**Applied Intelligence manuscript No.**
(will be inserted by the editor)

# SMARTIX: A database indexing agent based on reinforcement learning

**Gabriel Paludo Licks** · **Julia Colleoni Couto** ·
**Priscilla de Fátima Miehe** · **Renata De Paris** ·
**Duncan Dubugras Ruiz** · **Felipe Meneguzzi**

Received: date / Accepted: date

**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. In this paper, we develop the SMARTIX 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. We train and evaluate SMARTIX performance using TPC-H, a standard, and scalable database benchmark. Our empirical evaluation shows that SMARTIX converges to indexing configurations with superior performance compared to standard baselines we define and other reinforcement learning methods used in related work.

**Keywords** artificial intelligence · reinforcement learning · database · indexing

## 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 [31]. Slow information retrieval in databases entails not only wasted time for a business but also indicates a high computational cost being paid. Unnecessary indexes 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 [5, 6]. 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].

One of the options to improve database query performance is by creating indexes [21]. Indexes are usually created by Database Administrators (DBAs) either proactively during

G. Licks
School of Technology - PUCRS
Porto Alegre, RS - Brazil
E-mail: gabriel.licks@edu.pucrs.br

the schema implementation, or reactively according to the response time of the most executed queries. Most recent versions of Database Management Systems (DBMS), such as Oracle [14] and Azure SQL Database [19], can automatically adjust indexes by itself. However, the usual scenario is that the DBMS just helps to decide when and where to create an index by offering recommendations and statistics for optimizing queries, being the final decision taken by the DBA.

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). Our architecture allows continuous assessment and automatic modification of the database index configuration, according to the performance tests. We modeled the state of the environment as the set of currently indexed columns and measure agent performance in this environment through a standardized set of queries, insertions and deletions, which we refer to as the database workload. The resulting agent then explores this environment to find the optimal set of indexes concerning the current workload.

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

In this section, we review the reinforcement learning concepts and methods we use to model our agent for performing indexing in relational databases, followed by an overview of database indexing and the database benchmark we used to measure agent performance.

### 2.1 Reinforcement Learning

Reinforcement learning is the closest machine learning paradigm to how humans learn, with its algorithms strongly inspired by biological aspects [23, Ch. 1, p. 4]. It is characterized by a trial-and-error learning method, where an agent interacts and transitions through states of an environment by taking actions and observing rewards [23, Ch. 1, p. 1-2]. The objective of a reinforcement learning agent is to maximize its accumulated reward in the environment the agent is acting on, ultimately leading a policy specifying which actions maximize the utility in each state.

Reinforcement learning is an approach to learn optimal agent policies in stochastic environments modeled as Markov Decision Processes (MDPs) [3]. It is characterized by 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, p. 1-2]. These MDP environment models can be formally defined as a tuple $\mathscr{M} = \langle \mathscr{S}, \mathscr{A}, \mathscr{P}, \mathscr{R}, \gamma \rangle$, where $\mathscr{S}$ is the state space, $\mathscr{A}$ is the action space, $\mathscr{P}$ is a transition probability function which defines the dynamics of the MDP, $\mathscr{R}$ is a reward function and $\gamma \in [0,1]$ is a discount factor [23, Chapter 3].

The way agents an agent behaves in an MDP environment is by following a policy, which is a function that maps states to actions [23, Ch. 1, p. 6]. After taking an action at a given state, an immediate reward is given to the agent as feedback from the environment [23, Ch. 1, p. 6]. The solution to an MDP, thus, is a policy $\pi$ that maximizes the reward an agent receives over the long run. The expected reward is estimated by a state-action value function $Q_\pi$ that, for each state-action pair, returns a value computed based on the amount of reward an agent might expect in the long run by taking a particular action on that state. The value function for a state-action pair, following a policy $\pi$, is computed using the Bellman Expectation Equation [23, Ch. 3, p. 62]:

$$Q_\pi(s,a) = \mathbb{E}_\pi[r_{t+1} + \gamma q_\pi(s_{t+1}, a') \mid s, a], \tag{1}$$

where expectation operator $\mathbb{E}$ determines that the value of a given state-action pair is an average value of what is expected from its successor states in the long run. Suppose there are two policies $\pi$ and $\pi'$, $\pi$ is better than $\pi'$ if $Q_\pi(s,a) \geqslant Q_{\pi'}(s,a)$, where $Q_\pi(s,a)$ is the utility of a state-action pair estimated by the value function under a policy $\pi$ [23, Ch. 3, p. 62]. The optimal solution, thus, is a policy $\pi_*$ that is better than or equal to all other policies [23, Ch. 3, p. 62].

In order to solve an MDP, however, an agent needs to know the state-transition and the reward functions. 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 [24]. The Q-learning algorithm is the natural choice for solving such MDPs [25]. This method learns the values of state-action pairs, denoted by $Q(s,a)$, representing the value of taking action $a$ in a state $s$ [26]. The basic idea of the algorithm is to use the update function of Equation 2 to incrementally estimate values of $Q(s,a)$ based on reward signals from each action taken.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma max_{\mathrm{a}'} Q(s',a') - Q(s,a)) \tag{2}$$

The most naïve implementation of Q-learning relies on storing state-action values using tabular structures. The problem with tabular structures is that, when we have a large state space and branching factor, it is infeasible to visit all state-action pairs enough times that the estimates of their values are close enough to the true value to be able to compute an optimal policy [23, Chapter 9, pp 196-197]. 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 [29]. 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].

