PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

AUTOMATED DATABASE INDEXING USING MODEL-FREE REINFORCEMENT LEARNING

GABRIEL PALUDO LICKS

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Felipe Rech Meneguzzi

Porto Alegre
2021

REPLACE THIS PAGE WITH
THE LIBRARY CATALOG
PAGE

REPLACE THIS PAGE WITH THE COMMITTEE FORMS

ACKNOWLEDGMENTS

I thank my advisor Felipe Rech Meneguzzi and my colleagues Julia Colleoni Couto and Leonardo Rosa Amado for fundamental insights on the development of this research.

This work was supported by SAP SE. I thank SAP Labs Latin America for funding and providing feedback to carry out this research.

INDEXAÇÃO DE BANCOS DE DADOS AUTOMÁTICA UTILIZANDO APRENDIZADO POR REFORÇO

RESUMO

A configuração de bancos de dados para uma execução eficiente de *queries* é uma tarefa complexa, ficando a cargo de um administrador de banco de dados. Para isso, são utilizados índices, estruturas que facilitam a busca de registros e reduzem o tempo de resposta das *queries*, especialmente ao processar *queries* complexas. Porém, resolver o problema de criar índices que realmente otimizam o acesso ao banco de dados requer uma quantidade substancial de conhecimento do banco de dados e do domínio, cuja falta geralmente resulta em espaço e memória desperdiçados com índices irrelevantes, comprometendo o desempenho do banco de dados para *queries* e, certamente, degrada o desempenho da atualização de registros no banco. Nesta pesquisa, desenvolvemos a arquitetura SmartIX para resolver o problema de indexar automaticamente um banco de dados utilizando aprendizado por reforço para otimizar *queries* indexando dados ao longo da utilização de um banco de dados. Para avaliar seu desempenho, utilizamos o banco de dados TPC-H, referência na literatura para *benchmarking* de bancos de dados. Nossa avaliação experimental mostra que nossa arquitetura converge para configurações de índices com desempenho superior em comparação à trabalhos relacionados que utilizam aprendizado por reforço e algoritmos genéticos, constantemente mantendo configurações de índices próximas do ótimo e eficientemente escalando para bancos de dados maiores.

Palavras-Chave: inteligência artificial, aprendizado por reforço, bancos de dados, indexação automática.

AUTOMATED DATABASE INDEXING USING MODEL-FREE REINFORCEMENT LEARNING

ABSTRACT

Configuring databases for efficient querying is a complex task, often carried out by a database administrator. To reduce the response time of queries, especially complex ones, index structures are created to facilitate the search for data. However, 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 research, we develop the 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. We train our reinforcement learning agent and evaluate its performance in experiments using TPC-H, a standard, and scalable database benchmark. 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.

Keywords: artificial intelligence, reinforcement learning, databases, automated indexing.

CONTENTS

1. INTRODUCTION

Database indexes optimize queries and reduce their response time, especially when computing complex queries [Ramakrishnan and Gehrke, 2003, Ch. 8, p. 296]. More than finding the correct columns to index, it is important to balance the number of indexes created in a database [Ramakrishnan and Gehrke, 2003, Ch. 20, p. 654]. Too many indexes can result in an overhead when maintaining their organization to match data updates, whereas too few indexes might not be enough to have a positive performance impact [Ramakrishnan and Gehrke, 2003, Ch. 20, p. 654]. Thus, indexing is task that needs to be performed continuously with diligence, as its configuration directly impacts on a database's overall performance.

Index tuning is a task that is usually carried out by a human database administrator (DBA). However, frequently analyzing the database workload in order to find the correct attributes to index and try each candidate index configuration is time consuming. Machine learning techniques are being used in a variety of tasks related to database management systems and automated database administration. These include broader aspects such as completely automated database management [Pavlo et al., 2017], narrowing down to specific tasks such as query optimization [Marcus and Papaemmanouil, 2018, Kraska et al., 2018]. The use of learning algorithms to tackle the problem of index tuning, especially due to the problem complexity [Ramakrishnan and Gehrke, 2003, Ch. 20, p. 664], can be beneficial for exploring and computing estimates to a higher number of possible configurations in order to achieve the best set.

