Automated Database Indexing Using Model-Free Reinforcement Learning

Gabriel Paludo Licks[†] **and Felipe Meneguzzi**[‡]
Pontifical Catholic University of Rio Grande do Sul (PUCRS), Brazil
Graduate Program in Computer Science, School of Technology
[†]gabriel.licks@edu.pucrs.br
[‡]felipe.meneguzzi@pucrs.br

Abstract

Configuring databases for efficient querying is a complex task, often carried out by a database administrator. Solving the problem of building indexes that truly optimize database access requires a substantial amount of database and domain knowledge, the lack of which often results in wasted space and memory for irrelevant indexes, possibly jeopardizing database performance for querying and certainly degrading performance for updating. We develop an architecture to solve the problem of automatically indexing a database by using reinforcement learning to optimize queries by indexing data throughout the lifetime of a database. In our experimental evaluation, our architecture shows superior performance compared to related work on reinforcement learning and genetic algorithms, maintaining near-optimal index configurations and efficiently scaling to large databases.

1 Introduction

Despite the multitude of tools available to manage and gain insights from very large datasets, indexing databases that store such data remains a challenge with multiple opportunities for improvement [27]. Slow information retrieval in databases entails not only wasted time for a business but also indicates a high computational cost being paid. Unnecessary indexed columns, or columns that should be indexed but are not, directly impact the query performance of a database. Nevertheless, achieving the best indexing configuration for a database is not a trivial task [4, 5]. To do so, we have to learn from queries that are running, take into account their performance, the system resources, and the storage budget so that we can find the best index candidates [18].

In an ideal scenario, all frequently queried columns should be indexed to optimize query performance. Since creating and maintaining indexes incur a cost in terms of storage as well as in computation whenever database insertions or updates take place in indexed columns [21], choosing an optimal set of indexes for querying purposes is not enough to ensure optimal performance, so we must reach a trade-off between query and insert/update performance. Thus, this is a fundamental task that needs to be performed continuously, as the indexing configuration directly impacts on a database's overall performance.

We developed an architecture for automated and dynamic database indexing that evaluates query and insert/update performance to make decisions on whether to create or drop indexes using Reinforcement Learning (RL). We performed experiments using a scalable benchmark database, where we empirically evaluate our architecture results in comparison to standard baseline index configurations, database advisor tools, genetic algorithms, and other reinforcement learning methods applied to database indexing. The architecture we implemented to automatically manage indexes through reinforcement learning successfully converged in its training to a configuration that outperforms all baselines and related work, both in performance and in storage usage by indexes.

2 Background

2.1 Reinforcement Learning

Reinforcement learning aims to learn optimal agent policies in stochastic environments modeled as Markov Decision Processes (MDPs) [2]. It is a trial-and-error learning method, where an agent interacts and transitions through states of an MDP environment model by taking actions and observing rewards [23, Ch. 1]. MDP are formally defined as a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{P}$ is a transition probability function which defines the dynamics of the MDP, $\mathcal{R}$ is a reward function and $\gamma \in [0, 1]$ is a discount factor [23, Ch. 3].

In order to solve an MDP, an agent needs to know the state-transition and the reward functions. However, in most realistic applications, modeling knowledge about the state-transition or the reward function is either impossible or impractical, so an agent interacts with the environment taking sequential actions to collect information and explore the search space by trial and error [23, Ch. 1]. The Q-learning algorithm is the natural choice for solving such MDPs [23, Ch. 16]. This method learns the values of state-action pairs, denoted by $Q(s, a)$, representing the value of taking action a in a state s [23, Ch. 6].

Assuming that states can be described in terms of features that are well informative, such problem can be handled by

using linear function approximation, which is to use a parameterized representation for the state-action value function other than a look-up table [25]. The simplest differentiable function approximator is through a linear combination of features, though there are other ways of approximating functions such as using neural networks [23, Ch. 9, p. 195].

2.2 Indexing in Relational Databases

An important technique to file organization in a DBMS is *indexing* [21, Ch. 8, p. 274], and is usually managed by a DBA. However, index selection without the need of a domain expert is a long-time research subject and remains a challenge due to the problem complexity [27, 5, 4]. The idea is that, given the database schema and the workload it receives, we can define the problem of finding an efficient index configuration that optimizes database operations [21, Ch. 20, p. 664]. The complexity stems from the potential number of attributes that can be indexed and all of its subsets.

While DBMSs strive to provide automatic index tuning, the usual scenario is that performance statistics for optimizing queries and index recommendations are offered, but the DBA makes the decision on whether to apply the changes or not. Most recent versions of DBMSs such as Oracle [13] and Azure SQL Database [19] can automatically adjust indexes. However, it is not the case that the underlying system is openly described.