We approximate state-action values using Equation 3

$$\hat{Q}(s,a) \leftarrow \theta_0 + \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s) \tag{3}$$

where $\theta \in \Theta$ is a parameter associated to a state feature $f \in F$. We adjust its parameters through agent experience to approximate the true state-action value function by employing gradient descent in the error with regard to each feature parameter:

$$\theta_i \leftarrow \theta_i + \alpha[r(s) + \gamma max_{a'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)] \frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i} \tag{4}$$

Using Equation 4, we adjust our $\Theta$ parameters in the direction of decreasing the error after each trial [29]. Function approximation allows us to estimate the value function of new state-action pairs by generalizing from known state-action pairs. This means that we can predict state-action values by learning and updating $\Theta$ parameters throughout algorithm iterations.

However, these approximations can produce higher variance updates in the value function, which can result in steps that greatly vary in size and lead to parts of the space with a different gradient [23, Ch. 11, p. 283]. In order to avoid oscillations in the parameters, literature employs a method called experience replay [12]. This method stores agent's experience tuples $e = \langle s,a,r,s' \rangle$ at each time-step in a replay memory $D = \{e_1,\ldots,e_n\}$. At each time step, multiple updates are performed based on a mini-batch of experiences, $e \sim D$, sampled uniformly at random from the replay memory [23, Ch. 16, p. 440]. The aim is to reduce the variance of updates as successive updates are not correlated with one another as they would be with standard Q-learning [12].

## 2.2 Indexing in Relational Databases

A DBMS is a software designed to manage databases and facilitate organizing collections of data efficiently [21, Ch. 1, p. 4]. In particular, we address relational DBMSs, which are based on the relational model [21, Ch. 1, p. 10], where data collections can be thought as tables whose rows represent records and columns represent attributes [21, Ch. 3, p. 60]. The way a DBMS stores data internally is through *files*, each of which consists of pages [21, Ch. 8, p. 273]. However, it is not trivial to maintain these records organized in order to facilitate data retrieval. For example, maintaining a set of numeric records sorted can be a good strategy for later retrieval, though it becomes computationally expensive when records are constantly modified [21, Ch. 8, p. 274].

An important technique to file organization in a DBMS is *indexing* [21, Ch. 8, p. 274]. Indexes are data structures that optimize retrieval operations with regard to a search key. Suppose there is a set of records containing attributes *age* and *salary*, such that these are sorted according to the *age* attribute. This organization facilitates the searching for records using the key *age*, but searching for salary can be computationally expensive, even requiring a complete sweep of the records in the worst case. If an index with the *salary* key was available, searches involving *salary* could be significantly improved [21, Ch. 8, p. 276].

The way indexes are organized depend on the data structure it uses. The two main techniques to maintain data indexed are hash-based and tree-based [21, Ch. 8, p. 278] structures. The former technique organizes records by hashing records according to a search key, and these are grouped into buckets according to a hash function [21, Ch. 8, p. 279]. The latter technique organizes records using a tree-like structure, which arranges records in a sorted order and directs the search trough intermediate tree levels until leaves containing data entries are reached [21, Ch. 8, p. 280].

The type of an index depends on the attributes it comprises. Indexes can be created both on attributes that are primary keys (a record's unique identifier) or secondary keys (non-unique attribute values), respectively called primary indexes and secondary indexes [21, Ch. 8, p. 277]. The difference is that a primary index is guaranteed to be unique, while secondary indexes can contain duplicates, which means that search keys on secondary indexes can lead to more than one record [21, Ch. 8, p. 278]. Indexes can also be composite when one is created to comprise more than one attribute. Composite indexes can be beneficial when the search key includes conditions on more than one attribute, thus supporting a broader range of queries [21, Ch. 8, p. 296].

Indexing is a task usually undertaken by the Database Administrator (DBA), a person who has considerable domain knowledge in order to make such decisions [21, Ch. 8, p. 291]. Although indexes are helpful in improving query performance, creating too many indexes will slow down INSERT, UPDATE, and DELETE operations. This is due to the fact that, whenever one of these operations affects a record, the whole collection of records and indexes have to be updated in order to match the organization being maintained, which implies in a computational overhead [21, Ch. 8, p. 290-291]. Consequently, there is a trade-off in the number of indexes one might want to have and the computational overhead one is willing to pay [21, Ch. 20, p. 654]. Thus, the DBA has to balance this trade-off to achieve the best performance.

Techniques for index selection without the need of a domain expert is a long-time research subject and remains a challenge due to the problem complexity [31, 6, 5]. The idea is that, given the database schema and the workload it receives, we can define the problem of finding a proper index configuration that optimizes database operations [21, Ch. 20, p. 664]. The complexity of this task comes from the potential number of attributes that can be indexed and all of its subsets. Suppose we have $n$ attributes that compose our records, let us calculate the number of different indexes we can create. We have $n$ choices of attributes for the first index, $n-1$ for the second, such that for an index with up to $c$ attributes we have

$$\sum_{i=1}^{c} \frac{n!}{(n-i)!} \tag{5}$$

possibilities in total. That is, for collections of records with 10 attributes, there are 10 different possibilities of 1-attribute indexes, 90 different possibilities of 2-attribute indexes, and 30240 different possibilities of 5-attribute indexes [21, Ch. 20, p. 654].

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 [14] and Azure SQL Database [19] can automatically adjust indexes. However, it is not the case that the underlying system is openly described. The former does not explain the strategy and techniques used to accomplish it. The latter goes slightly into more detail by briefly describing the actions it performs: it identifies indexes that could improve the performance of queries that read data from the tables; and identifies the redundant indexes or indexes that were not used in a longer period of time that could be removed [19].