Specifically, recent research applied reinforcement learning (RL) methods to the case of index tuning [Basu et al., 2016, Sharma et al., 2018]. Reinforcement learning is a class of machine learning algorithms that aims to optimize decision making while maximizing the total reward of an agent in an environment. Analogous to the case of index tuning, frequent decisions on creating or dropping indexes have to be taken in order to optimize the performance of a database to a given workload. Such analogy makes reinforcement learning likely to decide what index tuning requires and abstracts away the DBA's task of frequently analyzing all candidate columns to index.

We developed an architecture for automated and dynamic database indexing that evaluates queries to make decisions on whether to create or drop indexes using reinforcement learning. Our architecture allows continuous assessment and automatic modification of the database index configuration, according to the queries the database receives. The architecture is composed of a reinforcement learning agent, an environment representation of the database that interacts with the agent, and an interface to apply changes to the database. The agent then explores this environment to find the optimal set of indexes concerning the current workload of queries. By exploring different index configurations, we expect the agent

to learn the impact of each index with regard to the workload and to optimize performance accordingly, allowing automatic modification of the index configuration.

We perform 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 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. Results show that the architecture consistently maintains near-optimal index configurations both in fixed and shifting database workloads, and scales to large databases without the need to be re-trained.

This manuscript is structured as follows. First, in Chapter 2, we provide background about indexing in relational databases and reinforcement learning. In Chapter 3, we detail the architecture we built for automated database indexing. In Chapter 4, we report the results of training the reinforcement learning agent and compare its performance to other approaches. In Chapter 5, we describe related work on the topic of reinforcement learning for database indexing. And lastly, in Chapter 6, we show the contributions, limitations, and future work of this research.

2. BACKGROUND

This chapter provides the background required to understand the remained of this manuscript. In Section 2.1, we provide an overview of the use of indexes in relational databases. In Section 2.2, we introduce the basic concepts about reinforcement learning and how problems can be modeled and solved through its techniques.

2.1 Indexing in relational databases

A DBMS is a software designed to manage databases and facilitate organizing collections of data efficiently [Ramakrishnan and Gehrke, 2003, Ch. 1, p. 4]. In particular, we address relational DBMSs, which are based on the relational model [Ramakrishnan and Gehrke, 2003, Ch. 1, p. 10], where data collections can be thought as tables whose rows represent records and columns represent attributes [Ramakrishnan and Gehrke, 2003, Ch. 3, p. 60]. The way a DBMS stores data internally is through *files*, each of which consists of pages [Ramakrishnan and Gehrke, 2003, 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 [Ramakrishnan and Gehrke, 2003, Ch. 8, p. 274].

An important technique to file organization in a DBMS is *indexing* [Ramakrishnan and Gehrke, 2003, 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 [Ramakrishnan and Gehrke, 2003, Ch. 8, p. 276].

2.1.1 Index data structures and types

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 [Ramakrishnan and Gehrke, 2003, 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 [Ramakrishnan and Gehrke, 2003, 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 [Ramakrishnan and Gehrke, 2003, Ch. 8, p. 280].

The type of an index depend 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 [Ramakrishnan and Gehrke, 2003, 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 [Ramakrishnan and Gehrke, 2003, 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 [Ramakrishnan and Gehrke, 2003, Ch. 8, p. 296].

2.1.2 Index tuning

Indexing is a task usually undertaken by the Database Administrator (DBA), a person who has considerable domain knowledge in order to make such decisions [Ramakrishnan and Gehrke, 2003, 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 [Ramakrishnan and Gehrke, 2003, 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 [Ramakrishnan and Gehrke, 2003, 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 the problem complexity [Wang et al., 2015, Elfayoumy and Patel, 2012, Duan et al., 2009]. The idea is that, given the database schema and the workload it receives, we can define the problem of finding a good index configuration that optimizes database operations [Ramakrishnan and Gehrke, 2003, 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 amount 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

$$\sum_{i=1}^{c} \frac{n!}{(n-i)!} \tag{2.1}$$

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 [Ramakrishnan and Gehrke, 2003, 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 decision to apply changes is made by the DBA. Most recent versions of DBMSs such as Oracle [Olofson, 2018] and Azure SQL Database [Popovic, 2017] can automatically adjust indexes. 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 performance of queries that read data from the tables; and identifies the redundant indexes or indexes that were not used in longer period of time that could be removed [Popovic, 2017].

