5-6-2022

Indexer++: Workload-Aware Online Index Tuning With Transformers and Reinforcement Learning

Vishal Sharma
Utah State University, vishal.sharma@usu.edu

Curtis Dyreson
Utah State University, curtis.dyreson@usu.edu

Recommended Citation

Vishal Sharma and Curtis Dyreson. 2022. Indexer++: Workload-Aware Online Index Tuning with Transformers and Reinforcement Learning. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22), April 25–29, 2022, Virtual Event,* . ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3477314.3507691

Indexer++: Workload-Aware Online Index Tuning with Transformers and Reinforcement Learning

Vishal Sharma
Utah State University
Department of Computer Science
vishal.sharma@usu.edu

Curtis Dyreson
Utah State University
Department of Computer Science
curtis.dyreson@usu.edu

ABSTRACT

With the increasing workload complexity in modern databases, the manual process of index selection is a challenging task. There is a growing need for a database with an ability to learn and adapt to evolving workloads. This paper proposes **Indexer++**, an autonomous, workload-aware, online index tuner. Unlike existing approaches, **Indexer++** imposes *low overhead* on the DBMS, is *responsive to changes* in query workloads and *swiftly selects* indexes. Our approach uses a combination of text analytic techniques and reinforcement learning. **Indexer++** consist of two phases: *Phase (i)* learns workload trends using a novel trend detection technique based on a pre-trained transformer model. *Phase (ii)* performs *online*, *i.e.,* continuous or while the DBMS is processing workloads, index selection using a novel online deep reinforcement learning technique using our proposed priority experience sweeping. This paper provides an experimental evaluation of **Indexer++** in multiple scenarios using benchmark (TPC-H) and real-world datasets (IMDB). In our experiments, **Indexer++** effectively identifies changes in workload trends and selects the set of optimal indexes.

CCS CONCEPTS

• **Information systems** → **Autonomous database administration**; **Data management systems**; **Database design and models**; *Database management system engines.*

KEYWORDS

Workload Trend Detection, Online Index Selection, Pre-trained Transformers, Reinforcement Learning

ACM Reference Format:
Vishal Sharma and Curtis Dyreson. 2022. Indexer++: Workload-Aware Online Index Tuning with Transformers and Reinforcement Learning. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22), April 25–29, 2022, Virtual Event, .* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3477314.3507691

1 INTRODUCTION

Choosing an optimal set of configuration parameters is critical to the performance of a DBMS. Among all configuration parameters, the selection of indexes is arguably the most important and challenging [9]. The *index selection* problem is to select a set of indexes that minimizes the cost of query evaluation. Although selecting the right set of indexes is critical to achieving good performance [11], the *index selection* problem is *NP-hard* [34]. With the growing number of database configuration parameters and increasing complexity of query workloads, there is an upsurge of interest in *self-managing DBMSs*, that is, in DBMSs that can automate their configuration and adapt that configuration as needed in response to changes in their environment. Previous research in self-managing databases using Machine Learning (ML), has shown that automated index selection can outperform manual index selection [46].

In a manual index selection process, a Database Administrator (DBA) analyzes a representative workload selected from the workload history to perform experiments and create indexes. A manual index selection process has several limitations. *First*, identifying a representative workload from thousands of queries is a challenging task, and random sampling may miss edge-case scenarios. *Second*, the selected set of indexes does not generalize for complete historical workloads. *Third*, the process does not address current or future workload trends. *Finally*, the process has a latency concerns, a DBA will usually modify the index set when individuals complain about a drop in performance. A manual index selection is a static and cumbersome process. In contrast, the workloads can change frequently and the optimal set of indexes is a moving target [45]. The limitations of manual index selection can be overcome with an index selection method that is a *continuous/online* decision making process. To motivate this problem we share an example:

Example 1.1. ***App-X*** *is a framework for trading on the stock exchange, where users can check a stock ticker (abbreviation of stock and current price), conduct chart analysis, and purchase and sell stocks. Over a trading day,* ***App-X*** *could see a massive shift in database query workloads. The value-based stock queries could be frequent at the opening bell, but technology-based could gain traction near the closing bell.* ***App-X*** *needs to change its indexes over the day to process queries efficiently and respond quickly to end-users, but can only do so if it can detect and respond to (patterns of) workload trends.*

Previously automated index selection research has primarily focused on *offline* methods. An offline indexer uses the current state of the database and a representative query workload to determine an optimal set of indexes. Offline indexers use heuristics [10, 15, 31, 44], machine learning [1, 12, 21, 22], or reinforcement learning [3, 19, 39, 41] to recommend a set of indexes. The set of indexes generated by an offline indexer can become *stale* if the workload changes. A stale set can be refreshed by re-running the offline indexer to recreate the indexes. But the cost of offline indexers can be high (on the order of hours of computation time) since an algorithm may re-run thousands of queries and re-build many indexes. An offline indexer's periodic runs degrade DBMS throughput and response time. In contrast to offline index selection, the goal of *online* index selection is to *keep the set of indexes fresh.* An online indexer initially generates a set of indexes and then progressively adapts the set in response to workload or database changes.

An online indexer faces several challenges not shared by their offline counterparts.

(C1) **Noise Resilience**: An online indexer must react to shifting trends while avoiding short-lived once that might harm DBMS performance by updating the indexes frequently. A challenge here is identifying how to *recognize a trend*, *i.e.,* how many and what kind of queries suggest a distinct pattern.

(C2) **Overhead**: The tuning process must be able to run concurrently with other DBMS operations without interfering with processing. Low overhead ensures that an online indexer can (in effect) run in parallel and respond to changing trends.

(C3) **Trend Detection**: An inability to detect trends quickly may result in delayed index selection. A significant delay in trend detection, in turn, reduces the utility of new indexes.

(C4) **Responsiveness**: The index tuning response should be quick, on the order of minutes. If index selection takes many hours or longer, the tuner may miss trends. An online indexer should ideally be able to respond in real-time.

Previous research in online index selection [7, 28, 33, 35, 36, 40] focused on analyzing a query workload window to recommend a set of indexes. These approaches do not adapt to changing workload patterns. They also suffer from high cost (computation and responsiveness) and have a long delay between trend detection and index selection. The high cost and delay reduces the overall utility of having an online indexer.

This paper presents **Indexer++**, a workload-aware, online index tuner and makes the following contributions.

- We propose and evaluate our novel *workload embedding* and *trend detection* technique using a pre-trained transformer based model, dimentionality reduction and clustering algorithm.
- We utilize deep reinforcement learning techniques to learn the interactions between indexes [23] that helps improve performance [37]. We design our reward function with a constraint on the total storage cost of the selected set of indexes.
- We propose and evaluate a novel *Priority Experience Sweeping* technique for online learning that extends the Deep Q-Network (DQN) algorithm. For a command-based database interaction, we design and build an OpenGym environment.
- We present an extensive evaluation of our framework on two datasets (IMDB and TPC-H). The experiments demonstrate that **Indexer++** can identify workload trends and select indexes effectively while addressing challenges (C1)-(C4) listed above.