A general way of evaluating DBMS performance is through benchmarking. Since DBMSs are complex pieces of software, and each has its own techniques for optimization, external organizations have defined protocols to evaluate their performance [21, Ch. 20, p. 682]. The goals of benchmarks are to provide measures that are portable to different DBMSs and evaluate a wider range of aspects of the system, e.g., transactions per second and price-performance ratio [21, Ch. 20, p. 683].

2.3 TPC-H Benchmark

The tools provided by TPC-H include a database generator (DBGen) able to create up to 100 TB of data to load in a DBMS, and a query generator (QGen) that creates 22 queries with different levels of complexity. Using the database and workload generated using these tools, TPC-H specifies a benchmark that consists of inserting records, executing queries, and deleting records in the database to measure the performance of these operations.

The TPC-H Performance metric is expressed in Queries-per-Hour ($QphH@Size$), which is achieved by computing the $Power@Size$ and the $Throughput@Size$ metrics [24]. The resulting values are related to its scale factor ($@Size$), i.e., the database size in gigabytes. The $Power@Size$ evaluates how fast the DBMS computes the answers to single queries. This metric is computed using Equation 1:

$$Power@Size = \frac{3600}{\sqrt[24]{\pi_{i=1}^{22} QI(i,0) \times \pi_{j=1}^{2} RI(j,0)}} \times SF \quad (1)$$

where 3600 is the number of seconds per hour and $QI(i,s)$ is the execution time for each one of the queries i. $RI(j,s)$ is the execution time of refresh functions j (insert/update) in the query stream s, and SF is the scale factor or database size, ranging from 1 to $100,000$ according to its $@Size$.

The $Throughput@Size$ measures the ability of the system to process the most queries in the least amount of time, taking advantage of I/O and CPU parallelism [24]. It computes the performance of the system against a multi-user workload performed in an elapsed time, using Equation 2:

$$Throughput@Size = \frac{S \times 22}{T_S} \times 3600 \times SF \quad (2)$$

where S is the number of query streams executed, and T_S is the total time required to run the test for s streams.

$$QphH@Size = \sqrt{Power@Size \times Throughput@Size} \quad (3)$$

Equation 3 shows the Query-per-Hour Performance ($QphH@Size$) metric, which is obtained from the geometric mean of the previous two metrics and reflects multiple aspects of the capability of a database to process queries. The $QphH@Size$ metric is the final output metric of the benchmark and summarizes both single-user and multiple-user overall database performance.

3 Architecture

In this section, we introduce our database indexing architecture to automatically choose indexes in relational databases, which we refer to as SmartIX. The main motivation of SmartIX is to abstract the database administrator's task that involves a frequent analysis of all candidate columns and verifying which ones are likely to improve the database index configuration. For this purpose, we use reinforcement learning to explore the space of possible index configurations in the database, aiming to find an optimal strategy over a long time horizon while improving the performance of an agent in the environment.

The SmartIX architecture is composed of a reinforcement learning agent, an environment model of a database, and a DBMS interface to apply agent actions to the database. The reinforcement learning agent is responsible for the decision making process. The agent interacts with an environment model of the database, which computes system transitions and rewards that the agent receives for its decisions. To make

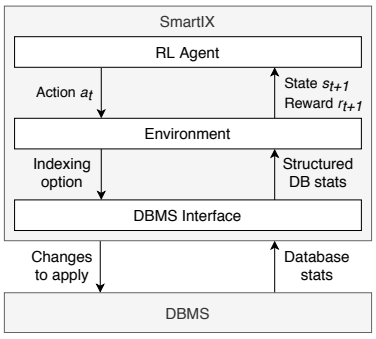


Figure 1: SmartIX architecture.

changes persistent, there is a DBMS interface that is responsible for communicating with the DBMS to create or drop indexes and get statistics of the current index configuration.

3.1 Agent

Our agent is based on the Deep Q-Network agent proposed by [11], depicted in Algorithm 1. The algorithm consists of a Q-learning method that uses a neural network for function approximation trained using experience replay. The neural network is used to approximate the action-value function and is trained using mini-batches of experience randomly sampled from the replay memory. At each time step, the agent performs one transition in the environment. That is, the agent chooses an action using an epsilon-greedy exploration function at the current state, the action is then applied in the environment, and the environment returns a reward signal and the next state. Finally, each transition in the environment is stored in the replay buffer, and the agent performs a mini-batch update in the action-value function.

3.2 Environment

The environment component is responsible for computing transitions in the system and computing the reward function. To successfully apply a transition, we implement a model of the database environment, modeling states that contain features that are relevant to the agent learning, and a transition function that is able to modify the state with regard to the action an agent chooses. Each transition in the environment outputs a reward signal that is fed back to the agent along with the next state, and the reward function has to be informative enough so that the agent learns which actions yield better decisions at each state.