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

TPC[1] stands for Transaction Processing Performance Council, a well-known non-profit corporation that produces benchmarks to measure database performance [28]. The identifier "H" represents one of its decision support benchmark versions. The TPC-H is a good proxy for querying tasks because it has business-oriented ad-hoc queries that can scale to large volumes of data. Its relational model is composed of 8 tables, briefly described as follows:

- REGION: contains the continents of the world.
- NATION: contains a list with some countries of the world.
- CUSTOMER: a person who buys parts from suppliers.
- SUPPLIER: an organization that provides parts.
- PART: pieces made available by suppliers.
- PARTSUPP: the relationship between suppliers and parts.
- ORDERS: data related to purchase orders.
- LINEITEM: the biggest table in the dataset. It contains details of all orders of each customer, with a list of their parts.

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. Based on the benchmark, we gather outputs from three metrics, named $QphH@Size$, $Power@Size$, and $Throughput@Size$. The resulting values are related to its scale factor ($@Size$), i.e., the database size in gigabytes.

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 [27]. The $Power@Size$ evaluates how fast the DBMS computes the answers to single queries. It is composed of: (1) the first Refresh Function (RF1) that inserts into tables ORDERS and LINEITEM a set of 0.1% of records based on the initial population of these two tables; (2) a single query stream composed of 22 queries generated by QGen; (3) the second Refresh Function (RF2), that drops the same percentage of rows as the RF1. This metric is computed using the formula in Equation 6:

$$Power@Size = \frac{3600}{\sqrt[24]{\pi_{i=1}^{22} QI(i,0) \times \pi_{j=1}^{2} RI(j,0)}} \times SF \tag{6}$$

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 the refresh functions $j$ in the query stream $s$, and $SF$ is the scale factor or database size, which may range from 1 to $100,000$ according to its $@Size$. As the $Power@Size$ metric is based on the geometric mean, the root of the product is the overall execution time from one stream (22 queries) and the two RFs.

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 [27]. It computes the performance of the system against a multi-user workload performed in an elapsed time, using the formula in Equation 7:

$$Throughput@Size = \frac{S \times 22}{T_S} \times 3600 \times SF \tag{7}$$

[1] TPC: `http://www.tpc.org/`

where $S$ is the number of query streams executed, and $T_S$ is the total time required to run the throughput test for $s$ streams. Equation 8 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.

$$QphH@Size = \sqrt{Power@Size \times Throughput@Size} \tag{8}$$

Since the $QphH@Size$ metrics summarizes both single-user and overall database performance, we consider this to be an informative reward signal for our agent, as we see in Section 4.

## 3 SMARTIX

In this section, we introduce SMARTIX, a reinforcement learning agent to automatically choose indexes in relational databases. 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. Reinforcement learning is exactly the technique that solves the problem of finding an optimal strategy over a long time horizon while improving the performance of an agent in the environment. To evaluate the SMARTIX agent, we use the database provided by TCP-H and its benchmarking protocol in order to evaluate the index configurations explored by the agent.

The agent works based on five components, which are illustrated in Figure 1. The Database State Representation component converts the information about the existing indexes in the database into an agent state. The Performance Benchmarking component computes the performance metrics (e.g., $QphH@Size$) from the current database index configuration, which we use as a reward signal. The Learning Algorithm predicts the state-action values using function approximation and updates its parameters based on the algorithms from Section 2.1. The Exploration Function is responsible for choosing an action to be executed in the database, either to exploit the learned information or to explore by choosing sub-optimal actions. Finally, the Action Execution component sends to the database an indexing option according to the previously chosen action.

### 3.1 Problem Formalization

We start analyzing the problem by verifying the number of columns available to index in the database. For that, we map for each table in the TPC-H database the total number of columns it contains, separated by the columns that are indexed by default (primary and foreign keys, which are not modified by the agent), and the remaining columns that are available for indexing. These numbers are shown in Table 1 and, by summing the number of indexable columns in each table, we have a total of 45 columns that are available for indexing.

Considering that a column is either indexed or not, there are two possibilities for each of the 45 columns. This scenario indicates that we have exactly 35,184,372,088,832 ($2^{45}$) possible index configurations. We can think of it as a matrix of 45 columns by over 35 trillion lines containing all possible combinations. Thus, this is the number of index configurations that can be assumed by the database, and therefore, the number of states in which the agent can explore.

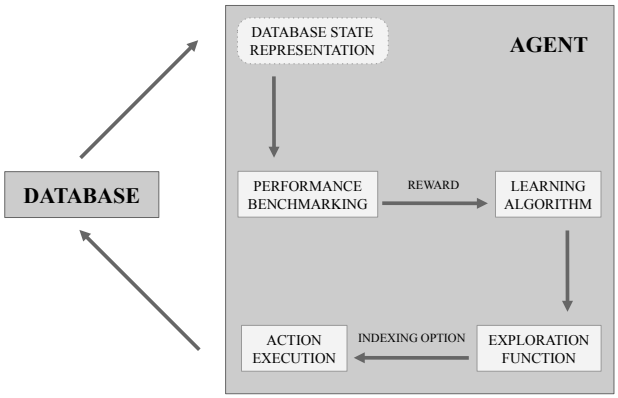



Fig. 1: Conceptual architecture of SMARTIX

Table 1: Amount of indexes per table in the TPC-H database.

| Table | Total Columns | Indexed Columns | Indexable Columns |
|---|---|---|---|
| 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 |

### 3.2 States and Actions

The state is the formal representation of the environment information used by the agent in the learning process. Thus, to decide which information should be used to define a state of the environment is critical for the 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 does not learn a useful policy. In contrast, if the state encodes too much information, there is a risk that the learning algorithm needs so many samples of the environment that it does not converge to a policy.

We represent our states as a finite set $S = C_1, \cdots, C_{45}$, where $C_i$ is one of the 45 available indexable columns of the database. We define $C_i$ as a tuple of $\langle Col_i, Bit_t \rangle$, where $Col_i$ is the column identifier and $Bit_t$ is the column information at time $t$, containing 0 or 1 according to whether each column is indexed or not. Consequently, the state space of our problem is the power set $\mathscr{P}(C_i)$ of the set of columns in a database.

