in terms of the cache hit ratio. The combination of the prefetching and the cost estimation functions (i.e., using the RNN-FFNN) comes in second place in the batch scenario.

Results: Benchmarking the Random Scenario. Following the cache hit ratio results in Table 4, the combination of the RNN-FFNN over LFU reaches 0.45 cache hit ratio, as well as the RNN solely.

As listed in Table 4, in small cache sizes, the RNN and the combination of the RNN-FFNN over LFU perform the best with a cache hit ratio of 0.19. For the medium cache sizes, the RNN and the RNN-FFNN over LFU show the best performance with a cache hit ratio of 0.51. Finally, for the large cache sizes, the RNN-FFNN over LFU performs the best with a cache hit ratio of 0.68.

On average, the two mechanisms (RNN or RNN-FFN) over LFU perform the best. In a random scenario, the prefetching mechanism suffers from degradation compared with the other scenarios. This is due to the absence of recognizable patterns for the recurrent network to learn. However, adding the notion of prefetching still improves the performance over the reactive caching mechanisms. In addition, adding the notion of cost estimation helps to improve the prefetching in larger cache sizes.

Summary of benchmarking the random scenario

The prefetching function (RNN) or the combination of the prefetching and the cost estimation functions (i.e., using the RNN-FFNN) on top of LFU outperforms the reactive caching mechanisms (e.g., LRU, LFU, and LRFU) in the random scenario.

Table 5: The percentage of improvement in cache hit ratio of our framework over the traditional mechanisms.

Baseline mechanism	% of improvement		
	Sequence	Batch	Random
LRU	35.4%	5.4%	8.7%
LFU	17.3%	7.3%	8.3%
LRFU	34.7%	6.1%	5.6%
Average	29.1%	6.3%	7.5%

5.3 RQ3: What is the percentage of improvement of our framework over the traditional mechanisms?

Motivation: In this RQ, we study the percentage of improvement of our framework over the reactive caching mechanisms in the three benchmark scenarios. This can help demonstrate the added benefits in using our proactive mechanism on top of the existing reactive caching mechanisms.

Approach: The percentage of improvement is calculated as the percentage of the increase in the cache hit ratio of one mechanism over another. Table 5 shows the percentage of improvement of our framework (i.e., the RNN-FFNN) over the LRU, the LFU, and the LRFU mechanisms.

Results: Our proactive caching framework (i.e., the RNN-FFNN) improves LRU from 5.4% to 35.4%. The RNN-FFNN improves LFU from 7.3% to 17.3%. In addition, the RNN-FFNN improves LRFU from 5.6% to 34.7%. The highest percentages of improvements are in the sequence scenario that is 29% on average. The sequence scenario represents the occurrences of queries in a well-defined pattern where the work of the prefetching (i.e., RNN) would excel.

In the batch scenario, our framework improves over the traditional mechanisms by 6% on average. The improvements are lower than the improvements of our framework in the sequence scenario since the traditional mechanisms (LRU, LFU, and LRFU) use a greedy approach to cache that fits the nature of the batch scenario. The batch scenario could fit more repetitions of the same query in a shorter time span than the sequence scenario where the greedy caching attains better results.

For the random scenario, the improvement of our framework over the traditional mechanisms is of 7% on average as it is hard to predict the right upcoming queries in a scenario with no clear patterns.

Summary of RQ3

Using our proactive caching framework, the percentage of improvement in the cache hit ratio ranges from 6% to 29%, on average, over the traditional reactive caching mechanisms (i.e., LRU, LFU, and LRFU).

6 THREATS TO VALIDITY

This section addresses the threats to the validity of our approach as follows. First, our approach estimates the cost of queries by feeding the FFNN with the query represented as a word embedding. This is a generalized approach that can be employed in any database system. However, our approach has a drawback as the generated embeddings may differ when the same query is written using aliases or views (i.e., when queries written in a different syntax).

Second, the benchmark is inducted on three generated datasets. We try to simulate the workloads in a system, whether in a sequential, a batch, or a random manner. This simulation grasps the real-life occurrences of queries to a certain degree, but it is not fully accurate. In a perfect scenario, we would have based our study on a recorded history of queries, but we could not find any available rich queries history.