State representation The state is the formal representation of the environment information used by the agent in the learning process. Thus, deciding which information should be used to define a state of the environment is critical for task performance. The amount of information encoded in a state imposes a trade-off for reinforcement learning agents. Specifically, that if the state encodes too little information, then the agent might not learn a useful policy, whereas if the state encodes too much information, there is a risk that the learning algorithm needs too many samples of the environment that it does not converge to a policy.

For the database indexing problem, the state representation is defined as a feature vector $\vec{S} = \vec{I} \cdot \vec{Q}$, which is a result of a concatenation of the feature vectors $\vec{I}$ and $\vec{Q}$. The feature vector $\vec{I}$ encodes information regarding the current index configuration of the database, with length $|\vec{I}| = C$, where C is a constant of the total number of columns in the database schema. Each element in the feature vector $\vec{I}$ holds a binary value, containing 1 or 0, depending on whether the column that corresponds to that position in the vector is indexed or not. The second part of our state representation is a feature vector $\vec{Q}$, also with length $|\vec{Q}| = C$, which encodes information regarding which indexes were used in last queries received by the database. To organize such information, we set a constant value of H that defines the horizon of queries that we keep track of. To each of the last queries in a horizon H, we verify whether any of the indexes currently created in the database are used to run such queries. Each position in the vector $\vec{Q}$ corresponds to a column and holds a binary value that is assigned 1 if such column is indexed and used in the last H queries, else 0. Finally, we concatenate $\vec{I}$ and $\vec{Q}$ to generate the state vector $\vec{S}$ with length $|\vec{S}| = 2C$.

Algorithm 1 Database indexing agent. Adapted from [11].

```
Random initialization of the value function
Empty initialization of a replay memory D
s ← DB initial index configuration
for each step do
    a ← epsilon greedy(s)
    s', r ← execute(a)
    Store experience e = ⟨s, a, r, s'⟩ in D
    Sample random mini-batch of experiences e ∼ D
    Perform experience replay using mini-batch
    s ← s'
```

Actions In our environment, we define the possible actions as a set A of size $C + 1$. Each one of the C actions refers to one column in the database schema. These actions are implemented as a "flip" to create or drop an index in the current column. Therefore, for each action, there are two possible behaviors: CREATE INDEX or DROP INDEX on the corresponding column. The last action is a "do nothing" action, that enables the agent not to modify the index configuration in case it is not necessary at the current state.

Reward Deciding the reward function is critical for the quality of the ensuing learned policy. On the one hand, we want the agent to learn that indexes that are used by the queries in the workload must be maintained in order to optimize such queries. On the other hand, indexes that are not being used by queries must not be maintained as they consume system resources and are not useful to the current workload. Therefore, we compute the reward signal based on the next state's feature vector $\vec{S}$ after an action is applied, since our state representation encodes information both on the current index configuration and on the indexes used in the last queries, i.e. information contained in vectors $\vec{I}$ and $\vec{Q}$. Our reward function is computed using Equation 4:

$$\begin{aligned} R(op, use) = (1 - op)((1 - use)(1) + (use)(-5)) \\ +(op)((1 - use)(-5) + (use)(1)) \end{aligned} \quad (4)$$

where $op = I_c$ and $use = Q_c$. That is, the first parameter op holds 0 if the last action represents a dropped index in column c, or 1 if created an index. The latter parameter, use, holds 0 if an index in column c is not being used by the last H horizon queries, and 1 otherwise.

Therefore, our reward function returns a value of $+1$ if an index is created and it actually benefits the current workload, or if an index is dropped and it is not beneficial to the current workload. Otherwise, the function returns -5 to penalize the agent if an index is dropped and it is beneficial to the current workload, or an index is created and it does not benefit the current workload. The choice of values $+1$ and

-5 is empirical. However, we want the penalization value to be at least twice smaller than the $+1$ value, so that the values do not get canceled when accumulating with each other. Finally, if the action corresponds to a "do nothing" operation, the environment simply returns a reward of 0, without computing Equation 4.

4 Experiments

4.1 Experimental setup

Database setup Due to its usage in literature for measuring database performance, we choose to run experiments using the database schema and data provided by the TPC-H benchmark. The tools provided by TPC-H include a data generator (DBGen), which is able to create up to 100TB of data to load in a DBMS, and a query generator (QGen) that creates 22 queries with different levels of complexity. The database of these experiments is populated with 1GB of data. To run benchmarks using each baseline index configuration, we implemented the TPC-H benchmark protocol using a Python script that runs queries, fetches execution time, and computes the performance metrics.

To provide statistics on the database, we show in Table 1 the number of columns that each table contains and an analysis on the indexing possibilities. For that, we mapped for each table in the TPC-H database the total number of columns, the columns that are already indexed (primary and foreign keys, indexed by default), and the remaining columns that are available for indexing.