We structure our state information using a binary vector to represent all indexable table columns in the database. Figure 2 illustrates a snippet of this binary vector, where we used columns of the LINEITEM table as a sample and an example of the available actions in a state. There are always two possible actions: "CREATE INDEX" or "DROP INDEX". Considering that there are 45 columns in a state that can either be dropped or created an index, the agent has 45 other different states in which it can transition to, depending on the chosen

action. The indexes created are $B^+$ tree structures, since the select queries in the workload are more efficient when we use an ordered structure type.

### 3.3 Reward Function

Deciding the reward function is critical for the quality of the ensuing learned policy. To have a feedback on whether a given index configuration has increased or decreased the database performance, we employed the TPC-H *QphH@Size* metric in our reward function. Thus, whenever the agent transitions to a new state, the benchmark is executed to provide a scalar reward. The highest performance metric rewarded is used to evaluate whether the agent found an optimizing index configuration among the explored ones.

### 3.4 Agent Training

We organize the agent training in episodes, each of which is composed of 100 steps in the environment. Each step consists of the agent choosing an action, executing it in the environment, and then letting the DBMS compute the reward using the benchmark we specified in Section 2.3. Specifically, at any given step, the agent follows its policy and executes an action of either creating or dropping an index in a given column. This index is created or dropped, and the benchmark returns the resulting performance metric in order to evaluate how good that action is in such state.

We employed an $\varepsilon$-greedy policy for agent exploration. It means that, with $\varepsilon$-probability, the agent chooses whether to take a $random\ action_a$ or an $arg\,max_a$ among the available actions in a given state. At the end of each episode, we decayed our epsilon value by 25% of its value, so that the agent initially explores different states with high probability and starts stabilizing later on. Since our environment does not have a terminal state, the agent explores the state-space until the last step within an episode. To avoid local minima, we drop all indexes at the end of each episode so that the agent can restart the exploration of indexes from the default state.

Algorithm 1 depicts the steps performed by the SMARTIX agent. The algorithm consists of a reinforcement learning approach built around the Q-learning method, using linear function approximation and experience replay. In the following section, we describe the algorithm's performance and training statistics with regard to the methods applied.

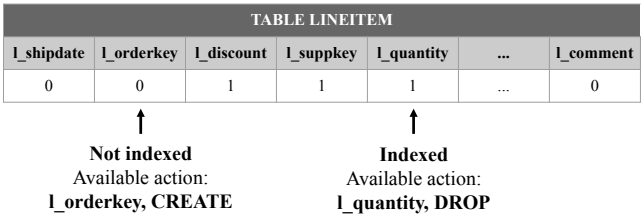



Fig. 2: State representation and available actions

**Algorithm 1** Q-learning indexing agent with function approximation and experience replay. Adapted from [26] and [12].

Random initialization of parameters $\Theta$
Empty initialization of replay memory $D$
**for** each episode **do**
  $s \leftarrow$ *DB initial index configuration mapped as initial state*
  **for** each step of episode **do**
    $a \leftarrow \begin{cases} \arg\max_{a \in A} Q(s,a) & \textit{with probability } 1-\varepsilon \\ \textit{random}_{a \in A} & \textit{with probability } \varepsilon \end{cases}$
    $s', r \leftarrow execute(a)$
    **for** $\theta_i \in \Theta$ **do**
      Update $\theta_i$ according to Equation 4
    **end for**
    Store experience $e = \langle s, a, r, s' \rangle$ in $D$
    Sample random mini-batch of experiences $e \sim D$
    Perform experience replay using sampled data
    $s \leftarrow s'$
  **end for**
**end for**

## 4 Experimental Evaluation

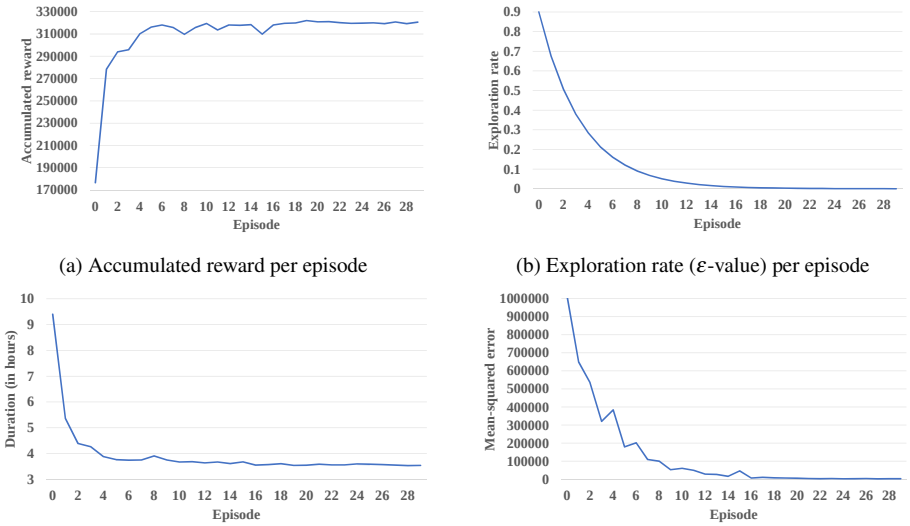
In this section, we report the experiments performed using SMARTIX to optimize the index configuration of the TPC-H database. We evaluate the index configuration in which our algorithm converged to in comparison to other baseline and related work methods that use reinforcement learning and genetic algorithms. We evaluate index configurations using the query-per-hour performance metric (Equation 8) provided by the TPC-H benchmark and analyze the disk space occupied by indexes.