2.1.3 Performance optimization and measurement

In order to optimize the performance of queries, a DBMS evaluates different plans of performing the operations a query is composed of [Ramakrishnan and Gehrke, 2003, Ch. 12, p. 404]. An efficient query execution plan tries to minimize the cost each operation incurs, and such plans can benefit from the presence of indexes that consequently diminish the cost of information retrieval [Ramakrishnan and Gehrke, 2003, Ch. 12, p. 404]. A cost model is usually predefined and contains estimates that take into account aspects related to CPU, network transmission, and disk I/O [Ramakrishnan and Gehrke, 2003, Ch. 8, p. 284].. The total cost of execution of a query can be interpreted as a performance measure to how the current DBMS configuration can respond to a query, such that the lower the cost is, the better.

A more general way of evaluating performance is through benchmarking. Since DBMSs are complex pieces of software and each has its own techniques to optimization, external organizations have defined protocols to evaluate their performance [Ramakrishnan and Gehrke, 2003, Ch. 20, p. 682]. The Transaction Processing Performance Council (TPC), for example, is an organization created to define benchmarks that evaluate the performance of database transactions in various aspects and contexts of workloads [Ramakrishnan and Gehrke, 2003, Ch. 20, p. 683]. 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 [Ramakrishnan and Gehrke, 2003, Ch. 20, p. 683].

TPC-H Benchmark

The TPC[1] is a non-profit corporation that produces benchmarks to measure database performance [TPC, 1998]. 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 [Thanopoulou. et al., 2012]. 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

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

second Refresh Function (RF2), that drops the same percentage of rows as the RF1. This metric is computed using the formula in Equation 2.2:

$$Power@Size = \frac{3600}{\sqrt[24]{\pi_{i=1}^{22} QI(i,0) \times \pi_{j=1}^{2} RI(j,0)}} \times SF \tag{2.2}$$

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 [Thanopoulou. et al., 2012]. It computes the performance of the system against a multi-user workload performed in an elapsed time, using the formula in Equation 2.3:

$$Throughput@Size = \frac{S \times 22}{T_S} \times 3600 \times SF \tag{2.3}$$

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.

$$QphH@Size = \sqrt{Power@Size \times Throughput@Size} \tag{2.4}$$

Equation 2.4 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.

2.2 Reinforcement Learning

Reinforcement learning is the closest machine learning paradigm to how humans learn, with its algorithms strongly inspired by biological aspects [Sutton and Barto, 2018, 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 [Sutton and Barto, 2018, Ch. 1, p. 1-2]. The objective of a reinforcement learning agent is to maximize its accumulated reward in the environment it is acting on, ultimately leading to a policy that maps the best actions to take in each state.

In this section, we first introduce Markov Decision Processes (MDPs), a mathematical framework used to model reinforcement learning problems (Subsection 2.2.1). Second, we introduce methods that can solve MDPs when its entire model is available (Subsection 2.2.2). Third, we describe methods capable of solving MDPs without complete knowledge of the environment (Subsection 2.2.3). Last, we detail techniques used to deal with MDPs with large state spaces through generalization (Subsection 2.2.4).

2.2.1 Markov Decision Processes

Introduced by [Bellman, 1957], a Markov Decision Process (MDP) is a formalization to decision making problems and is widely used to model reinforcement learning problems. The way an agent transitions through an MDP is by sequential decision making, i.e. choosing actions that lead from one state to another. State transitions and decision making in MDPs are characterized by: (1) a stochastic transition system, which determines the probabilities to which state the decision making agent reaches after taking an action; and (2) the *Markov property*, which dictates that every transition between states depends exclusively on the last visited state, rather than the history of states before that [Sutton and Barto, 2018, Ch. 3, p. 49].

An MDP is 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 [Sutton and Barto, 2018, Ch. 3]. More specifically, at each time step t the agent interacts with the environment by taking an action $a_t \in \mathcal{A}$ in state $s_t \in \mathcal{S}$. As a consequence, the agent receives a reward $r_{t+1} \in \mathcal{R}$ and reaches a new state $s_{t+1} \in \mathcal{S}$ with probability $p(s_{t+1}|s_t, a_t)$ given the transition probability function [Sutton and Barto, 2018, Ch. 3, p. 48]. Figure 2.1 illustrates the agent-environment interaction in a MDP.