By summing the number of indexable columns in each table, we have a total of 45 columns that are available for indexing. Since a column is either indexed or not, there are two possibilities for each of the remaining 45 indexable columns. This scenario indicates that we have exactly $35,184,372,088,832$ (2^{45}), i.e. more than 35 trillion, possible configurations of *simple* indexes. Thus, this is also the number of states that can be assumed by the database indexing configuration and therefore explored by the algorithms.

For comparison purposes, we run a brute force procedure to identify which columns compose the ground truth optimal index configuration among all possibilities. That is, we identify for each index possibility whether it is used to compute at least one query within the 22 TPC-H queries. To check whether an index is used or not, we use the EXPLAIN command to view the execution plan of each query. Finally, we have 6 columns from the TPC-H that compose our ground truth optimal indexes: C_ACCTBAL, L_SHIPDATE, O_ORDERDATE, P_BRAND, P_CONTAINER, P_SIZE.

Table 1: TPC-H database - Table stats and indexes

Table	Total	Indexed	Indexable
REGION	3	1	2
NATION	4	2	2
PART	9	1	8
SUPPLIER	7	2	5
PARTSUPP	5	2	3
CUSTOMER	8	2	6
ORDERS	9	2	7
LINEITEM	16	4	12
Totals	61	16	45

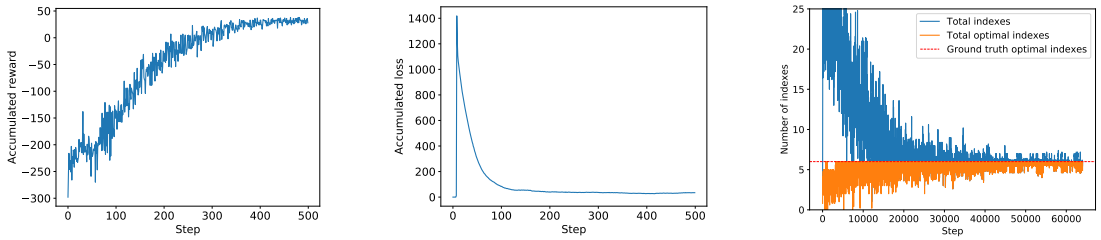
Baselines The baselines comprise different indexing configurations using different indexing approaches, including commercial and open-source database advisors, and related work on genetic algorithms and reinforcement learning methods. Each baseline index configuration is a result of training or analyzing the same workload of queries, from the TPC-H benchmark, in order to make an even comparison between the approaches. The following list briefly introduces each of them. *Default*: indexes only on primary and foreign keys; *All indexed*: all columns indexed. *Random*: indexes randomly explored by an agent; *EDB* 2019 and *POWA* 2019: indexes obtained using a comercial and an open-source advisor tool, respectively. *ITLCS* 2018 and *GADIS* 2019: indexes obtained using genetic algorithms related work; *NoDBA* 2018 and *rCOREIL* 2016: indexes obtained using reinforcement learning related work.

The EDB 2019, POWA 2019, and ITLCS 2018 index configurations are a result of a study conducted by Pedrozo, Nievola, and Ribeiro [17]. The authors [17] employ these methods to verify which indexes are suggested by each method to each of the 22 queries in the TPC-H workload, whose indexes constitute the respective configurations we use in this analysis. The index configurations of GADIS 2019, NoDBA 2018, and rCOREIL 2016 are a result of experiments we ran using source-code provided by the authors. We execute the author's algorithms without modifying any hyper-parameter except configuring the database connection. The index configuration we use in this analysis is the one in which each algorithm converged to, when the algorithm stops modifying the index configuration or reaches the end of training.

4.2 Agent training

Training the reinforcement learning agent consists of time steps of agent-environment interaction and value function updates until it converges to a policy as desired. In our case, to approximate the value function, we use a simple multi-layer perceptron neural network with two hidden layers and ReLU activation, and an Adam optimizer with mean-squared error loss, both PyTorch 1.5.1 implementations using default hyperparameters [15]. The input and output dimensions depend on the number of columns available to index in the database schema, as shown in Section 3.2.

The hyperparameters used while training are set as follows. The first, *learning rate* $\alpha = 0.0001$ and *discount factor* $\gamma = 0.9$, are used in the update equation of the value function. The next are related to experience replay, where *replay memory size* $= 10000$ defines the number of experiences the agent is capable of storing, and *replay batch size* $= 1024$ defines the number of samples the agent uses at each time step to update the value function. The last are related to the epsilon-greedy exploration function, where we define an *epsilon initial* $= 1$ as maximum epsilon value, an *epsilon final* $= 0.01$ as epsilon minimum value, a percentage in which *epsilon decays* $= 1\%$, and the interval of *time steps at each decay* $= 128$.



(a) Accumulated reward per 128 steps.

(b) Accumulated loss per 128 steps.

(c) Index configurations while training.

Figure 2: Training statistics.