Third, to train the FFNN, we labeled the queries by executing them on the Spark framework. The process is strenuous and requires approximately a week to complete the execution of all the queries. Alternatively, if there exists a system that collects the history of the executed queries with their respective runtime and memory consumption, our work can be replicated easily.

Fourth, the prefetching mechanism that uses the RNN predicts the five upcoming queries. The upcoming queries are either cached using the prefetching mechanism solely or fed to the FFNN to prefetch and cache the cost-effective queries. We chose number five as a proof of concept in our work. However, the number of predicted queries could be tuned based on multiple factors, such as the repetitiveness of the executed queries. Further studies can extend our work by optimizing the number of predicted queries based on the nature of the query execution scenario.

Finally, our mechanisms assume that the history is fixed and no unidentifiable incoming queries would occur in the system. That is valid for benchmarking our prefetching and cost estimation approaches. But in real-life scenarios, the two mechanisms should be monitored as with the increasing number of unidentifiable queries the cost estimation and the prefetching might suffer from more errors. Hence, we can monitor the queries in history periodically and find a threshold where the number of new unidentifiable queries would cause a clear degradation in the performance of our mechanisms (tested by the cache hit ratio) that would drive us to retrain our two neural networks.

7 RELATED WORK

The various work in query cache is divided into two main strategies (1) reactive caching and (2) proactive caching. In this section, we discuss the main query caching techniques. In addition, we describe the background about building and designing the used neural networks (i.e., the FFNN and the RNN).

7.1 Reactive Caching

The work on reactive caching focuses on the eviction and replacement mechanisms that are concerned with identifying the stale queries in the cache to be evicted. The eviction relieves the memory overhead in the cache as it frees space for more beneficial queries to be cached. The eviction mechanisms define protocols based on recency and frequency to detect the stale queries in the cache.

Lange et al. [36] introduce the Least Recently Used (LRU) cache eviction mechanism in their work on the CPUs' registers caching. The algorithm maintains an array of timestamps for each register and the eviction happens by removing the data in the register with the lowest timestamp. That simple idea was then reintegrated as a generic caching algorithm for different fields, such as mobile network and database systems [32, 46, 68].

Matick et al. [44] introduce Least Frequently Used (LFU); another caching eviction mechanism following their work on caching in CPUs' registers. The algorithm stores access counts for each register and evicts the data with the least count. This simple approach proved to be efficient and is considered on par with LRU, as frequency and recency are the two most influential factors for identifying stale cache. This mechanism also spanned to be reused in databases and networks [15, 28].

Lee et al. [37] combine the work of LRU and LFU to create the Least Recently Frequently Used (LRFU) mechanism. The comparison against the two algorithms showed that LRFU outperforms the other two in a spectrum of cache capacities. The combination of the recency and frequency is more effective for larger cache sizes.

In-memory database systems such as Redis [41] or Apache Ignite [3, 77] are widely used nowadays for the fast access and data caching/manipulation. Redis and Ignite are often employed as a middleware layer that helps fast access to hot data to eliminate the necessity of data access from the main traditional relational database [34, 75]. Redis and Ignite employ LRU and LFU specifically as cache eviction mechanisms due to their fast reactive nature of eliminating stale cached queries [1, 2, 63].

Hon et al. [27] employ a dynamic web caching protocol. Web pages are stored as HTML data in a cache placed as an intermediate layer between the web server and the client. The protocol uses a synchronization daemon that invalidates and evicts the cached pages that are out of sync with the server.

Different from the aforementioned reactive caching studies, our approach aims to predict queries that need to be cached and proactively cache the cost-efficient queries.

7.2 Proactive Caching

The work in proactive caching revolves mainly on identifying the beneficial data to cache in the system. The main objective is to cache the content that maximizes the reuse of the cache and avoid thrashing in the system. The work in proactive caching branches from studying the popularity of data [11, 39, 40, 66], the data patterns [11, 50, 76, 78], to the structure of the data [62], which eventually leads to maximizing the cache benefits.

Caching Based on Content Popularity. Luo et al. [40] develop a framework for caching the most popular queries in the system. The system follows statistical measures to consider the set of the most used queries that are prioritized for caching.