We carried out all experiments in a 4-core Intel Core i5-4590 CPU @ 3.30GHz and 8 GB of RAM running Ubuntu Linux 18.04 and Python version 3.6.6. We implemented the TPC-H benchmark protocol in a Python script to run the queries in the database and to compute performance metrics for each configuration. For each execution of the TPC-H benchmark, we first calculate the Power (Equation 6) and Throughput (Equation 7) metrics in a scale factor of 1 and using the response time of the 22 benchmark queries in MySQL, according to the execution rules from the TPC-H benchmark protocol [28]. Then, we calculated the QphH (Equation 8) for each configuration.

### 4.1 Using SmartIX for Automated Indexing

We trained the SMARTIX agent throughout 30 episodes, composed of 100 steps each, which took approximately 4.9 days of training. At each step, the agent executed the TPC-H benchmark to retrieve the reward metric, which took approximately 3-4 minutes per execution. From the 3000 states visited by the agent, only 1116 of them are distinct states, and the remaining are states that were visited more than once. As a result of the $\varepsilon$-greedy exploration function, out of the 3000 actions taken by the agent during training, 2657 of were *arg max* actions, and the remaining 343 were random actions. The entire description and source-code for replicating the experiments here described are available at Zenodo [10].

Statistics from the training episodes are in Figure 3. We show in Sub-Figure 3a the total reward accumulated by the agent per episode. Note that, the accumulated sum of rewards




(a) Accumulated reward per episode

(b) Exploration rate ($\varepsilon$-value) per episode

(c) Duration of each episode

(d) Mean-squared error per episode

Fig. 3: Agent training statistics

stabilizes after the 16th episode, which indicates that the agent is converging to a stable index configuration. Sub-Figure 3b shows the corresponding $\varepsilon$-value for each episode during the learning, decaying the exploration rate. Sub-Figure 3c shows that the time to train each episode decreases as the agent optimizes the index configuration, as better index configurations require less time to run the queries in the benchmark at each step of an episode. Sub-Figure 3d illustrates the errors in predictions of state-action values and how it decreases towards zero as function parameters are adjusted, and the agent approximates the true value function.

### 4.2 Baseline Indexing Configurations

This subsection describes the baseline database indexing approaches that we use to compare the performance with our SMARTIX reinforcement learning agent. These baselines comprise different indexing configurations that we obtained using different indexing approaches, including commercial and open-source database advisors, genetic algorithms, and reinforcement learning. Appendix A lists the indexed columns for each configuration of the following comparable approaches:

1. **Initial state:** it is the default TPC-H configuration and contains indexes on primary and foreign keys only.
2. **Expert-based:** it is based on an analysis of the database workload, contains the indexes from the initial state in addition to indexes that decrease queries execution cost, the manner a DBA would do.
3. **All indexed:** it is the configuration that contains all columns indexed.
4. **Random policy:** it is the highest performing configuration explored by an agent following a policy that selects indexing options randomly over the course of 1000 iterations

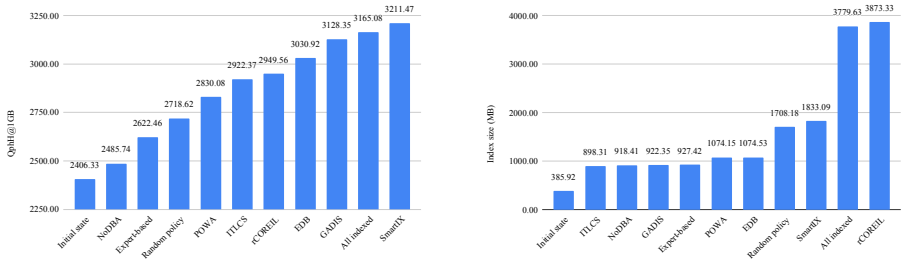
5. **EDB** [7]: it is the index configuration suggested by EnterpriseDB, a commercial database advisor tool.
6. **POWA** [20]: it is the index configuration suggested by the PostgreSQL Workload Analyser, an open-source advisor tool.
7. **ITLCS** [17]: the Index Tuning with Learning Classifier System (ITLCS), which combines a learning classifier and genetic algorithms to discover rules for efficient indexing.
8. **GADIS** [13]: the Genetic Algorithm for Database Index Selection (GADIS), which uses a genetic algorithm to explore index configurations encoded as individuals.
9. **NoDBA** [22]: a system based on a cross-entropy deep reinforcement learning method applied to recommend indexes for given workloads.
10. **rCOREIL** [2]: a system based on a policy iteration reinforcement learning method, which estimates a database cost model and suggests indexes that decrease such cost.

The expert-based index configuration is a result of a traditional DBA-style analysis, where we examined the database schema and the queries performed in the workload. We analyzed the columns contained in each "WHERE" clause of each of the 22 queries. Then, we employed the query EXPLAIN command to visualize the cost of each query step, which allows us to identify indexing opportunities for columns that impact on higher costs. We created candidate indexes and ran the EXPLAIN plan again to evaluate whether the queries cost actually decreased. The expert-based index configuration is constituted of all the candidate indexes that resulted in a reduction in the queries execution cost.

The EDB [7], POWA [20] and ITLCS [17] 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 [13], NoDBA [22] and rCOREIL [2] 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.