We train our agent for the course of 64 thousand time steps in the environment. Training statistics are gathered every 128 steps and are shown in Figure 2. Sub-figure 2a shows the total reward accumulated by the agent at each 128 steps in the environment, which consistently improves over time and stabilizes after the 400th x-axis value. Sub-figure 2b shows the accumulated loss at each 128 steps in the environment, i.e. the errors in predictions of the value function during experience replay, and illustrates how it decreases towards zero as parameters are adjusted and the agent approximates the true value function.

To evaluate the agent behavior and the index configuration in which the agent is converging to, we plot in Figure 2c each of the index configurations explored by the agent in the 64 thousand training steps. Each index configuration is represented in terms of *total indexes* and *total optimal indexes* a configuration contains. *Total indexes* is simply a count on the number of indexes in the configuration, while *total optimal indexes* is a count on the number of ground truth optimal indexes in the configuration. The lines are smoothed using a running mean of the last 5 values, and a fixed red dashed line across the x-axis represents the configuration in which the agent should converge to. As we can see, both the total amount of indexes *and* the total optimal indexes converge towards the ground truth optimal indexes. That is, the agent learns both to keep the optimal indexes in the configuration, as well as to drop irrelevant indexes for the workload.

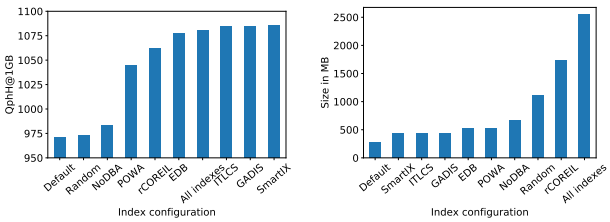
4.3 Performance Comparison

We now evaluate each baseline index configuration in comparison to the one in which our agent converged to in the last episode of training. We show the TPC-H performance metric (QphH, i.e. the query-per-hour metric) and the index size of each configuration. Figure 3a shows the query-per-hour metric of each configuration (higher values denote better performance). The plotted values consist of a trimmed mean of 12 executions of the TPC-H benchmark for each index configuration, removing the highest and the lowest result and averaging the 10 remaining results. Figure 3b shows the disk space required for the indexes in each configuration (index size in MB), which allows us to analyze the trade-off in the number of indexes and the resources needed to maintain it. In an ideal scenario, the index size is just the bare minimum to maintain the indexes that are necessary to support query performance.

Yet SmartIX achieves the best query-per-hour-metric, the two genetic algorithms [12] and [17] have both very similar query-per-hour and index size metrics in comparison to our agent. GADIS [12] itself uses a similar state-space model to SmartIX, with individuals being represented as binary vectors of the indexable columns. The fitness function GADIS optimizes is the actual query-per-hour metric, and it runs the whole TPC-H benchmark every time it needs to compute the fitness function. Therefore, it is expected that it finds an individual with a high performance metric, although it is unrealistic for real-world applications in production due to the computational cost of running the benchmark.

Indexing all columns is among the highest query-per-hour results and can seem to be a natural alternative to solve the indexing problem. However, it results in the highest amount of disk used to maintain indexes stored. Such alternative is less efficient in a query-per-hour metric as the benchmark not only takes into account the performance of SELECT queries, but also INSERT and DELETE operations, whose performance is affected by the presence of indexes due to the overhead of updating and maintaining the structure when records change [21, Ch. 8, p. 290-291]. It has the lowest ratio due to the storage it needs to maintain indexes.

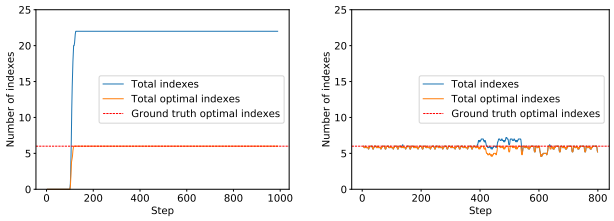
While rCOREIL [1] is the most competitive among the reinforcement learning baselines, the amount of storage used to maintain its indexes is the highest among all baselines (except for having all columns indexed). rCOREIL does not handle whether primary and foreign key indexes are already created, causing it to create duplicate indexes. The policy iteration algorithm used in rCOREIL is a dynamic programming method used in reinforcement learning, which is characterized by complete sweeps in the state space at each iteration in order to update the value function. Since dynamic programming methods are not suitable to large state spaces [23, Ch. 4, p. 87], this can become a problem in



(a) Query-per-hour metric

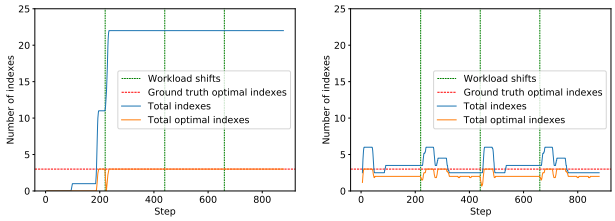
(b) Index size (in MB)

Figure 3: Static index configurations results.