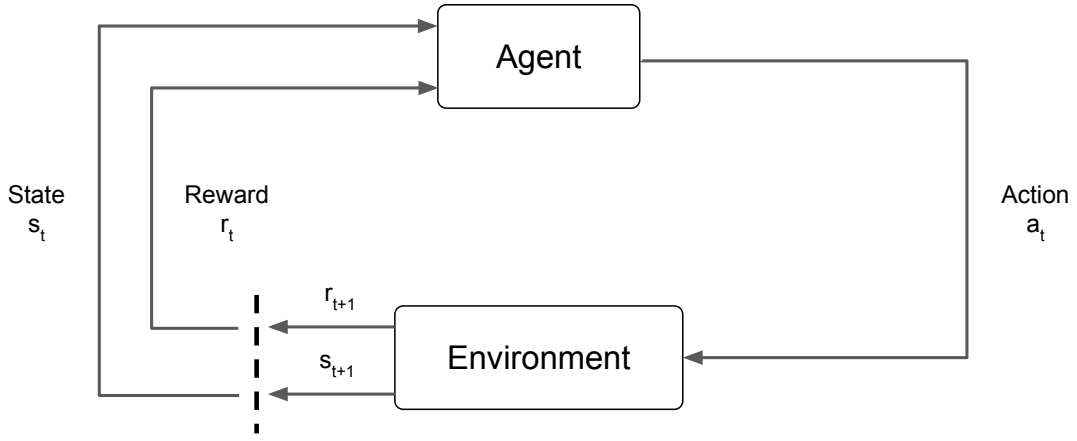


Figure 2.1: Agent-environment interaction in an MDP [Sutton and Barto, 2018, Ch. 3, p. 48].

The way an agent learns to behave is by taking actions and observing rewards, and choosing which actions to take depends on the policy an agent is following. Ideally, an agent starts exploring different states in order to learn the actions that lead to better rewards, and then starts exploiting the states that maximize the accumulated reward over time. Such behavior is defined as an ϵ-greedy policy. It means that, with ϵ-probability, the agent chooses whether to take a *random action*$_a$ or an *argmax*$_a$ among the available actions in a given state. The epsilon value is decayed after a predefined number of transitions, so that the agent initially explores with a higher probability different states and starts exploiting later on.

In order to learn which states are better than others, the agent approximates what is called the value function. The value function, denoted as V_π, returns an estimation of the accumulated reward an agent might expect from a state while following a policy π onwards. The value function for a state following a policy π is computed using the Bellman Expectation Equation [Sutton and Barto, 2018, Ch. 3, p. 62]:

$$v_\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma v_\pi(s_{t+1}) \mid s] \,. \tag{2.5}$$

The expectation operator $\mathbb{E}$ determines that the value of a given state is an average value of what is expected from its successor states in the long run. Suppose there are two policies π and π', π is better than π' if $v_\pi(s) \geqslant v_{\pi'}(s)$, $\forall s \in \mathcal{S}$, where $v_\pi(s)$ is the utility of a state estimated by the value function under a policy π [Sutton and Barto, 2018, Ch. 3, p. 62]. The optimal solution, thus, is a policy π_* that is better than or equal to all other policies [Sutton and Barto, 2018, Ch. 3, p. 62].

2.2.2 Dynamic programming methods

When a perfect model of the environment is available, i.e. the whole MDP dynamics, there are dynamic programming algorithms that are capable of finding its optimal solution (the optimal policy π_*) [Sutton and Barto, 2018, Ch. 4, p. 73]. These are not learning algorithms, and it is possible to compute them using a closed form, however it may be more efficient to compute the solution iteratively. The strategy to find the solution is to approximate the value function by iteratively sweeping the state space and updating their values through the Bellman Equation. In reinforcement learning, such algorithms with complete information of the environment are classified as model-based algorithms. Algorithms that deal with incomplete information of the environment are classified as model-free algorithms, which we address in the next subsection.