Table 2: TPC-H metrics for each index configuration. The highest performance for each metric is highlight in bold.

| **Index config.** | **Power@1GB** | **Throughput@1GB** | **QphH@1GB** | **Index size (MB)** |
|---|---|---|---|---|
| Initial state | 3250.24 | 1781.91 | 2406.33 | 385.92 |
| Expert-based | 3581.70 | 1920.37 | 2622.46 | 927.42 |
| All indexed | 4016.19 | 2494.37 | 3165.08 | 3779.63 |
| Random policy | 3793.92 | 1948.09 | 2718.62 | 1708.18 |
| EDB [7] | 3767.30 | 2438.52 | 3030.92 | 1074.53 |
| POWA [20] | 3558.02 | 2251.09 | 2830.08 | 1074.15 |
| ITLCS [17] | 3701.13 | 2307.86 | 2922.37 | 898.31 |
| GADIS [13] | 3936.64 | 2486.06 | 3128.35 | 922.35 |
| NoDBA [22] | 3333.66 | 1854.29 | 2485.74 | **918.41** |
| rCOREIL [2] | 3799.52 | 2290.75 | 2949.56 | 3873.33 |
| **SmartIX** | **4081.65** | **2526.84** | **3211.47** | 1833.09 |




(a) Query-per-hour metric comparison.

(b) Disk space occupied by indexes.

Fig. 4: Query-per-hour metric and index size of each configuration (ascending order)

### 4.3 Results and Discussion

We compute the TPC-H performance metrics for each baseline configuration and compare them to the one in which the SMARTIX agent converged to in the last episode of training. Table 2 shows the performance metrics for each configuration, where higher values denote better performance for each metric. These performance metrics are a result of a trimmed mean, where we run the TPC-H benchmark 12 times for each index configuration, removing the highest and the lowest result and averaging the 10 remaining results. The last column in Table 2 shows the storage space required for the indexes in each configuration, which allows us to analyze the trade-off in the number of indexes and the resources needed to maintain it. To better visualize such information, we plot the query-per-hour metric and the index size for each configuration in ascending order in Figure 4. We achieve the highest query-per-hour metric in comparison to the baseline configurations and related work.

Indexing all columns yields the second highest QphH@1GB and can seem to be a natural alternative to solve the indexing problem. However, all columns indexed results in the second 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].

While rCOREIL [2] is the most competitive reinforcement learning method in comparison to SMARTIX, the amount of storage used to maintain its indexes is the highest. rCOREIL does not handle whether primary and foreign key indexes are already created, causing it to create duplicate indexes. It is the only algorithm among the baselines to work with composite indexes. Composite indexes can benefit a broader range of queries [21, Ch. 8, p. 296], but are structures that occupy a higher amount of disk space, and incur a higher overhead to maintain. The algorithm converges after 169 iterations, after which it makes no changes to the index configuration. 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 databases that contain a larger number of columns to index.

The most competitive related work using genetic algorithms is GADIS [13]. The algorithm uses a similar state-space model to SMARTIX, with individuals being represented as binary vectors of the indexable columns, and the fitness function being the query-per-hour

metric just as the reward function used by SMARTIX. At each individual evaluation, one benchmark execution is needed to compute the performance metric to the configuration, as in SMARTIX when a new state is reached. The genetic algorithm, however, took 50 generations of 100 individuals to converge to the index configuration. The configuration occupies approximately 50.32% less disk space than SMARTIX, but has a lower performance metric.

Among the database advisors, the commercial tool EDB [7] achieves the highest query-per-hour metric in comparison to the open-source tool POWA [20], while its indexes occupy virtually the same disk space. Other baselines and related work are able to optimize the index configuration and have lightweight index sizes, but are not competitive in comparison to the previously discussed methods in terms of the query-per-hour performance metric. Finally, the SMARTIX agent converges to an index configuration that has the highest query-per-hour metric while trading-off storage space for indexes (approximately 51.5% less space than simply indexing all columns). Our agent learns the value of each indexing option and stabilizes in configurations with a performance metric that is on average higher than 3100 QphH@1GB after the 4th episode, which already surpasses most of the index configurations we evaluate in this analysis.

## 5 Related Work

Machine learning techniques are used in a variety of tasks related to database management systems and automated database administration [30]. One example is the work from Kraska et al. [9], 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] research culminated on developing the first self-driving DBMS, called Peloton, which has autonomic capabilities of optimizing the system to incoming workload and also uses predictions to prepare the system to future workloads using predictions. In this chapter, 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. [2] developed a technique for index tuning based on a cost model that is learned with reinforcement learning. It is stated that current DBMS's cost estimates can be highly erroneous; thus, the authors propose a tuning strategy without a predefined model. They learn a cost model through linear regression and approximate the cost of executing queries at the current configuration, and instantiate their approach to the case of index tuning, to find a indexing configuration that minimizes the cost function. 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 [4] to reduce the state space by considering only indexes that are recommended by the DBMS. Our approach, on the other hand, is focused 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] explore the use of a cross-entropy deep reinforcement learning method to administer databases automatically. Their motivation is that DBMSs currently have a large number of configuration settings that can be set, and it is typically up to a human administrator to tweak it based on its own experience [22]. They instantiate their approach to index tuning by evaluating how well their system selects indexes to a given workload. 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. Their evaluation relies on the TPC-H relational model as the database [28], just as our approach. However, they do not strictly follow the TPC-H protocol, as they do not use the query workload provided by TPC-H, but use synthetic queries consisting of select counts on the LINEITEM table, which does not consider INSERT or DELETE operations (highly affected by the presence of indexes). 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 in milliseconds.

Unlike previous papers, Pavlo et al. [16] present an entire self-driving *in-memory* DBMS, called Peloton, that predicts the expected arrival rate of queries and deploys physical, data and runtime actions. Their approach uses clustering algorithms to classify queries and recurrent neural networks to generate models that predict the arrival rate of queries from an expected workload in order to plan and execute actions. They specify actions regarding creating and dropping indexes, though there are no detailed results on the system's approach to indexing, making it difficult to make an actual comparison to what we propose here. Preliminary results of the proposed architecture, however, rely on analyzing the accuracy of the workload predicted by their models in comparison to the actual workload sent to the *in-memory* DBMS. Nevertheless, it is a strongly related work in terms of what we are working.