(a) rCOREIL (b) SmartIX

Figure 4: Agent behavior with a fixed workload.



(a) rCOREIL (b) SmartIX

Figure 5: Agent behavior with a shifting workload.

databases that contain a larger number of columns to index.

Among the database advisors, the commercial tool EDB [6] achieves the highest query-per-hour metric in comparison to the open-source tool POWA [20], while its indexes use virtually the same disk space. Other baselines and related work can optimize the index configuration with lightweight index sizes, but are not competitive in comparison to the previously discussed methods in terms of the query-per-hour performance metric. Finally, among all the baselines, the index configuration obtained using SmartIX not only yields the best query-per-hour metric but also the smallest index size (except for the default configuration), i.e. it finds the balance between performance and storage, as shown in the ratio plot.

5 Dynamic configurations

This section aims to evaluate the behavior of algorithms that generate policies, i.e. generate a function that guides an agent's behavior. The three algorithms that generate policies are SmartIX, rCOREIL, and NoDBA. The three are reinforcement learning algorithms, although using different strategies (see Sec. 6). While rCOREIL and SmartIX show a more interesting and dynamic behavior, the NoDBA algorithm shows a fixed behavior and keeps only three columns indexed over the whole time horizon, without changing the index configuration over time (see its limitations in Sec. 6). Therefore, we do not include NoDBA in the following analysis and focus the discussion on rCOREIL and SmartIX.

5.1 Fixed workload

We now evaluate the index configuration of rCOREIL and SmartIX over time while the database receives a fixed workload of queries. Figure 4 shows the behavior of rCOREIL and SmartIX, respectively. Notice that rCOREIL takes some time to create the first indexes in the database, after receiving about 150 queries, while SmartIX creates indexes at the very beginning of the workload. On the one hand, rCOREIL shows a fixed behavior maintains all ground truth optimal indexes, but it creates a total of 22 indexes, 16 of those being unnecessary indexes and the remaining 6 are optimal indexes. On the other hand, SmartIX shows a dynamic behavior and consistently maintains 5 out of the 6 ground truth optimal indexes, and it does not maintain unnecessary indexes throughout most of the received workload.

5.2 Shifting workload

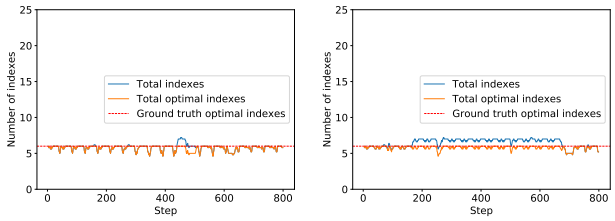
We now evaluate the algorithm's behavior while receiving a workload that shifts over time. To do so, we divide the 22 TPC-H queries into two sets of 11 queries, where for each set there is a different ground truth set of indexes. That is, out of the 6 ground truth indexes from the previous fixed workload, we now separate the workload to have 3 indexes that are optimal first set of queries, and 3 other indexes that are optimal for the second set of queries. Therefore, we aim to evaluate whether the algorithms can adapt the index configuration over time when the workload shifts and a different set of indexes is needed according to each of the workloads.

The behavior of each algorithm is shown in Figure 5. The vertical dashed lines placed along the x-axis represent the time step where the workload shifts from one set of queries to another, and therefore the set of ground truth optimal indexes also changes. On the one hand, notice that rCOREIL shows a similar behavior from the one in the previous fixed workload experiment, in which it takes some time to create the first indexes, and then maintains a fixed index configuration, not adapting as the workload shifts. On the other hand, SmartIX shows a more dynamic behavior with regard to the shifts in the workload. Notice that, at the beginning of each set of queries in the workload, there is a peak in the total indexes, which decreases as soon as the index configuration adapts to the new workload and SmartIX drops the unnecessary indexes with regard to the current workload. Even though rCOREIL maintains all 3 ground truth indexes over time, it still maintains 16 unnecessary indexes, while SmartIX consistently maintains 2 out of 3 ground truth optimal indexes and adapts as the workload shifts.

5.3 Scaling up database size

In the previous sections, we showed that the SmartIX architecture can consistently achieve near-optimal index configurations in a database of size 1GB. In this section, we report experiments on indexing larger databases, where we transfer the policy trained in the 1GB database to perform indexing in databases with size 10GB and 100GB. We plot the behavior of our agent in Figure 6.

As we can see, the agent shows a similar behavior to the one using a 1GB database size reported in previous experiments. The reason is that both the state features and the reward function are not influenced by the database size. The only information relevant to the state and the reward function is the current index configuration and the workload being received. Therefore, we can successfully transfer the



(a) 10GB TPC-H database. (b) 100GB TPC-H database.

Figure 6: Agent behavior in larger databases.

value function learned in smaller databases to index larger databases, consuming fewer resources to train the agent.