This paper is organized as follows. § 2 describes related work and § 3 formulates the problem. In § 4 we describe **Indexer++**, introduce a workload embedding technique, and use the technique to detect change in workload patterns. The section also describes our algorithm for online index selection. § 5 presents experiments, and finally § 6 concludes the paper.

2 RELATED WORK

Workload Embedding: In the realm of Natural Language Processing (NLP), the technique of generating a vector representation of a word (*word embedding*) has received tremendous attention. Hinton *et al.* [16] pioneered the notion of vector representation. Mikolov *et al.* [29] proposed *word2vec* a model that can learn latent relationship between words from a text corpus. This advancement led to addressing several challenging problems in NLP such as semantic analogy [26], sentiment analysis [38], and document classification [27]. The *word2vec* model was later expanded to sentences and documents [24]. To improve the vector representation other deep learning techniques were also employed such as CNN, RNN and LSTM. The quality of word embeddings were significantly improved by the introduction of Deep Bidirectional Transformers, examples BERT, Transformer-XL, and XML.

Several previous attempts were made to embed a SQL workload. Bordawekar *et al.* [5, 6] utilize pre-trained *word2vec* models for query embedding. Jain *et al.* [18] performs error prediction and workload summarization using vector representation learned by training an LSTM-based autoencoder. Bandyopadhyay *et al.* [2] propose a database column embedding using Bi-LSTM for drug-drug interaction prediction. Cappuzzo *et al.* [8] propose data integration using graph-based representation. Günther *et al.* [14] proposes enriching database queries using pre-trained *word2vec* model. The *word2vec* embedding is a pioneer in text vector representation. It does, however, generate static lower-dimensional and non-contextual vectors. **Indexer++** employs a pre-trained transformer based model (BERT, Transformer-XL) for query embedding, which generates dense and context-dependent vectors that aids in learning the semantic and syntactic relationship between queries.

Online index tuning: The study of the problem of *online index selection* dates back to the 1970s. Hammer *et al.* [15] used heuristics for index selection on a single table. Frank *et al.* [13] proposed an online tuning method for single index selection using workload statistics for change detection and using heuristics for index selection. Kołaczkowski *et al.* [20] uses an evolution technique for selecting indexes in a query execution plan. A solution using heuristics may fail in scalability (Challenge C4). Bruno *et al.* [7] design an ad-hoc approach for index selection. Their approach is fast and scalable, but it lacks noise resilience (Challenge C1). For a small DBMS application, such an approach may work, but it can worsen the performance of a large system. Schnaitter *et al.* [36] proposed a very effective online indexing solution that monitors the incoming queries and minimizes the overhead cost. Their index selection uses heuristics. However, it is slow for real-time online index selection (C4) and susceptible to noise (C1). Sadri *et al.* [35] proposed an index tuning algorithm using deep reinforcement learning. As described by the authors, their approach takes a long time for index selection (C4). None of the previous approaches overcomes all of the challenges. They mostly fail to address noise resilience (C1) or scalability (C4). Furthermore, most previous approaches do not take index disk storage cost into account as an optimization parameter. The storage cost is very crucial for modern databases that place a high priority on minimizing the cost of data storage. Unnecessary indexes can worsen the DBMS performance. Our proposed approach overcomes all of the challenges (C1)-(C4).

3 PROBLEM FORMULATION

The *index selection problem* is to select a set of indexes that minimizes the time to evaluate a workload within a given storage size constraint; it is a *constrained combinatorial optimization* problem. The space constraint is very crucial because the selected set of indexes should be *parsimonious*. In addition to straining disk space and increasing database overhead, creating more indexes than required also slow down data modifications.

A workload, W, is a sequence of SQL queries, $[Q_1, Q_2, \ldots, Q_m]$. An index configuration, I, is a set of indexes. The cost of the evaluation of a workload on database D is the sum of the cost of the evaluation of each individual query in the workload given by $Cost(Q_j, I, D)$ and can be described as follows.

$$Cost(W, I, D) = \sum_{j=1}^{m} Cost(Q_j, I, D)$$

Please note, that the workload cost gives the same weight to each query in the workload, though weighted costs could be trivially included by replicating individual queries. The index selection problem is to find a set of indexes $I_{optimal}$ that minimizes the total cost of a workload, $Cost(W, I, D)$, and has a storage cost of at most C.

$$I_{optimal} = \min_{S(I) \leq C} Cost(W, I, D)$$

In this equation $S(I)$ is the total space cost of I. The *online index selection* problem can be formulated as follows. Let a workload stream, W^*, be a sequence of workloads, $[W_1, W_2, \ldots, W_k]$. Let $Diff(W_1, W_2, \ldots, W_k)$ be a metric that quantifies the difference between workloads in the stream. Then online index selection can be defined as:

$$reindex_{config}(I_{optimal}, D) = \min_{Diff(W^*) \geq \lambda} Cost(W^*, I^*, D)$$

where, λ is a measure of the sensitivity to re-configure the indexes based on the workload stream difference (λ is described in §4.2), I^* is the set of indexes that changes with a change in workload trends, and $I_{optimal}$ is the final optimal set of indexes.

4 INDEXER++

A DBMS should be *workload-aware* and *self-tune* indexes in response to changing workloads [32], however developing such a DBMS presents various challenges.

- There are various types of workloads, such as analytical, data manipulation, and data definition. They have different functionality and execution cost. As a result, a workload-aware DBMS should be able to distinguish between them while also learning and adapting to *heterogeneous* workloads.
- In recent years the *scale and volume* of data has grown substantially. A workload can consist of thousands of SQL queries. Therefore, the workload trend can change in a relatively short amount of time. Such rapid change in workload trends requires an *online* process of representing and understanding queries.
- *Workload representation* is challenging. A workload-aware DBMS needs to extract workload patterns in real-time. Understanding the syntax and semantics of a workload requires it to be represented in a standard form. It requires a pattern extraction approach that must be able to learn the relationship among workloads and withstand the workload volume.