Other papers show that reinforcement learning can also be explored in the context of query optimization by predicting query plans: Marcus et al. [11] proposed a proof-of-concept to determine the join ordering for a fixed database; Ortiz et al. [15] 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.

## 6 Conclusion

In this paper, we developed the SMARTIX agent architecture for automated database indexing using reinforcement learning. We implemented the TPC-H protocol to benchmark and give feedback on the agent actions, aiming to optimize the index configuration of a relational database. The experimental results show that our agent consistently outperforms the baseline index configurations and related work on genetic algorithms and reinforcement learning. Our agent also proved itself to find a trade-off concerning the disk space its index configuration uses and the performance metric it achieves.

Our architecture can also be adapted to be used in other systems. We can develop the agent to listen to queries that are being received in the database and measure their response time, modeling a reward function similarly to TPC-H's benchmark. In a real application, the agent is expected to manipulate these indexes automatically, without any direct human supervision. Therefore, for future work, we plan to:

1. evaluate all possible combinations among tables working with composite indexes;
2. analyze queries to predict candidate columns to be indexed for a dynamic workload;
3. expand the experimental setting for different reinforcement learning strategies, such as auto-encoded state compression [1]; and
4. evaluate SMARTIX on big data ecosystems (e.g., Hadoop).

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 [8]. 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 lack 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 expert indexing recommendations.

**Acknowledgements** 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.

## Conflict of interest

The authors declare that they have no conflict of interest.

## A Index configurations

The corresponding indexed columns in each of the configurations we use in our experiments are shown in Table 3.

Table 3: Indexing configuration for all the baselines and SmartIX

| **Index config.** | **Indexed columns** |
|---|---|
| Initial state | Only primary and foreign keys are indexed. |
| Expert-based | **LINEITEM**: l_shipdate; **ORDERS**: o_orderdate; **PART**: p_brand, p_container, p_name, p_size, p_type. |
| All indexed | All columns are indexed. |
| Random policy | **LINEITEM**: l_shipdate, l_extendedprice, l_returnflag, l_commitdate, l_shipmode; **ORDERS**: o_orderdate, o_orderstatus, o_clerk, o_comment; **PARTSUPP**: ps_supplycost, ps_comment; **CUSTOMER**: c_mktsegment, c_phone; **SUPPLIER**: s_address, s_comment, s_phone; **PART**: p_mfgr, p_comment, p_retailprice, p_brand, p_size; **NATION**: n_comment. |
| EDB [7] | **CUSTOMER**: c_mktsegment; **ORDERS**: o_orderdate; **PART**: p_type, p_name; **LINEITEM**: l_returnflag, l_shipdate; **SUPPLIER**: s_name. |
| POWA [20] | **CUSTOMER**: c_mktsegment; **LINEITEM**: l_shipdate, l_returnflag; **ORDERS**: o_orderdate; **PART**: p_type, p_name. |
| ITLCS [17] | **LINEITEM**: l_shipdate; **PART**: p_type; **ORDERS**: o_orderdate. |
| GADIS [13] | **CUSTOMER**: c_phone; **LINEITEM**: l_shipdate; **ORDERS**: o_orderdate; **PART**: p_container; **PARTSUPP**: ps_availqty. |
| NoDBA [22] | **LINEITEM**: l_discount, l_quantity, l_orderkey. |
| rCOREIL [2] | **NATION**: n_nationkey, [n_nationkey, n_name], [n_name, n_nationkey], [n_regionkey, n_nationkey], [n_regionkey, n_name] **REGION**: [r_name, r_regionkey]; **ORDERS**: o_custkey, [o_orderkey, o_custkey], [o_orderkey, o_totalprice], [o_orderstatus, o_orderkey], [o_orderdate, o_orderkey], [o_orderdate, o_custkey], [o_orderdate, o_orderpriority], [o_orderdate, o_shippriority]; **CUSTOMER**: [c_acctbal, c_custkey], [c_mktsegment, c_custkey], [c_custkey, c_name], [c_custkey, c_nationkey] **PART**: [p_name, p_partkey], [p_brand, p_size], [p_type, p_partkey], [p_size, p_partkey], [p_size, p_brand], [p_size, p_type], [p_container, p_brand] **PARTSUPP**: [ps_partkey, ps_suppkey], [ps_suppkey, ps_partkey]; **SUPPLIER**: [s_suppkey, s_name], [s_nationkey, s_suppkey], [s_nationkey, s_name]; **LINEITEM**: [l_orderkey, l_suppkey], [l_orderkey, l_linenumber], [l_partkey, l_suppkey], [l_partkey, l_extendedprice], [l_partkey, l_discount], [l_suppkey, l_linenumber], [l_suppkey, l_discount], [l_suppkey, l_shipdate], [l_shipdate, l_discount], [l_shipdate, l_tax], [l_discount, l_shipdate], [l_returnflag, l_orderkey], [l_shipmode, l_receiptdate], [l_shipmode, l_shipinstruct]. |
| SMARTIX | **CUSTOMER**: c_name, c_address, c_phone, c_mktsegment; **LINEITEM**: l_discount, l_tax, l_shipdate, l_commitdate; **NATION**: n_comment; **ORDERS**: o_orderstatus, o_totalprice, o_orderdate, o_clerk, o_shippriority; **PART**: p_brand, p_type, p_size, p_container, p_comment; **PARTSUPP**: ps_availqty, ps_supplycost, ps_comment; **REGION**: r_comment; **SUPPLIER**: s_name, s_acctbal. |