6 Related Work

Machine learning techniques are used in a variety of tasks related to database management systems and automated database administration [26]. One example is the work from Kraska et al. [8], which outperforms traditional index structures used in current DBMS by replacing them with learned index models, having significant advantages under particular assumptions. Pavlo et. al [16] developed Peloton, which autonomously optimizes the system for incoming workloads and uses predictions to prepare the system for future workloads. In this section, though, we further discuss related work that focused on developing methods for optimizing queries through automatic index tuning. Specifically, we focus our analysis on work that based their approach on reinforcement learning techniques.

Basu et al. [1] developed a technique for index tuning based on a cost model that is learned with reinforcement learning. However, once the cost model is known, it becomes trivial to find the configuration that minimizes the cost through dynamic programming, such as the policy iteration method used by the authors. They use DBTune [3] to reduce the state space by considering only indexes that are recommended by the DBMS. Our approach, on the other hand, focuses on finding the optimal index configuration without having complete knowledge of the environment and without heuristics of the DBMS to reduce the state space.

Sharma et al. [22] use a cross-entropy deep reinforcement learning method to administer databases automatically. Their set of actions, however, only include the creation of indexes, and a budget of 3 indexes is set to deal with space constraints and index maintenance costs. Indexes are only dropped once an episode is finished. A strong limitation in their evaluation process is to only use the LINEITEM table to query, which does not exploit how indexes on other tables can optimize the database performance, and consequently reduces the state space of the problem. Furthermore, they do not use the TPC-H benchmark performance measure to evaluate performance but use query execution time.

Reinforcement learning can also optimize queries by predicting query plans: Marcus et al. [10] proposed a proof-of-concept to determine the join ordering for a fixed database; Ortiz et al. [14] developed a learning state representation to predict the cardinality of a query. These approaches could possibly be used alongside ours, generating better plans to query execution while we focus on maintaining indexes that these queries can benefit from.

7 Conclusion

In this research, we developed the SmartIX architecture for automated database indexing using reinforcement learning. The experimental results show that our agent consistently outperforms the baseline index configurations and related work on genetic algorithms and reinforcement learning. Our agent is able to find the trade-off concerning the disk space the index configuration occupies and the performance metric it achieves. The state representation and the reward function allows us to successfully index larger databases while training in smaller databases and consuming fewer resources.

Regarding the limitations of our architecture, we do not yet deal with composite indexes due to the resulting state space of all possible indexes that use two or more columns. Our experiments show results using workloads that are read-intensive (i.e. intensively fetching data from the database), which is exactly the type of workload that benefits from indexes. However, experiments using write-heavy workloads (i.e. intensively writing data to the database) can be interesting to verify whether the agent learns to avoid indexes in write-intensive tables. Considering these limitations, in future work, we plan to: (1) investigate techniques that allow us to deal with composite indexes; (2) improve the reward function to provide feedback in case of write-intensive workloads; (3) investigate pattern recognition techniques to predict incoming queries to index ahead of time; and (4) evaluate SmartIX on big data ecosystems (e.g. Hadoop).

Our contributions include: (1) a formalization of a reward function shaped for the database indexing problem, independent of DBMS's statistics, that allows the agent to adapt the index configuration according to the workload; (2) an environment representation for database indexing that is independent of schema or DBMS; and (3) a reinforcement learning agent that efficiently scales to large databases, while trained in small databases consuming fewer resources. The model in this paper is novel in comparison to early work previously published at the Applied Intelligence journal [9].

In closing, we envision this kind of architecture being deployed in cloud platforms such as Heroku and similar platforms that often provide database infrastructure for various clients' applications. The reality is that these clients do not prioritize, or it is not in their scope of interest to focus on database management. Especially in the case of early-stage start-ups, the aim to shorten time-to-market and quickly ship code motivates externalizing complexity on third party solutions [7]. From an overall platform performance point of view, having efficient database management results in an optimized use of hardware and software resources. Thus, in the absence of a database administrator, the SmartIX architecture is a potential stand-in solution, as experiments show that it provides at least equivalent and often superior indexing choices compared to baseline indexing recommendations.

Acknowledgement: This work was supported by SAP SE. We thank our colleagues from SAP Labs Latin America who provided insights and expertise that greatly assisted the research.

References

[1] Basu, D.; Lin, Q.; Chen, W.; Vo, H. T.; Yuan, Z.; Senellart, P.; and Bressan, S. 2016. Regularized cost-model oblivious database tuning with reinforcement learning. *Trans. on Large-Scale Data- and Knowledge-Centered Systems* 28:96–132.

[2] Bellman, R. 1957. A markovian decision process. *Journal of Mathematics and Mechanics* 6:679–684.

[3] DB Group at UCSC. 2019. DBTune. Retrieved from URL github.com/dbgroup-at-ucsc/dbtune.