Liu et al. [39] implement a deep learning approach to cache popular data in ICN networks. The neural network is adaptive and retrainable depending on the network. The popularity prediction is transformed into a discretized prediction problem where multiple classes are predicted. This approach added with the cache replacement scheme LRU showed a 15% to 40% improvement over the LRU mechanism.

Tanzil et al. [66] introduce a proactive predictive approach to cache popular YouTube content in cellular networks. The approach uses a neural network to predict a 10 class popularity, where the content with a higher class inflicts a more popular content. Their work implements a segmented LRU algorithm for cache eviction.

Caching Based on Data Patterns. Chan et al. [11] describe a generic predictive algorithm based on a recurrence based probabilistic method to cache and prefetch the content with high hit rates at the Wireless Edge. The approach is compared with other predictive methods and shows on average a 12.5% improvement over LRU and LFU and a 5% improvement over other predictive methods.

Zeydan et al. [76] work on caching in big data mobile networks (e.g., 5G networks). The work tackles caching the clients' information at the network edge using contextual information such as the browsing history and the spatio-temporal information.

Zhong et al. [78] introduce a Deep Reinforcement Learning approach that takes into consideration the recency and frequency factors to create an adaptive cache eviction mechanism. Their model was trained on network content and compared with the work of LRU and LFU. The reinforcement learning approach shows an improvement of 15% on average over LRU and LFU work over the varying cache capacities.

Caching Based on Data Structure. Shim et al. [62] consider in their work the partial reuse of queries consisting of a single JOIN operation. Their work aims to identify the expensive subqueries to be cached based on a dynamic query cost function that combines the execution time, the memory requirements, and the frequency of usage of the subquery. Their system also evicts from the cache the stale content using the same cost function when newer queries with higher cost need to be cached.

The aforementioned studies use different machine learning techniques to manage the cache based on the predicted popularity or the expected usage patterns of the cached content. Inspired by the existing work, we use machine learning techniques (i.e., RNN) to predict the upcoming queries. Then, our approach prefetches the cost-efficient queries using the FFNN.

8 CONCLUSION

Query caching is an essential technique to ameliorate the performance of a database system. The work on query caching spans from the decision of queries to cache by estimating the execution time savings or by greedy caching mechanisms that focus on the eviction and replacement policies of the cache.

In this work, we combine the two concepts of cache decision and cache eviction to develop a proactive caching framework. Our framework employs cost estimation and prefetching mechanisms combined with a reactive cache eviction policy. We compare our work to the previous cache replacement policies that are employed in current systems. From our experiments, we observe remarkable improvements of 6% to 29% by using our framework comparing with the existing work in terms of the cache hit ratio.

Our work was benchmarked in three scenarios that represent the different workloads in real life, whether queries occurring in sequence, batch, or random. The recursive neural network (RNN) that serves the prefetching solely or with the combination of the feed-forward neural network (FFNN) that estimates the cost of the queries exceeded in performance all the other caching mechanisms (i.e., LRU, LFU, and LRFU).

Moreover, our work is not specific to a particular database system and can be employed for any database system. Our framework can also be used on an intermediate level between the client and the database system to form an in-memory middleware layer that deals with caching the queries as a whole. In the future, we plan to test our approach on real data captured from real-world usage scenarios.

REFERENCES

[1] 2009. Redis cache eviction policies. https://docs.redislabs.com/latest/rs/administering/database-operations/eviction-policy/. Accessed: 2019-12-01.

[2] 2015. Apache Ignite cache eviction policies. https://apacheignite.readme.io/docs/evictions. Accessed: 2019-12-01.

[3] Sujoy Acharya. 2018. Apache Ignite Quick Start Guide: Distributed data caching and processing made easy. Packt Publishing Ltd.

[4] Sibel Adali, K Selçuk Candan, Yannis Papakonstantinou, and VS Subrahmanian. 1996. Query caching and optimization in distributed mediator systems. In ACM SIGMOD Record, Vol. 25. ACM, 137–146.

[5] Awny K Al-omari, Tom C Reyes, and Robert Wehrmeister. 2010. Hybrid database query caching. US Patent 7,743,053.

[6] Ahmad Alwosheel, Sander van Cranenburgh, and Caspar G Chorus. 2018. Is your dataset big enough? sample size requirements when using artificial neural networks for discrete choice analysis. Journal of choice modelling 28 (2018), 167–182.