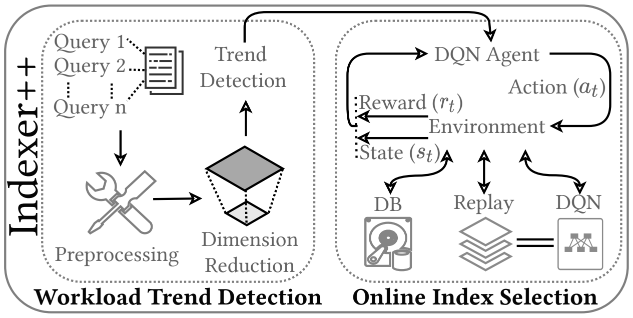

Figure 1: Indexer++ framework with two phases (i) Workload Trend Detection (left) (ii) Online Index Selection (right)

Indexer++ overcomes these challenges in two phases. The first phase represents a workload using embedding techniques and identifies workload trends using *K-medoids* clustering. The second phase selects the set of indexes using online deep reinforcement learning. The first phase communicates with the second, but they function independently as shown in Figure 1 which depicts our framework.

4.1 Workload Embedding

Queries in a workload may vary depending on the user and the application. From among the many workloads some workloads are similar. **Indexer++** has to recognize similarity in workloads to detect trends and optimally select indexes. *Representation learning* has been shown to be effective in learning the syntax and semantics of text [4]. Representation learning uses a d-dimension vector for each word from a corpus of text [6]. The vector encodes the meaning of a word and captures its latent features in such a way that relationships between words can be expressed using arithmetic operations *e.g., king to man is what to woman? Answer: queen* [26].

Previous work on workload representation has proposed two ways to represent a workload **(i)** using statistics generated by executing the workload [1], and **(ii)** training a deep learning model on a large corpus of queries and performing inference [18]. Unfortunately, both of these approaches have drawbacks. The former technique may be impractical in an online tuning process since it executes the workload, which may escalate database load and interrupt existing operations, whereas the latter needs run-time training, which is computationally intensive.

Indexer++ uses a third way: *universal embedding*. Universal embedding is a form of *transfer learning* in which a model is pre-trained on an extensive corpus of text. Universal embedding offers several advantages. *First*, the inference time[1] has no slack, which makes it a feasible solution for online tuning. *Second*, universal embedding has no workload evaluation-time overhead. It does not collect any information from the database and does not interrupt running processes or applications. Experimentally, we found universal embedding to be able to represent a workload effectively and identify heterogeneous workloads (experimental results are given in § 5). One drawback of a pre-trained model is that it may have a minor bias [43]. However, we prioritize speed for workload embedding in **Indexer++**, which is an advantage of a pre-trained model, so a minor bias is less relevant.

[1]Time required to convert a word to a vector representation using a pre-trained model

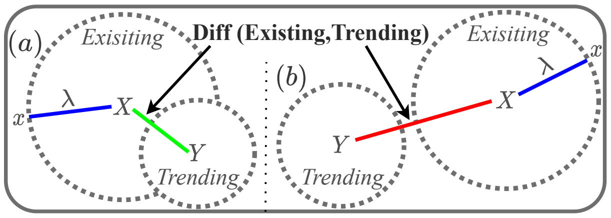

Figure 2: Visual description of trigger parameters λ, centroids X, Y and $Diff(Existing, Trending)$

4.2 Workload Trend Detection

In the previous section, we discussed about transforming a workload to a vector representation. There are several applications of such embedding, for example, workload summarization, cost estimation, and query classification. In this section, we will introduce another application, *Workload Trend Detection*.

Representation: Indexer++ performs dimensionality reduction on an embedded workload using *t-distributed stochastic neighbor embedding* (t-SNE), a non-linear dimensionality reduction technique. The purpose is to reduce the embedding dimension for further analysis (clustering), noise reduction, and visualization. The feature space of t-SNE is found by searching for a low-dimensional projection of data equivalent to the actual data using a stochastic neighbor embedding. The neighbor embedding is constructed in two steps: (i) determine a probability distribution such that similar objects are assigned a similar probability, and (ii) construct a similar distribution in a lower dimension while minimizing the entropy using the KL divergence of the two distributions.

In our workloads, we assume there are two sets of queries: *existing*, which is the historical workload, and *trending*, which is the forthcoming/current workload. In the reduced dataset, **Indexer++** performs *K-medoids* clustering on the reduced dataset (after using t-SNE) to locate the span of both workloads in vector space. The dimensionality reduction is a standard preprocessing technique for clustering algorithms to decrease noise in the data and to improve quality of clusters [17]. **Indexer++** uses the *K-medoids* clustering algorithm due to the interpretability of the selected centroids. *K-medoids* finds a centroid that is a sample from the dataset, while other techniques like *K-means* may output a non-data point as a centroid. The centroid selected by *K-medoids* can be traced back to the actual query, which helps in interpretation and visualization.

Trend Detection: Indexer++ then determines if the *trending* workload is sufficiently different than the *existing* workload. Workloads are represented as a cluster. The radius of the *existing* workload cluster is the euclidean distance between the furthermost point (x) and the cluster centroid (X), which we designate as λ. The value of λ quantifies the span of the *existing* workload in vector space. We also compute the euclidean distance between the two centroids *existing* and *trending*. When this difference is not greater than λ, the *trending* workload is deemed to be similar to the *existing* workload since the centroid of the *trending* workload lies within the span of the *existing* workload. In this case, the *trending* workload is combined with the *existing* workload and the centroid of the *existing* workload and λ are recomputed. Otherwise, the *trending* workload is different and **Indexer++** triggers online index selection before merging the *trending* workload with the *existing*.

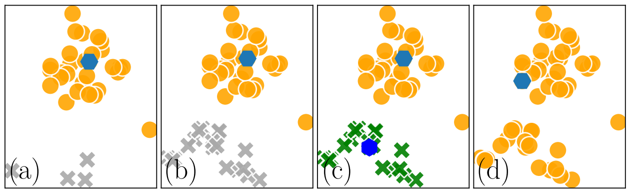

Figure 3: Workload Trend Detection on two workloads (Workload 1, 2) of TPC-H Random query dataset where blue hexagon indicates the centroids of clusters

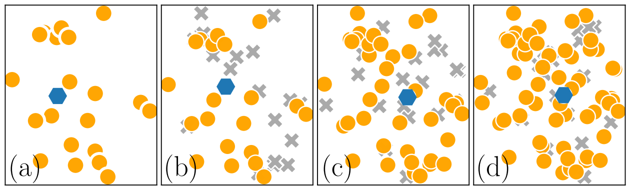

Figure 4: Workload Trend Detection on TPC-H Random query dataset (Workload 1, 3). The *trending* queries are similar to *existing* workload. The blue hexagon indicates the centroid of the cluster. The *existing* workload is displayed in orange circle and new queries in gray cross

Visual Description: We describe our notion of *Workload Trend Detection* using a visual representation in Figure 2. In the figure, X and Y represent the cluster centroids of the *existing* and *trending* workloads, respectively. λ is the distance between the centroid (X) and furthest point (x). In Fig 2 (a), the euclidean distance between X and Y is less than λ, thus, the index set will not be recomputing and workloads will be merged since they are relatively similar. In scenario (b), the distance is greater, and the index selection is triggered before merging the workloads. The *Workload Trend Detection* is triggered after 32 queries (batch size for the online DQN algorithm presented in § 4.3) in the *trending* workload.