Dynamic programming methods for solving MDPs are built upon two processes: policy evaluation and policy improvement. Policy evaluation is the process of computing the

value function v_π for a policy π , whereas policy improvement is the process of generating a new policy $\pi' \geqslant \pi$ by acting greedily with respect to π [Sutton and Barto, 2018, Ch. 4, p. 74-79]. We compute the value function so that we can find better policies by greedily selecting actions using the computed value function of the neighborhood of each state [Sutton and Barto, 2018, Ch. 4, p. 80]. The *policy improvement theorem* states that, if a new greedy policy π' is as good as (but not better) than the previous policy π, both π and π' must be optimal policies [Sutton and Barto, 2018, Ch.4, p. 79]. A dynamic programming algorithm that effectively combines both of these processes at each sweep in the state space is the Value Iteration algorithm [Sutton and Barto, 2018, Ch. 4, p. 83], as shown in Algorithm 2.1.

Algorithm 2.1: Value Iteration pseudo-code [Sutton and Barto, 2018, Ch. 6, p. 83]

Input: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$ arbitrarily, $\forall s \in \mathcal{S}$ except for V(terminal) = 0
repeat
  for $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta \leftarrow max(\Delta, |v - V(s)|)$
until $\Delta < \theta$
Output: a policy $\pi \approx \pi_*$ such that $\pi(s) = argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

For each state of the set, the algorithm gets the current value of the state from the value function and computes a new value for that state by greedily selecting the value that corresponds to the maximum value achieved by acting in that state. Then, it calculates the difference between the previous state value and the new greedy state value and stores the maximum between this error and the error of the previous seen state. Once the algorithms updates the local value of the entire state space, it can converge if the maximum error is less than a threshold of accuracy, or otherwise it runs another sweep of the state space. The output is a deterministic policy resulting from greedily acting with regard to the value function computed by the algorithm.

2.2.3 Temporal-Difference Learning

Temporal-difference learning (TD learning) is a method formally introduced by [Sutton, 1988] for predicting future behavior using past experience in incomplete models of the environment, where the transition and reward functions are not known. Alongside TD learning methods, there are also Monte Carlo methods that solve model-free reinforcement learning problems, though we do not go into further detail about such methods. Regardless, the difference between the two is that Monte Carlo estimates rely on a complete sequence of

experiences from the state it is estimating to the terminal state. TD methods, rely on a technique called *bootstrapping*, which means that the update of a state value is based on the existing estimate of its successor state [Sutton and Barto, 2018, Ch. 4, p. 89]. Such characteristics imply that Monte Carlo methods are only defined to learning in *episodic* environments (terminating), whereas TD methods can learn in *continuing* environments (non-terminating) [Sutton and Barto, 2018, Ch. 5, p. 91]. The reason we do not go into detail about Monte Carlo methods is that, since learning indexes in databases is not assumed to terminate, such methods are not suitable for the task.

We now introduce the notion of a state-action value $q(s, a)$, i.e. the action-value function, which does not exclusively estimate the value of a state, but the value of taking an action at a particular state. This is the first step to control behavior optimally in the environment [Sutton and Barto, 2018, Ch. 6, p. 129], as we do not know the transition probabilities of the MDP. Computing the Bellman Expectation Equation for state-action values q_π is essentially the same as the state-value function v_π, but now considering transitions of state-action pairs [Sutton and Barto, 2018, Ch. 6, p. 129]:

$$q_\pi(s, a) = \mathbb{E}_\pi[r_{t+1} + \gamma q_\pi(s_{t+1}, a') \mid s, a] \,, \tag{2.6}$$

which represents the expected discounted reward an agent might receive by following policy π after taking action a_t at s_t onwards. The action-value function is computed with regard to a policy through incremental updates

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha[r_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)] \,, \tag{2.7}$$

where $\alpha \in [0, 1]$ is a step size with regard to the *TD error*, which is the difference between the the previous estimate $q(s_t, a_t)$ and a better estimate $r_{t+1} + \gamma q(s_{t+1}, a_{t+1})$.