[7] Garen Arevian. 2007. Recurrent neural networks for robust real-world text classification. In IEEE/WIC/ACM International Conference on Web Intelligence (WI'07). IEEE, 326–329.

[8] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, and Riccardo Torlone. 1999. Database systems: concepts, languages & architectures. Vol. 1. McGraw-Hill London.

[9] Leif Azzopardi, Mark Girolami, and Keith Van Rijsbergen. 2003. Investigating the relationship between language model perplexity and IR precision-recall measures. (2003).

[10] Matthew Broadbent, Daniel King, Sean Baildon, Nektarios Georgalas, and Nicholas Race. 2015. OpenCache: A software-defined content caching platform. In Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft). IEEE, 1–5.

[11] Chien Aun Chan, Ming Yan, Andre F Gygax, Wenwen Li, Li Li, I Chih-Lin, Jinyao Yan, and Christopher Leckie. 2019. Big Data Driven Predictive Caching at the Wireless Edge. In 2019 IEEE International Conference on Communications Workshops (ICC Workshops). IEEE, 1–6.

[12] Zheng Chang, Lei Lei, Zhenyu Zhou, Shiwen Mao, and Tapani Ristaniemi. 2018. Learn to cache: Machine learning for network edge caching in the big data era. IEEE Wireless Communications 25, 3 (2018), 28–35.

[13] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. 2004. Estimating progress of execution for SQL queries. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data. ACM, 803–814.

[14] Arthur F Cochcroft Jr. 1998. Method and apparatus for detecting thrashing in a cache memory. US Patent 5,752,261.

[15] Mieso K Denko and Jun Tian. 2006. Cooperative caching with adaptive prefetching in mobile ad hoc networks. In 2006 IEEE International Conference on Wireless and Mobile Computing, Networking and Communications. IEEE, 38–44.

[16] KR Dittrich, AM Kotz, and JA Mülle. 1985. A multilevel approach to design database systems and its basic mechanisms. In Proc. IEEE COMPINT, Montreal, Vol. 183.

[17] Klaus R Dittrich, Willi Gotthard, and Peter C Lockemann. 1987. DAMOKLESâĂŤa database system for software engineering environments. In Advanced programming environments. Springer, 353–371.

[18] Gunhan Dundar and Kenneth Rose. 1995. The effects of quantization on multilayer neural networks. IEEE Transactions on Neural Networks 6, 6 (1995), 1446–1451.

[19] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. 1999. Learning to forget: Continual prediction with LSTM. (1999).

[20] Parke Godfrey and Jarek Gryz. 1997. Semantic Query Caching for Hetereogeneous Databases.. In KRDB. 6–1.

[21] Yoav Goldberg and Omer Levy. 2014. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. arXiv preprint arXiv:1402.3722 (2014).

[22] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. 6.2. 2.3 softmax units for multinoulli output distributions. In Deep Learning. MIT Press, 180–184.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. Deep Learning. MIT Press. http://www.deeplearningbook.org.

[24] David J Hand and Robert J Till. 2001. A simple generalisation of the area under the ROC curve for multiple class classification problems. Machine learning 45, 2 (2001), 171–186.

[25] James A Hanley and Barbara J McNeil. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology 143, 1 (1982), 29–36.

[26] Olaf Hartig and Ralf Heese. 2007. The SPARQL query graph model for query optimization. In European Semantic Web Conference. Springer, 564–578.

[27] Lenny K Hon, Leon Kuperman, Louis S Mau, and Alexander Mohelsky. 2001. Caching dynamic web pages. US Patent 6,185,608.

[28] Jian hua Ran, Na Lv, Ding Zhang, Yuan yuan Ma, and Zhen yong Xie. 2013. On performance of cache policies in named data networking. In 2013 International Conference on Advanced Computer Science and Electronics Information (ICACSEI 2013). Atlantis Press.

[29] Jin Huang and Charles X Ling. 2005. Using AUC and accuracy in evaluating learning algorithms. IEEE Transactions on knowledge and Data Engineering 17, 3 (2005), 299–310.

[30] Frederick Jelinek, Bernard Merialdo, Salim Roukos, and Martin Strauss. 1991. A dynamic language model for speech recognition. In Speech and Natural Language: Proceedings of a Workshop Held at Pacific Grove, California, February 19-22, 1991.