Validation on TPC-H: To evaluate our workload trend detection approach, we experiment on the `TPC-H Random` dataset. We split `TPC-H Random` queries into three workloads, *Workload 1*, *Workload 2*, and *Workload 3* with 25 queries each. Each workload represents a set of queries and *Workload 1* and *Workload 3* are similar workloads, whereas *Workload 1* and *Workload 2* are dissimilar. *Workload 1* is assumed as the *existing* workload and *Workload 2* and *Workload 3* are introduced later as *trending* workloads to create two different scenarios as shown in Figures 3 and 4. An inference on *Workload 1* SQL queries is performed using pre-trained RoBERTa (§5 evaluates superiority of RoBERTa). The output of the inference returns a *high*-dimensional vector. This *high*-dimensional vector is reduced using t-SNE. A visualization of *Workload 1* (orange circles) and a few queries from *Workload 2* (gray cross) is shown in Figure 3 (a). The number of dots may not be equal to the number of queries due to overlapping. The *K-medoids* clustering on the reduced *Workload 1* dataset returns the centroid as shown in a blue hexagon. We introduce a few more queries from *Workload 2* shown in Figure 3 (b) and In Figure 3 (c), clustering is performed (in green) and the *trending* workload (*Workload 2*) centroid is computed. Index selection is triggered because the difference between the centroids is greater than λ. The workloads are then merged and a new centroid is computed as shown in the final panel in Figure 3 (d). In a similar way, *Workload 3* queries are introduced to *Workload 1* as shown in Figure 4. In this scenario, as the centroids are recomputed, the workloads are similar and merged without new index selection.

Constraint: The final step in trend detection is to remove *infrequent* queries from the *existing* workload. **Indexer++** sets a maximum threshold for the size of the *existing* workload (value varies w.r.t. system configuration). When this threshold is exceeded, infrequent queries are removed. The centroid and λ are recomputed. The intuition is that frequent queries are more likely to be re-evaluated in the future so should be retained in the *existing* workload over one-off queries. If the query frequency is the same, older queries are removed from the *existing* workload in favor of younger queries.

4.3 Online Index Selection

A query optimizer may use a combination of indexes rather than a single index to optimize a query. The choice of adding an index to the set can be modeled as a *Markovian Decision Processes* (MDP) where the selection of an index impacts future choices. *Reinforcement Learning* (RL) is a popular technique to optimize MDPs and has been used to discover directed acyclic graphs [47], and vehicle routes [30]. In contrast to traditional machine learning, where training requires a labeled dataset, an RL agent interacts with an environment and learns from *experiences.* At a given *time (t)* and *state (s_t)* an *agent* performs an *action (a)* following a *policy* and proceeds to the next *state (s_{t+1}).* It receives a scalar *reward* during the state transition. To model online index selection as an MDP, **Indexer++** extracts relevant information from a database to define a *state* and has a pipeline where at a given *time, t* there is a *state$_t$* to *action$_t$* mapping (*state* and *action* are deterministic). The goal of an MDP is to reach the final *state* maximizing cumulative *rewards* and identifying a *policy* that is, an optimal *state-action* mapping.

4.3.1 *Online DQN Algorithm.*

A Deep Q-Networks (DQN) is a popular offline RL based algorithm, where policy $\pi(s, a)$ and values $q(s, a)$ are represented using multi-layer neural networks (NN). The neural networks apply high-dimensional input data representation, generalizing similar experiences and unseen states. The hyper-parameters of the neural networks are trained by gradient descent minimizing a loss function. We use the mean squared error, given below, as the loss function.

$$Loss(\theta) = \mathbb{E}_{\pi}\left[\frac{1}{2}\,(\overbrace{Bellman}^{Target} - \overbrace{Q(s,a;\theta)}^{Estimate})^2\right]$$

where, θ is a parameter for a nonlinear function, in our approach a NN. We expand the equation using the Bellman Optimality [42]:

$$Loss(\theta) = \mathbb{E}_{\pi}\left[\frac{1}{2}\left((\overbrace{R(s,a)}^{Reward} + \overbrace{\beta \max_{a' \in A} Q(s',a';\theta))}^{Future\ Reward} - \overbrace{Q(s,a;\theta)}^{Estimate}\right)^2\right]$$

where, β is the discount rate. In the above equation, we approach to learn the weights of $Q(s, a; \theta)$ using stochastic gradient descent optimization. To learn the parameters a gradient update w.r.t $Q(s, a; \theta)$ can be performed as shown below:

$$Q(s,a;\theta) = Q(s,a;\theta) - \alpha \frac{\partial}{\partial Q(s,a;\theta)} Loss(\theta)$$

replacing $Loss(\theta)$ and taking a partial derivative yields:

$$Q(s,a;\theta) = Q(s,a;\theta) + \alpha\left(\underbrace{\overbrace{R(s,a) + \beta \max_{a' \in A} Q(s',a';\theta)}^{TD\ Target} - \overbrace{Q(s,a;\theta)}^{Estimate}}_{Temporal\ Difference\ (TD)\ Error}\right)$$

rearranging the above equation gives:

$$Q(s,a;\theta) = (1-\alpha)\, Q(s,a;\theta) + \alpha\left(R(s,a) + \beta \max_{a' \in A} Q(s',a';\theta)\right)$$

We use above equation to generalize the approximation of the Q-value function. A neural network training assumes that input data are independent and sampled from similar distributions. A neural network will overfit/underfit if such assumptions are not satisfied. For Reinforcement Learning, we can observe that the target Q value depends on itself, making training on a neural network difficult since it would chase a non-stationary target. To solve this problem, we implement a target network θ', which stays stationary for a certain period and later synchronizes with θ. The above equation can be re-written using target network θ' as follows:

$$Q(s,a;\theta) = (1-\alpha)Q(s,a;\theta) + \alpha \underbrace{\left(R(s,a) + \beta \max_{a' \in A} Q(s',a';\theta')\right)}_{TD\ Target}$$

A common problem occurs during the learning process when an agent tends to forget experiences after a few epochs. To solve this problem, we store a buffer called the *Experience Replay* to sample from historical experiences. The buffer also breaks the temporal dependency otherwise found in regular updates. However, due to uniform data sampling *Experience Replay* can lead to slow convergence. This is because a previous experience with a large estimated error may not be in the sampled data. To solve this problem, we use ***Priority Experience Replay (PER)***, where instead of sampling uniformly, samples are given weights. We use the *Temporal Difference (TD) Error* as weights to prioritize experiences.