There are two approaches that TD methods use to estimate the action-value function: on-policy and off-policy learning [Sutton and Barto, 2018, Ch. 6, p. 138]. In both approaches, the policy an agent follows is called the behavior policy. The behavior policy is usually an ϵ-greedy exploration function, which means that, with ϵ probability the agent chooses whether to take a *random action*$_a$ or an *argmax*$_a$ action of highest utility among the available actions at a given state [Sutton and Barto, 2018, Ch. 6, p. 129]. The difference lies on how the two approaches update action-values. The on-policy approach updates the action-value function with regard to the action selection from the same behavior policy it is following ($q(s_{t+1}, a_{t+1})$), as in Equation 2.7), whereas an off-policy approach updates values with regard to a greedy action selection ($max_a Q(s_{t+1}, a)$), as in the following Equation:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a q(s_{t+1}, a) - q(s_t, a_t)] \,. \tag{2.8}$$

The algorithm for learning on-policy is called SARSA [Rummery and Niranjan, 1994], which stands for the tuple $\langle S, A, R, S', A' \rangle$ of each transition update, targeted to the action that is selected following the behavior policy. The off-policy algorithm is called Q-Learning [Watkins, 1989], whose updates are targeted to the action that returns the maximum value (greedy selection with regard to the current action-value function). We show a pseudo code for Q-Learning in Algorithm 2.2. Notice that both algorithms can be represented essentially the same, except for the update rule.

Algorithm 2.2: Q-Learning pseudo-code [Sutton and Barto, 2018, Ch. 6, p. 131]

Define a step size $\alpha \in (0, 1]$ and $\epsilon \in (0, 1]$
Initialize $q(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$ arbitrarily, except that $q(terminal, \cdot) = 0$
for each episode **do**
    $s \leftarrow$ initial state
    repeat for each step of episode
        Choose a from s using policy (e.g. ϵ-greedy)
        Take action a, observe r, s'
        $q(s, a) \leftarrow q(s, a) + \alpha[r + \gamma max_a q(s', a) - q(s, a)]$
        $s \leftarrow s'$
    until s is terminal

2.2.4 Function Approximation

The methods described so far are called tabular methods, as each estimated action-value is updated and stored in a table of state-action values. The problem with tabular approaches 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 is close enough to the true value to be able to compute an optimal policy [Sutton and Barto, 2018, Ch. 9, p. 196-197]. Assuming that states can be described in terms of informative features, such problem can be handled by using linear function approximation, which is to use a parameterized representation for the action-value function other than a look-up table [Tsitsiklis and Van Roy, 1997]. 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 [Sutton and Barto, 2018, Ch. 9, p. 195].

We approximate action-values using Equation 2.9

$$\hat{q}(s, a) \leftarrow \theta_0 + \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s) \tag{2.9}$$

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 action-value function by employing gradient descent in the TD error with regard to each feature parameter:

$$\theta_i \leftarrow \theta_i + \alpha[r(s) + \gamma max_{a'} \hat{q}_\theta(s', a') - \hat{q}_\theta(s, a)] \frac{\partial \hat{q}_\theta(s, a)}{\partial \theta_i} \tag{2.10}$$

Using Equation 2.10, we adjust our Θ parameters to reduce the temporal difference between successive states and update these parameters in the direction of decreasing the error after each trial [Tsitsiklis and Van Roy, 1997]. 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 action-values by learning and updating Θ parameters throughout algorithm iterations.

Neural networks as function approximators

Since not all functions can be approximated linearly, an alternative is to use use neural networks with non-linear activation as approximators. In terms of the error in approximating the value function, the ideal is to find a set of parameters Θ that leads the agent to a global optimum. However, finding global optima is rarely possible for non-linear approximators, and sometimes possible for linear ones [Sutton and Barto, 2018, Ch. 9, p. 200]. It is generally enough, though, when non-linear approximators such as neural networks converge to a local optima [Sutton and Barto, 2018, Ch. 9, p. 200].

Though neural networks are effective function approximators in many tasks, it is not easy to apply them in the context of reinforcement learning due to the high correlation between data inputs to update the value function [Mnih et al., 2015]. Specifically, learning off-policy produces higher variance updates than learning on-policy, which can result in steps that greatly vary in size and lead to parts of the space with a different gradient [Sutton and Barto, 2018, Ch. 11, p. 283]. Neural networks within reinforcement learning become widely used only after [Mnih et al., 2015] successfully approximated the value function using deep neural networks to play Atari games.

In order to avoid oscillations in the parameters, [Mnih et al., 2015] propose a method called experience replay, which randomly samples mini-batches of past agent experience to update the value function. Experience replay is a method stores agent's experience tuples $e = \langle s, a, r, s' \rangle$ at each time-step in a replay memory $D = \{e_1, ..., e_n\}$. 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 [Sutton and Barto, 2018, Ch. 16, p. 440]. The aim is to reduce the variance of updates, as successive updates are not correlated with one another [Mnih et al., 2013]. Such method still requires many samples in order to converge to the true value function, e.g. 50 million transitions for the case of each Atari game in the work from [Mnih et al., 2015].