[31] Navin Kabra and David J DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In ACM SIGMOD Record, Vol. 27. ACM, 106–117.

[32] Harshad N Kamat and David J Clarke. 2012. Email server with enhanced least recently used (LRU) cache. US Patent 8,307,036.

[33] Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva. 2012. Bicliques for preimages: attacks on Skein-512 and the SHA-2 family. In International Workshop on Fast Software Encryption. Springer, 244–263.

[34] Doyoung Kim, Won Gi Choi, Hanseung Sung, and Sanghyun Park. 2019. A scalable and persistent key-value store using non-volatile memory. In Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. ACM, 464–467.

[35] Gary King and Langche Zeng. 2001. Logistic regression in rare events data. Political analysis 9, 2 (2001), 137–163.

[36] Ronald E Lange and Richard J Fisher. 1982. Cache memory utilizing selective clearing and least recently used updating. US Patent 4,322,795.

[37] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. IEEE transactions on Computers 12 (2001), 1352–1361.

[38] Ken CK Lee, Hong Va Leong, and Antonio Si. 1999. Semantic query caching in a mobile environment. ACM SIGMOBILE Mobile Computing and Communications Review 3, 2 (1999), 28–36.

[39] Wai-Xi Liu, Jie Zhang, Zhong-Wei Liang, Ling-Xi Peng, and Jun Cai. 2017. Content popularity prediction and caching for ICN: A deep learning approach with SDN. IEEE access 6 (2017), 5075–5089.

[40] Qiong Luo, Jeffrey F Naughton, Rajasekar Krishnamurthy, Pei Cao, and Yunrui Li. 2000. Active query caching for database web servers. In International Workshop on the World Wide Web and Databases. Springer, 92–104.

[41] Tiago Macedo and Fred Oliveira. 2011. Redis Cookbook: Practical Techniques for Fast Data Manipulation. " O'Reilly Media, Inc.".

[42] Bhushan Mandhani and Dan Suciu. 2005. Query caching and view selection for XML databases. In Proceedings of the 31st international conference on Very large data bases. VLDB Endowment, 469–480.

[43] Michael Martin, Jörg Unbehauen, and Sören Auer. 2010. Improving the performance of semantic web applications with SPARQL query caching. In Extended Semantic Web Conference. Springer, 304–318.

[44] Richard Edward Matick, Jaime H Moreno, and Malcolm Scott Ware. 2006. Cache with selective least frequently used or most frequently used cache line replacement. US Patent 7,133,971.

[45] Geoffrey J McLachlan, Kim-Anh Do, and Christophe Ambroise. 2005. Analyzing microarray gene expression data. Vol. 422. John Wiley & Sons.

[46] Nimrod Megiddo and Dharmendra S Modha. 2004. Outperforming LRU with an adaptive replacement cache algorithm. Computer 37, 4 (2004), 58–65.

[47] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In Eleventh annual conference of the international speech communication association.

[48] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In Proceedings of the 27th international conference on machine learning (ICML-10). 807–814.

[49] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The making of TPC-DS. In Proceedings of the 32nd international conference on Very large data bases. VLDB Endowment, 1049–1058.

[50] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. 2018. Deepcache: A deep learning based framework for content caching. In Proceedings of the 2018 Workshop on Network Meets AI & ML. ACM, 48–53.

[51] Sarang Narkhede. 2018. Understanding AUC-ROC Curve. Towards Data Science 26 (2018).

[52] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In 2011 IEEE 27th International Conference on Data Engineering. IEEE, 984–994.

[53] Hieu V Nguyen and Li Bai. 2010. Cosine similarity metric learning for face verification. In Asian conference on computer vision. Springer, 709–720.

[54] Takayuki Osogami. 2010. A fluid limit for a cache algorithm with general request processes. Advances in Applied Probability 42, 3 (2010), 816–833.

[55] M Tamer Özsu and Patrick Valduriez. 2011. Principles of distributed database systems. Springer Science & Business Media.

[56] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why you should run TPC-DS: a workload analysis. In Proceedings of the 33rd international conference on Very large data bases. VLDB Endowment, 1138–1149.