4.3.2 *DBMS representation:*

A *state* of a DQN agent represents an environment. We use the existing set of indexes in a database to represent its current *state.* The index set is represented in a one-dimensional vector where a binary value indicates the presence (or absence) of an index. The *action* space represents all possible index configurations. An agent learns the optimal policy by maximizing the *reward function.* We define our reward function as follows:

$$r_t = max\left(\frac{Cost(W, I_{t_0}, D)}{Cost(W, I_{t_{n-1}}, D) - Cost(W, I_{t_n}, D)} - 1, 0\right) + r_{size} \quad (1)$$

$$\text{where,}\quad r_{size} = \begin{pmatrix} 1 & \text{total index size} < max\_allowed\_size \\ -1 & \text{otherwise} \end{pmatrix}$$

where, the numerator is the workload cost with no index and the denominator is the workload cost with the selected index for the next step. We also introduce the reward for the disk size constraint, where the total size of the selected indexes has an upper bound of *max_allowed_size*, *e.g.,* 100MB. Our reward function is generic and can be modified for other constraints.

4.3.3 *Priority Experience Sweeping (PES):.*

By nature, the DQN algorithm is an online learning process. It interacts with an environment and, by maximizing rewards, learns the optimal behavior.

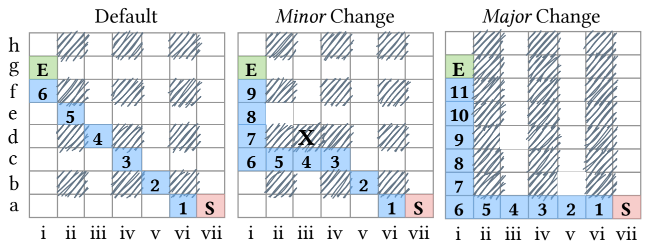

Figure 5: A robot navigation in a warehouse from a start point (in red) to end point (in green), blue boxes show an optimal route and gray boxes are obstacles

Similarly, the index tuning process learns optimal behavior (selecting indexes) by reducing the query cost. However, when a workload changes, the optimal policy of the environment will change, which requires retraining of the DQN agent. The retraining process is time consuming, and if it takes longer time, that will reduce the utility of having online index tuning. To avoid retraining and to handle this scenario, we introduce *Priority Experience Sweeping*. The motivation for *Priority Experience Sweeping* is that the DQN agent has already learned an optimal policy for the environment. The state and action of the environment do not change. Instead, there is a slight change in the behavior. Rather than retraining from scratch *how can we utilize previously learned behavior?*

Example 4.1. *To give the context, we describe Priority Experience Sweeping using an example of robot navigation in a warehouse. Given a warehouse where a robot is required to navigate from a start point (S) to an endpoint (E), avoiding obstacles (gray boxes) as shown in Figure 5 (Default). A DQN agent can learn this behavior by trial and error and discover the optimal navigation path (blue boxes with step count). Consider a slight change in the environment with a new obstacle as shown by* **X** *in Figure 5 (Minor Change). With this minor change in the environment, the optimal path has changed. However, the new optimal path holds similarities with the previously learned path. The location of the endpoint, few initial steps, and direction to navigate are the same. Instead of re-training a DQN agent, the previously learned behavior from the Default environment can be utilized, and the minor changes can be learned. A DQN agent stores its previous experiences in a replay buffer. The neural network samples data from this buffer for training. When a change is observed in the environment, the previous experiences become irrelevant. The policy to interact in the newer environment requires experiences from it. Using this idea, when a change in the DBMS workload pattern is triggered, we remove all experiences from the replay buffer. The replay buffer is reloaded with the newer experiences. This allows the agent to sample from newer experiences and learn the new optimal policy. If there are Major changes to the environment, such an approach still (endpoint, the direction of navigation, and initial step remain the same) works, but this may take longer to adjust to the newer environment.*

The change in query workload trends could be a *minor* or a *major* change. Using *Priority Experience Sweeping* **Indexer++** can adjust to the changes without retraining from scratch. This process makes **Indexer++** a generic tool for online configuration tuning. We evaluate **Indexer++** in both *minor* and *major* changes in an environment as described in § 5.2.2.

Algorithm 1 Random Query Generator

```
1:  Initialize maximum number of columns in a query $C$      ▷ 1-3
2:  Initialize number of queries $Q$                          ▷ 100
3:  for each $q \in Q$ do                           ▷ number of queries
4:      for each $c \in C$ do                       ▷ number of columns
5:          Randomly extract a distinct value
6:          Randomly select operator [>, <, =, =>, <=]
7:          Randomly select predicates [and, or]
8:          Append $q$
9:      end for
10: end for
```

5 EXPERIMENTS

We perform multiple experiments in different scenarios to evaluate **Indexer++**. We also evaluate both phases of our framework individually on multiple datasets. This section describes the datasets, experimental objectives, results, and conclusion.

5.1 Dataset Description

(1) **IMDB** [25] is a real-world movie database. The dataset has 21 tables and 33 query sets consisting of 2-4 queries each (a total of 112 queries). We divide the set of queries into 3 workloads. The first ten sets with 37 queries create `Workload 1`, sets 11-20 with 38 queries are `Workload 2` and sets 21-33 with 37 queries form `Workload 3`. The purpose of splitting query sets into 3 workloads is create multiple scenarios and to analyze the performance of our framework for both trigger detection and online index recommendation with (potentially) changing workloads.
(2) **TPC-H** is an industrial benchmark for databases. `TCP-H` does not have a fixed set of queries, rather it has 22 query templates that can be used to generate queries. We use a custom template to create a random query generator; as shown in Algorithm 1.

We will refer to randomly generated queries as `TPC-H Random`, template queries as `TPC-H Template` and the IMDB queries as `IMDB`.

5.2 Experimental Setup and Results

All experiments were performed on a computer with Intel i7 5820k, Nvidia 1080ti, 32GB of RAM on Ubuntu 18.04 OS. We used Python 3.7 and its libraries. The models were trained using an Nvidia 1080ti with CUDA and cuDNN support for performance enhancement.

5.2.1 ***Experiment Objective 1:*** *What is the most effective combination of universal embedding, clustering, and dimensionality reduction methods for SQL workload representation?*

On the `TPC-H Random` and `IMDB` datasets, we perform experiments using several popular methods. We specifically evaluate pre-trained models based on BERT, ALBERT, RoBERTa, XLM, and Transformer-T5 bidirectional transformers. We use the prominent dimensionality reduction techniques Principal Component Analysis (PCA), t-Stochastic Neighborhood Embedding (t-SNE), and Uniform Manifold Approximation and Projection (UMAP), as well as the clustering methods K-Means, K-Means++, and K-Medoids.