## References


1. Amado, L., Meneguzzi, F.: Reinforcement learning applied to RTS games. In: Proceedings of the 2017 Workshop on Adaptive Learning Agents (ALA), pp. 1–8. ALA 2017, São Paulo. Brazil (2017)
2. Basu, D., Lin, Q., Chen, W., Vo, H.T., Yuan, Z., Senellart, P., Bressan, S.: Regularized cost-model oblivious database tuning with reinforcement learning. In: Transactions on Large-Scale Data-and Knowledge-Centered Systems XXVIII, pp. 96–132. Springer, Springer, Berlin (2016)
3. Bellman, R.: An introduction to artificial intelligence: Can computers think? Boyd & Fraser Pub. Co, The University of Michigan (1978)
4. DB Group at UCSC: DBTune (2019). URL `https://github.com/dbgroup-at-ucsc/dbtune`
5. Duan, S., Thummala, V., Babu, S.: Tuning database configuration parameters with ituned. Proceedings of the VLDB Endowment **2**(1), 1246–1257 (2009)
6. Elfayoumy, S., Patel, J.: Database performance monitoring and tuning using intelligent agent assistants. In: Proceedings of the International Conference on Information and Knowledge Engineering (IKE), p. 1. The Steering Committee of The World Congress in Computer Science, Computer ... (2012)
7. EnterpriseDB: Enterprise Database (2019). URL `https://www.enterprisedb.com`
8. Giardino, C., Paternoster, N., Unterkalmsteiner, M., Gorschek, T., Abrahamsson, P.: Software development in startup companies: the greenfield startup model. IEEE Transactions on Software Engineering **42**(6), 585–604 (2016)
9. Kraska, T., Beutel, A., Chi, E.H., Dean, J., Polyzotis, N.: The case for learned index structures. In: Proceedings of the 2018 International Conference on Management of Data, pp. 489–504. ACM (2018)
10. Licks, G.P., Couto, J.C., Meneguzzi, F., Ruiz, D.D., de Fátima Miehe, P., de Paris, R.: mir-pucrs/smartix-rl v1.0 (2019). DOI 10.5281/zenodo.3254967. URL `https://doi.org/10.5281/zenodo.3254967`
11. Marcus, R., Papaemmanouil, O.: Deep reinforcement learning for join order enumeration. CoRR **abs/1803.00055**, 1–7 (2018). URL `http://arxiv.org/abs/1803.00055`
12. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.A.: Playing atari with deep reinforcement learning. CoRR **abs/1312.5602**, 9 (2013). URL `http://arxiv.org/abs/1312.5602`
13. Neuhaus, P., Couto, J., Wehrmann, J., Ruiz, D., Meneguzzi, F.: GADIS: A genetic algorithm for database index selection. In: The 31st International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. 39–42 (2019). DOI 10.18293/SEKE2019-135
14. Olofson, C.W.: Ensuring a fast, reliable, and secure database through automation: Oracle autonomous database. White paper, IDC Corporate USA, Sponsored by: Oracle Corp. (2018)
15. Ortiz, J., Balazinska, M., Gehrke, J., Keerthi, S.S.: Learning state representations for query optimization with deep reinforcement learning. CoRR **abs/1803.08604**, 1–5 (2018). URL `http://arxiv.org/abs/1803.08604`
16. Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T.C., Perron, M., Quah, I., et al.: Self-driving database management systems. In: Conference on Innovative Data Systems Research (CIDR), pp. 1–6. CIDR, Chaminade, California (2017)
17. Pedrozo, W.G., Nievola, J.C., Ribeiro, D.C.: An adaptive approach for index tuning with learning classifier systems on hybrid storage environments. In: International Conference on Hybrid Artificial Intelligence Systems, pp. 716–729. Springer (2018)
18. Petraki, E., Idreos, S., Manegold, S.: Holistic indexing in main-memory column-stores. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, pp. 1153–1166. ACM, New York, NY, USA (2015). DOI 10.1145/2723372.2723719. URL `http://doi.acm.org/10.1145/2723372.2723719`
19. Popovic, J.: Automatic tuning - SQL Server (2017). URL `https://docs.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning`. Accessed: 2019-06-17
20. POWA: PostreSQL Workload Analyzer (2019). URL `http://powa.readthedocs.io`
21. Ramakrishnan, R., Gehrke, J.: Database Management Systems, 3 edn. McGraw-Hill, Inc., New York, NY, USA (2003)
22. Sharma, A., Schuhknecht, F.M., Dittrich, J.: The case for automatic database administration using deep reinforcement learning. CoRR **abs/1801.05643**, 1–9 (2018). URL `http://arxiv.org/abs/1801.05643`
23. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press, Cambridge, Massachusetts (2018)
24. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction, chap. Introduction, pp. 1–13. MIT press, Cambridge, MA (2018)
25. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction, chap. Applications and Case Studies, pp. 421–453. MIT press, Cambridge, MA (2018)

26. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction, chap. Temporal-Difference Learning, pp. 119–138. MIT press, Cambridge, MA (2018)
27. Thanopoulou, A., Carreira, P., Galhardas, H.: Benchmarking with TPC-H on off-the-shelf hardware. In: 14th International Conference on Enterprise Information Systems, pp. 205–208. Springer, Wroclaw, Poland (2012)
28. TPC: Transaction performance council website (TPC) (1998). `http://www.tpc.org/`
29. Tsitsiklis, J.N., Roy, B.V.: An analysis of temporal-difference learning with function approximation. Tech. rep., Report LIDS-P-2322). Laboratory for Information and Decision Systems, Massachusetts Institute of Technology (1996)
30. Van Aken, D., Pavlo, A., Gordon, G.J., Zhang, B.: Automatic database management system tuning through large-scale machine learning. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1009–1024. ACM (2017)
31. Wang, J., Liu, W., Kumar, S., Chang, S.F.: Learning to hash for indexing big data—a survey. Proceedings of the IEEE **104**(1), 34–57 (2016)