[57] Meikel Poess, Tilmann Rabl, and Hans-Arno Jacobsen. 2017. Analysis of TPC-DS: the first standard benchmark for SQL-based big data systems. In Proceedings of the 2017 Symposium on Cloud Computing. ACM, 573–585.

[58] D Rachmawati, JT Tarigan, and ABC Ginting. 2018. A comparative study of Message Digest 5 (MD5) and SHA256 algorithm. In Journal of Physics: Conference Series, Vol. 978. IOP Publishing, 012116.

[59] Qun Ren, Margaret H Dunham, and Vijay Kumar. 2003. Semantic caching and query processing. IEEE transactions on knowledge and data engineering 15, 1 (2003), 192–210.

[60] Xin Rong. 2014. word2vec parameter learning explained. arXiv preprint arXiv:1411.2738 (2014).

[61] Muhammad Zubair Shafiq, Alex X Liu, and Amir R Khakpour. 2014. Revisiting caching in content delivery networks. In ACM SIGMETRICS Performance Evaluation Review, Vol. 42. ACM, 567–568.

[62] Junho Shim, Peter Scheuermann, and Radek Vingralek. 1999. Dynamic caching of query results for decision support systems. In Proceedings. Eleventh International Conference on Scientific and Statistical Database Management. IEEE, 254–263.

[63] Cristiana-Stefania Stan, Adrian-Eduard Pandelica, Vlad-Andrei Zamfir, Roxana-Gabriela Stan, and Catalin Negru. 2019. Apache Spark and Apache Ignite Performance Analysis. In 2019 22nd International Conference on Control Systems and Computer Science (CSCS). IEEE, 726–733.

[64] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. 2012. LSTM neural networks for language modeling. In Thirteenth annual conference of the international speech communication association.

[65] Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. 1997. Introduction to multi-layer feed-forward neural networks. Chemometrics and intelligent laboratory systems 39, 1 (1997), 43–62.

[66] SM Shahrear Tanzil, William Hoiles, and Vikram Krishnamurthy. 2017. Adaptive scheme for caching YouTube content in a cellular network: Machine learning approach. Ieee Access 5 (2017), 5870–5881.

[67] Michael J Turmon and Terrence L Fine. 1995. Sample size requirements for feedforward neural networks. In Advances in Neural Information Processing Systems. 327–334.

[68] AI Vakali. 2000. LRU-based algorithms for Web cache replacement. In International conference on electronic commerce and web technologies. Springer, 409–418.

[69] Stijn Van Dongen and Anton J Enright. 2012. Metric distances derived from cosine similarity and Pearson and Spearman correlations. arXiv preprint arXiv:1208.3145 (2012).

[70] Xingxing Wang, LiMin Wang, and Yu Qiao. 2012. A comparative study of encoding, pooling and normalization methods for action recognition. In Asian Conference on Computer Vision. Springer, 572–585.

[71] William E Woods and Arthur Peters. 1982. Hit/miss logic for a cache memory. US Patent 4,363,095.

[72] Tian Xia, Dacheng Tao, Tao Mei, and Yongdong Zhang. 2010. Multiview spectral embedding. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 40, 6 (2010), 1438–1446.

[73] Hongliang Yan, Yukang Ding, Peihua Li, Qilong Wang, Yong Xu, and Wangmeng Zuo. 2017. Mind the class weight bias: Weighted maximum mean discrepancy for unsupervised domain adaptation. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2272–2281.

[74] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. HotCloud 10, 10-10 (2010), 95.

[75] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. 2017. Caching at the web scale. Proceedings of the VLDB Endowment 10, 12 (2017), 2002–2005.

[76] Engin Zeydan, Ejder Bastug, Mehdi Bennis, Manhal Abdel Kader, Ilyas Alper Karatepe, Ahmet Salih Er, and Mérouane Debbah. 2016. Big data caching for networking: Moving from cloud to edge. IEEE Communications Magazine 54, 9 (2016), 36–42.

[77] Michael Zheludkov, Timur Isachenko, et al. 2017. High Performance in-memory computing with Apache Ignite. Lulu. com.

[78] Chen Zhong, M Cenk Gursoy, and Senem Velipasalar. 2018. A deep reinforcement learning-based framework for content caching. In 2018 52nd Annual Conference on Information Sciences and Systems (CISS). IEEE, 1–6.