The chosen NLP models are trained over word tokens and use a specific set of rules to parse a given text. We employ the learnt tokenizer from the pre-trained models for initial preprocessing.

Vector Representation				Dimentionality Reduction				Clustering			
With SQL Keyword		No SQL Keyword		With SQL Keyword		No SQL Keyword		With SQL Keyword		No SQL Keyword	
BERT	0.77	BERT	0.66	UMAP	0.79	UMAP	0.74	K-Mn	0.78	K-Mn	0.72
ALBERT	0.70	ALBERT	0.57	PCA	0.77	PCA	0.70	K++	0.79	K++	0.72
RoBERTa	0.85	RoBERTa	0.82	TSNE	0.80	TSNE	0.73	K-Md	0.79	K-Md	0.73
XLM	0.57	XLM	0.48								
T5	0.80	T5	0.71								

Table 1: Average accuracies on TPC-H Random and IMDB on Dimentionality Reduction, Vector Representation and Clustering

The tokenization of a workload returns a high-dimensional tensor representation. We also reduce the dimensionality and build clusters from the tensor representation. For `TPC-H Random`, we randomly select three columns and generate 25 queries for each column, totalling 75 queries. The `TPC-H Random` and `IMDB` datasets are pre-labeled with respective clusters. The labeling on `IMDB` is based on the workload set and for `TPC-H Random` is based on the columns fetched in the queries. We measure the accuracy of the computed clusters (sorted w.r.t IDs) as follows:

$$accuracy(y', y) = \frac{1}{n}\frac{1}{m}\sum_{i=0}^{n-1}\sum_{j=0}^{m-1}(y'_{ij} == y_{ij}) \begin{cases} 1, & \text{if, } equal \\ 0, & otherwise \end{cases}$$

where n is the number of clusters and m is the number of data points in the cluster. We also conduct our experiment by maintaining and removing SQL keywords (*e.g.,* SELECT). The average accuracy is reported in Table 1. Overall, we observe the following.

- The average cluster detection performance with SQL keywords is 8.9% better than without SQL keywords in terms of accuracy.
- The average accuracy of RoBERTa out-performed other pre-trained models both with and without SQL keywords.
- K-Means++ performs best with keywords and K-Medoids without keywords. Taking the average of both (with and without keyword) K-Medoids performs better than K-Means++.
- t-SNE and UMAP perform better than the other dimensionality reduction techniques. Taking the average of both (with and without keyword) t-SNE overall performs best.

To visually observe the workload representation with and without SQL keywords we plot the reduced data in Figure 6 using RoBERTa and t-SNE. The visual representation for `TPC-H Template` queries has no major distinction. In general all queries are equally set apart. It shows that the queries are different from each other. This analysis is validated from the actual queries. The workload designed by QGEN using `TPC-H Template` represents 22 different queries. They retrieve different parts of the data and can be clearly distinguished. However, in the representation for `TPC-H Random` and `IMDB`, we observe similarities. The data can be observed in natural clusters. When comparing the visual quality of clusters, they are quite similar before and after pre-processing. Overall, in our experiments we observe RoBERTa, t-SNE, and K-Medoids to be an effective combination for SQL workload representation.

5.2.2 *Experiment Objective 2: Can DQN with priority experience sweeping learn and identify workload trends?*

To evaluate our DQN algorithm we simulate two different scenarios: *future* workloads that are entirely different from the *existing* workload (which we call a *major* change) using `IMDB`, and workloads

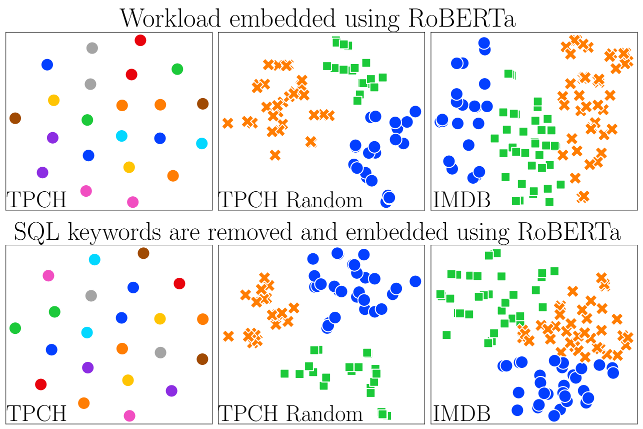

Figure 6: TPC-H Template (TPC-H T), TPC-H Random (TPC-H Ra), and IMDB workload representation using t-SNE (perplexity = 30, iterations = 5000) with and without keywords. The style of marker represents different workloads

that are slightly different (a *minor* change) using `TPC-H Random` dataset. In this experiment, we aim to measure the online efficacy of the workload trend detection. We divided the queries from both datasets into three workloads as described in § 5.1. Our generated random queries from `TPC-H-Random` consists of three workloads, where Workload 1 is queries on a single column, Workload 2 on exactly two columns and Workload 3 on three columns. Some sample queries from each workload are shown below:

```
SELECT COUNT(*) FROM LINEITEM WHERE L_PARTKEY = 30217
SELECT COUNT(*) FROM LINEITEM WHERE L_SUPPKEY = 17438 AND L_PARTKEY < 356077
SELECT COUNT(*) FROM LINEITEM WHERE L_SUPPKEY > 16616 AND L_TAX < 0.06 AND
        L_PARTKEY > 82374
```

The cumulative rewards graph for both datasets is shown in Figure 7 where each experiment has 15,000 episodes, and Workloads 2 and 3 are introduced at 5,000 and 10,000 episodes, respectively. We compare cumulative rewards generated by a DQN algorithm, DQN with PER (Priority Experience Replay), and DQN with PER and PES (Priority Experience Sweeping). We observe a sharp decline for `IMDB` in cumulative rewards at episodes 5,000 and 10,000, followed by a gradual increase. The sharp decline helps understand the change in workload and the ability of workload trend detection to identify the change and trigger a re-indexing. The indexes selected initially for Workload 1 were not as effective for Workload 2. This result aligns with the dataset as all three workloads of `IMDB` queries represent a *major* change. In the `TPC-H Random` the query workloads represent *minor* changes. As shown in Figure 7, both Workloads 2 and 3 improved their performance as a result of the