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.

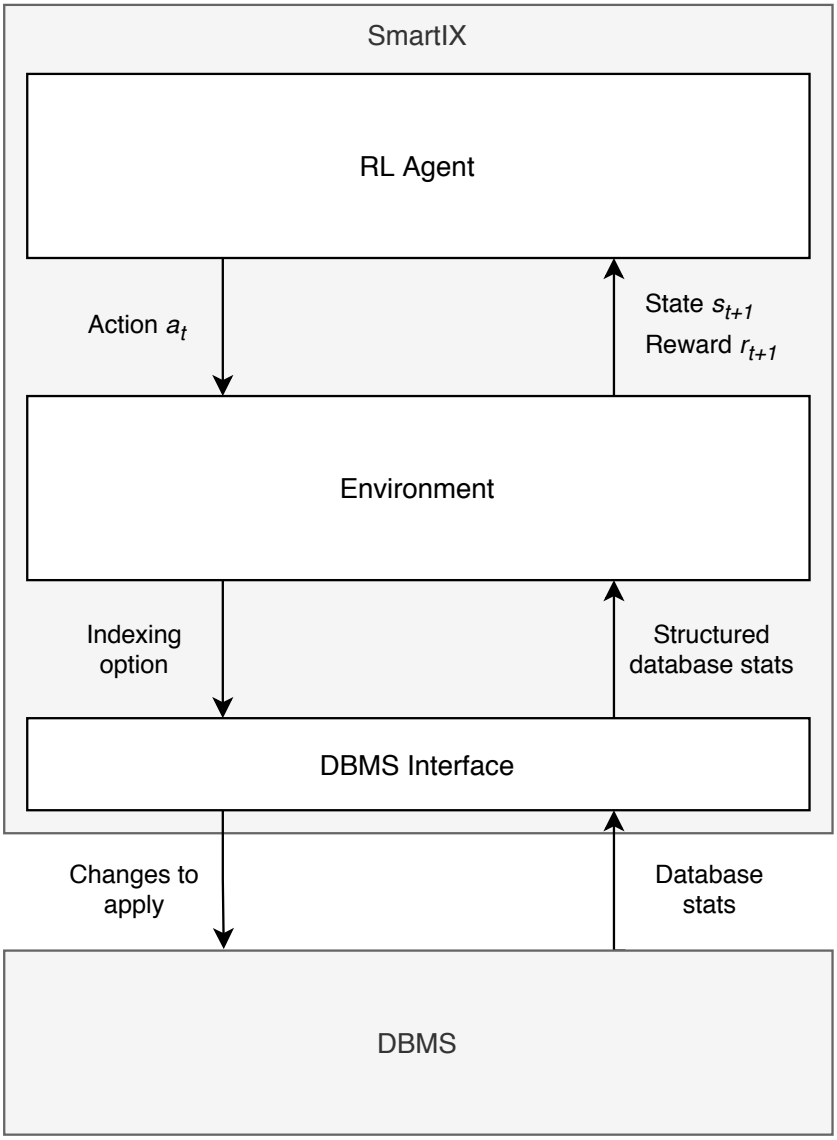


Figure 3.1: SmartIX architecture.

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 changes persistent, there is a DBMS interface that is responsible for communicating with the DBMS to create or drop indexes, and fetch statistics of the current index configuration. We describe in detail each of these components in the next sections.

3.1 Agent

Our agent is based on the Deep Q-Network agent proposed by [Mnih et al., 2015]. The algorithm consists of a reinforcement learning method built around the Q-learning, using a neural network for function approximation, and a replay memory for 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. Algorithm 3.1 depicts the steps performed by the agent.

Algorithm 3.1: Indexing agent with function approximation and experience replay. Adapted from [Sutton and Barto, 2018, Ch. 6, p. 131] and [Mnih et al., 2015].

1: Random initialization of the value function
2: Empty initialization of a replay memory D
3: $s \leftarrow$ *DB initial index configuration mapped as initial state*
4: **for** each step **do**
5: $\quad a \leftarrow$ *epsilon greedy*(s)
6: $\quad s', r \leftarrow$ *execute*(a)
7: $\quad$ Store experience $e = \langle s, a, r, s' \rangle$ in D
8: $\quad$ Sample random mini-batch of experiences $e \sim D$
9: $\quad$ Perform experience replay using sampled mini-batch
10: $\quad s \leftarrow s'$

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.