[4] Duan, S.; Thummala, V.; and Babu, S. 2009. Tuning database configuration parameters with iTuned. *Very Large Data Base Endowment* 2:1246–1257.

[5] Elfayoumy, S., and Patel, J. 2012. Database performance monitoring and tuning using intelligent agent assistants. In *Intl. Conf. on Information and Knowledge Engineering*, 1–5.

[6] EnterpriseDB. 2019. Enterprise Database. Retrieved from URL enterprisedb.com.

[7] Giardino, C.; Paternoster, N.; Unterkalmsteiner, M.; Gorschek, T.; and Abrahamsson, P. 2016. Software development in startup companies: the greenfield startup model. *IEEE Trans. on Software Engineering* 42:585–604.

[8] Kraska, T.; Beutel, A.; Chi, E. H.; Dean, J.; and Polyzotis, N. 2018. The case for learned index structures. In *Intl. Conf. on Management of Data*, 489–504. New York, USA: ACM.

[9] Licks, G. P.; Couto, J. C.; de Fátima Miehe, P.; De Paris, R.; Ruiz, D. D.; and Meneguzzi, F. 2020. SmartIX: A database indexing agent based on reinforcement learning. *Applied Intelligence* 50:2575–2588.

[10] Marcus, R., and Papaemmanouil, O. 2018. Deep reinforcement learning for join order enumeration. In *1st Intl. Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, 1–4. New York, USA: ACM.

[11] Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518:529–533.

[12] Neuhaus, P.; Couto, J.; Wehrmann, J.; Ruiz, D.; and Meneguzzi, F. 2019. GADIS: A genetic algorithm for database index selection. In *31st Intl. Conf. on Software Engineering and Knowledge Engineering*, 39–42. Pittsburgh, USA: KSI Research Inc.

[13] Olofson, C. W. 2018. Ensuring a fast, reliable, and secure database through automation: Oracle autonomous database. Technical report, IDC Corporate USA.

[14] Ortiz, J.; Balazinska, M.; Gehrke, J.; and Keerthi, S. S. 2018. Learning state representations for query optimization with deep reinforcement learning. *arXiv CoRR* abs/1803.08604:1–5.

[15] Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Kopf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; and Chintala, S. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc. 8026–8037.

[16] Pavlo, A.; Angulo, G.; Arulraj, J.; Lin, H.; Lin, J.; Ma, L.; Menon, P.; Mowry, T.; Perron, M.; Quah, I.; Santurkar, S.; Tomasic, A.; Toor, S.; Aken, D. V.; Wang, Z.; Wu, Y.; Xian, R.; and Zhang, T. 2017. Self-driving database management systems. In *Conf. on Innovative Data Systems Research*, 1–6. Chaminade, USA: CIDRDB.

[17] Pedrozo, W. G.; Nievola, J. C.; and Ribeiro, D. C. 2018. An adaptive approach for index tuning with learning classifier systems on hybrid storage environments. In *Intl. Conf. on Hybrid Artificial Intelligence Systems*, 716–729. Basel, Switzerland: Springer.

[18] Petraki, E.; Idreos, S.; and Manegold, S. 2015. Holistic indexing in main-memory column-stores. In *Intl. Conf. on Management of Data*, 1153–1166. New York, USA: ACM.

[19] Popovic, J. 2017. Automatic tuning – SQL Server. Retrieved from URL docs.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning.

[20] POWA. 2019. PostreSQL Workload Analyzer. Retrieved from URL powa.readthedocs.io.

[21] Ramakrishnan, R., and Gehrke, J. 2003. *Database Management Systems*. New York, USA: McGraw-Hill, 3 edition.

[22] Sharma, A.; Schuhknecht, F. M.; and Dittrich, J. 2018. The case for automatic database administration using deep reinforcement learning. *arXiv CoRR* abs/1801.05643:1–9.

[23] Sutton, R. S., and Barto, A. G. 2018. *Reinforcement learning: An introduction*. Cambridge, USA: MIT Press, 2 edition.

[24] Thanopoulou., A.; Carreira., P.; and Galhardas., H. 2012. Benchmarking with TPC-H on off-the-shelf hardware: An experiments report. In *14th Intl. Conf. on Enterprise Information Systems*, 205–208. Wroclaw, Poland: INSTICC.

[25] Tsitsiklis, J. N., and Van Roy, B. 1997. An analysis of temporal-difference learning with function approximation. *IEEE Trans. on Automatic Control* 42:674–690.

[26] Van Aken, D.; Pavlo, A.; Gordon, G. J.; and Zhang, B. 2017. Automatic database management system tuning through large-scale machine learning. In *Intl. Conf. on Management of Data*, 1009–1024. New York, USA: ACM.

[27] Wang, J.; Liu, W.; Kumar, S.; and Chang, S.-F. 2015. Learning to hash for indexing big data: a survey. *Proceedings of the IEEE* 104:34–57.