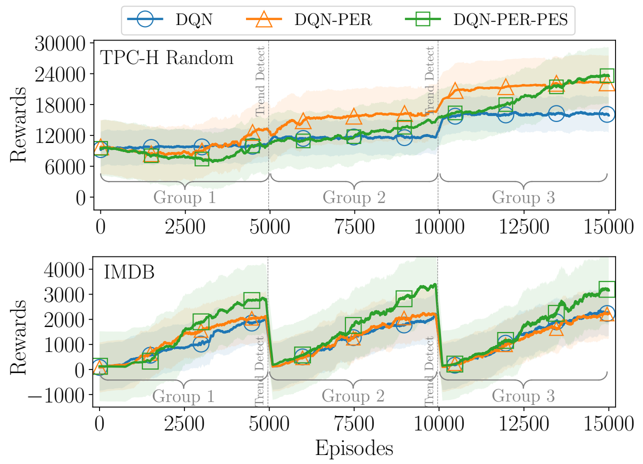

Figure 7: Cumulative Rewards on IMDB and TPC-H Random. The workloads are introduced in three stages starting with Workload 1 as the initial workload then 2 and 3

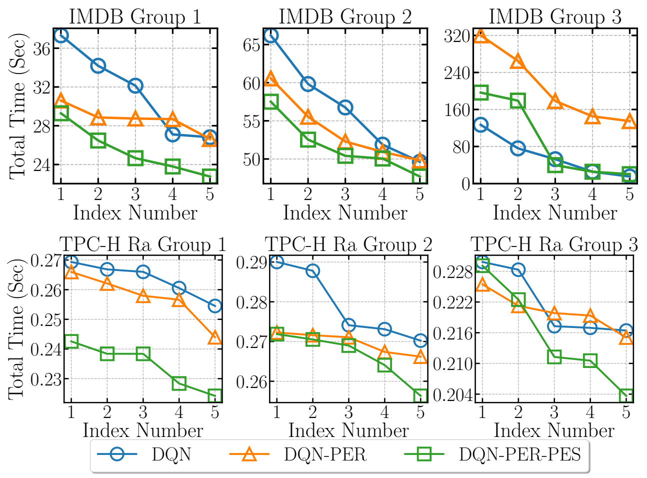

Figure 8: Workload execution cost with selected indexes on TPC-H Random and IMDB dataset

previous set of indices. When comparing all three forms of DQN, we observe that in `TPC-H Random` dataset cumulative rewards for DQN and DQN-PER saturates after a certain time. However, with DQN-PER-PES, the rewards collection is slow but consistent. The rewards are lower in Workloads 1 and 2; however, the change (Δ) in the rewards collection is greater. It shows the ability of the DQN model with PES to gradually learn the environment. With the help of the sweeping technique, the growth of cumulative rewards is progressive rather than saturated. We observe a similar behavior on `IMDB` dataset. Overall, the experiments show that **Indexer++** was able to detect both *major* and *minor* workload changes and was also able to learn and select appropriate indexes (as evidenced by the increase in cumulative rewards). This experiment evaluates the efficacy of our framework on two datasets in different scenarios.

5.2.3 ***Experiment Objective 3:*** *Are selected indexes efficient and reduce the overall workload execution cost?*

In this experiment, we evaluate the efficacy of selected indexes on the overall workload execution cost. To measure the effect of the selected indexes, we execute the benchmark datasets and estimate execution cost after materializing every index. The results of this experiment are displayed in Fig 8. We compare execution cost from DQN, DQN-PER, DQN-PER-PES algorithms. We observe an ideal behaviour where the execution cost in reduced with every index created. We also observe it is critical to get the first optimal index for the overall reduction of cost. In experiments with `IMDB`, we observe **Indexer++** with PES to be most effect for Workloads 1 and 2. In `TPC-H Random`, our framework was effective in all workloads.

5.2.4 ***Overcoming Challenges:*** In § 1, we introduced several challenges for achieving online index tuning. **Indexer++** addresses all of the challenges as described below.

(C1) **Noise Resilience**: Trend detection has a minimum query threshold (batch size of DQN $n = 32$) for a workload. This ensures that a few outliers do not trigger index selection. The clustering of workload embedding helps to reduce the impact of outliers. Finally, **Indexer++** also learns a sensitivity parameter, λ, to help mitigate noise.

(C2) **Overhead**: Unlike other approaches **Indexer++** uses hypothetical indexes[2], rather than creating an actual index. Index creation is expensive and if **Indexer++** were to create many indexes during index selection (as do other approaches) then the overhead would be very high, but hypothetical indexes are not physically materialized. This dramatically lowers the cost of index selection and ensures that the cost of tuning does not affect DBMS performance. **Indexer++** can be used concurrently with other DBMS activities without impacting.

(C3) **Trend Detection**: We perform workload embedding and analytics using pre-trained NLP models. A pre-trained model can infer a trend quickly, and without incurring the high cost of training while **Indexer++** is selecting indexes.

(C4) **Responsiveness**: We eliminate the cost of re-training of the DQN agent using *Priority Experience Sweeping*, further improving the response time. Our framework can analyze workload and select indexes in a short period of time (~15min).

6 CONCLUSION

In this paper, we propose **Indexer++** a real-time solution for online index selection using pre-trained NLP models and Reinforcement Learning. We describe our novel approach for detection of a change in workload pattern. In our extensive experiments, we observe SQL workload embedding to be effective using a combination of t-SNE, K-Medoids and RoBERTa. We propose a novel *priority experience sweeping* as an extension to DQN for online index selection. We evaluate our approach in multiple experiments using both real-world and benchmark datasets. In our experiments, we observe that **Indexer++** is able to address all the existing challenges for an online index tuning namely, noise resilience, overhead cost, trend detection time, and response time, and respond to changing workload patterns by selecting an optimal set of indexes. In the future, we plan to extend our workload trend detection to predict runtime, cardinality estimate and workload errors. We also intend to investigate the impact of SQL operators on workload representation.

[2]https://github.com/HypoPG

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Award No. DBI-1759965, *Collaborative Research: ABI Development: Symbiota2: Enabling greater collaboration and flexibility for mobilizing biodiversity data*. Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation. We would like to extend our appreciation to all of the manuscript reviewers who provided us with valuable feedback that helped us to significantly improve the overall quality of our work.

REFERENCES

[1] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. *Proceedings of the 2017 ACM International Conference on Management of Data* (2017).

[2] Bortik Bandyopadhyay, Pranav Maneriker, Vedang Patel, Saumya Yashmohini Sahai, Ping Zhang, and Srinivasan Parthasarathy. 2020. DrugDBEmbed : Semantic Queries on Relational Database using Supervised Column Encodings. *ArXiv* abs/2007.02384 (2020).

[3] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. 2015. Cost-Model Oblivious Database Tuning with Reinforcement Learning. In *DEXA*. 253–268.

[4] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 8 (2013), 1798–1828.

[5] Rajesh R. Bordawekar, Bortik Bandyopadhyay, and Oded Shmueli. 2017. Cognitive Database: A Step towards Endowing Relational Databases with Artificial Intelligence Capabilities. *ArXiv* abs/1712.07199 (2017).

[6] Rajesh R. Bordawekar and Oded Shmueli. 2017. Using Word Embedding to Enable Semantic Queries in Relational Databases. *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning* (2017).

[7] Nicolas Bruno and Surajit Chaudhuri. 2007. An Online Approach to Physical Design Tuning. *2007 IEEE 23rd International Conference on Data Engineering* (2007), 826–835.

[8] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020).

[9] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "What-If" Index Analysis Utility. *SIGMOD Rec.* 27, 2, 367–378. https://doi.org/10.1145/276305.276337

[10] Surajit Chaudhuri and Vivek R Narasayya. 1997. An efficient, cost-driven index selection tool for Microsoft SQL server. In *VLDB*, Vol. 97. Citeseer, 146–155.

[11] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. *Proceedings of the 2019 International Conference on Management of Data* (2019).

[12] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. *Proceedings of the 2019 International Conference on Management of Data* (2019).

[13] Martin R. Frank, Edward Omiecinski, and Shamkant B. Navathe. 1992. Adaptive and Automated Index Selection in RDBMS. In *EDBT '92*. 277–292.

[14] Michael Günther. 2018. FREDDY: Fast Word Embeddings in Database Systems. *Proceedings of the 2018 International Conference on Management of Data* (2018).

[15] Michael Hammer and Arvola Chan. 1976. Index Selection in a Self-Adaptive Data Base Management System. In *SIGMOD '76*. 1–8.

[16] Geoffrey E Hinton et al. 1986. Learning distributed representations of concepts. In *Eighth annual conference of the cognitive science society*, Vol. 1. 12.

[17] Anil K Jain, M Narasimha Murty, and Patrick J Flynn. 1999. Data clustering: a review. *ACM computing surveys (CSUR)* 31, 3 (1999), 264–323.

[18] Shrainik Jain, Bill Howe, Jiaqi Yan, and Thierry Cruanes. 2018. Query2Vec: An Evaluation of NLP Techniques for Generalized Workload Analytics. *arXiv preprint arXiv:1801.05613* (2018).

[19] Herald Kllapi, Ilia Pietri, Verena Kantere, and Yannis E Ioannidis. 2020. Automated Management of Indexes for Dataflow Processing Engines in IaaS Clouds. In *EDBT*.

[20] Piotr Kołaczkowski and Henryk Rybiński. 2009. Automatic index selection in RDBMS by exploring query execution plan space. In *Advances in Data Management*. Springer, 3–24.

[21] Jan Kossmann and R. Schlosser. 2019. A Framework for Self-Managing Database Systems. *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)* (2019), 100–106.

[22] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *CIDR*.

[23] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An Index Advisor Using Deep Reinforcement Learning. *Proceedings of the 29th ACM International Conference on Information & Knowledge Management* (2020).

[24] Quoc V. Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. *ArXiv* abs/1405.4053 (2014).

[25] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9 (2015), 204–215.

[26] Omer Levy and Yoav Goldberg. 2014. Linguistic Regularities in Sparse and Explicit Word Representations. In *CoNLL*.

[27] Joseph Lilleberg, Yun Zhu, and Yanqing Zhang. 2015. Support vector machines and Word2vec for text classification with semantic features. *2015 IEEE 14th International Conference on Cognitive Informatics & Cognitive Computing (ICCI*CC)* (2015), 136–140.

[28] Martin Lühring, Kai-Uwe Sattler, Karsten Schmidt, and Eike Schallehn. 2007. Autonomous Management of Soft Indexes. *2007 IEEE 23rd International Conference on Data Engineering Workshop* (2007), 450–458.

[29] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR*.

[30] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. 2018. Reinforcement learning for solving the vehicle routing problem. In *NIPS*. 9839–9849.

[31] Priscilla Neuhaus, Julia M. Colleoni Couto, Jonatas Wehrmann, Duncan Dubugras Alcoba Ruiz, and Felipe Meneguzzi. 2019. GADIS: A Genetic Algorithm for Database Index Selection (S). In *SEKE*.

[32] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*, Vol. 4. 1.

[33] Wendel Góes Pedrozo, Júlio Cesar Nievola, and Deborah Carvalho Ribeiro. 2018. An Adaptive Approach for Index Tuning with Learning Classifier Systems on Hybrid Storage Environments. In *Hybrid Artificial Intelligent Systems*. 716–729.

[34] Gregory Piatetsky-Shapiro. 1983. The optimal selection of secondary indices is NP-complete. *SIGMOD Rec.* 13 (1983), 72–75.

[35] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. Online Index Selection Using Deep Reinforcement Learning for a Cluster Database. *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)* (2020), 158–161.

[36] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. 2006. COLT: continuous on-line tuning. *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (2006).

[37] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. 2009. Index Interactions in Physical Design Tuning: Modeling, Analysis, and Applications. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 1234–1245. https://doi.org/10.14778/1687627.1687766

[38] Aliaksei Severyn and Alessandro Moschitti. 2015. Twitter Sentiment Analysis with Deep Convolutional Neural Networks. *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval* (2015).

[39] Ankur Kumar Sharma, Felix Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. *ArXiv* abs/1801.05643 (2018).

[40] Vishal Sharma. 2021. Deep Learning Data and Indexes in a Database. (2021). https://doi.org/10.26076/62bf-44a1

[41] Vishal Sharma, Curtis E. Dyreson, and Nicholas S. Flann. 2021. MANTIS: Multiple Type and Attribute Index Selection using Deep Reinforcement Learning. *25th International Database Engineering & Applications Symposium* (2021).

[42] Richard S. Sutton and Andrew G. Barto. 2005. Reinforcement Learning: An Introduction. *IEEE Transactions on Neural Networks* 16 (2005), 285–286.

[43] Yi Chern Tan and L. Elisa Celis. 2019. Assessing Social and Intersectional Biases in Contextualized Word Representations. In *NeurIPS*.

[44] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 advisor: an optimizer smart enough to recommend its own indexes. *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)* (2000), 101–110.

[45] Hannes Voigt, Thomas Kissinger, and Wolfgang Lehner. 2013. SMIX: Self-Managing Indexes for Dynamic Workloads. In *SSDBM*. Article 24, 12 pages.

[46] Ji Zhang, Ke Zhou, Guoliang Li, Yu Liu, Ming Xie, Bin Cheng, and Jiashu Xing. 2021. CDBTune+: An efficient deep reinforcement learning-based automatic cloud database tuning system. *The VLDB Journal* (2021).

[47] Shengyu Zhu, Ignavier Ng, and Zhitang Chen. 2020. Causal Discovery with Reinforcement Learning. In *ICLR*.