3.2.1 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. An example illustrating such vector is shown in Figure 3.2. 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, the concatenate both $\vec{I}$ and $\vec{Q}$ to generate our binary state vector $\vec{S}$ with length $|\vec{S}| = 2C$.

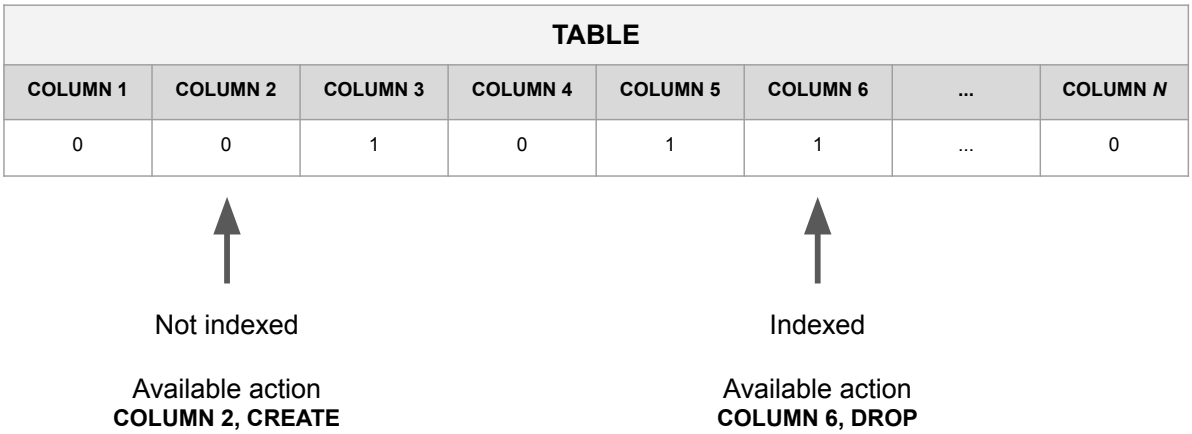


Figure 3.2: Representation of index configuration vector and available actions.

3.2.2 Actions

In our environment, we define the possible actions as a set A of size $C + 1$. Each one of the C actions refer 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. This behavior is illustrated in Figure 3.2, where each column can be either dropped or created an index on it. 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.

3.2.3 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, thus, can be computed using Equation 3.1:

$$R(op, use) = (1 - op)((1 - use)(1) + (use)(-5)) + (op)((1 - use)(-5) + (use)(1)) \quad (3.1)$$

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 3.1.

4. EXPERIMENTS

In this chapter, we report the experiments using the SmartIX architecture. First, we introduce the experimental setup and the baselines we use in this chapter to compare results (Section 4.1). Second, we report the results on training our reinforcement learning agent (Section 4.2). Third, we analyze the results of our agent in comparison to the baselines and related work, using the static index configuration in which each algorithm converged to after training (Section 4.3). Fourth, we report the results on the dynamic index configurations, i.e. changing indexes over time, while algorithms follow their trained policies (Chapter 4.4). Lastly, we show how the our agent efficiently indexes databases that are larger than the one used for training the agent (Chapter 4.5).

4.1 Experimental setup

We now detail the database setup we use for the experiments and each of the baselines used for performance comparison.

4.1.1 Database setup

Due to its usage in literature for measuring database performance [Thanopoulou. et al., 2012] [Pedrozo et al., 2018] [Neuhaus et al., 2019] [Basu et al., 2016], 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 we use for experiments is populated with 1GB of data, however we also show performance results in databases with 10GB and 100GB of data in later experiments. 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 the in Table 4.1 the number of rows 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. The complete schema of the TPC-H database is shown in Appendix A.

Table 4.1: TPC-H database - Table stats and indexes

Table	Total rows	Total Columns	Indexed Columns	Indexable Columns
REGION	5	3	1	2
NATION	25	4	2	2
PART	200000	9	1	8
SUPPLIER	10000	7	2	5
PARTSUPP	800000	5	2	3
CUSTOMER	150000	8	2	6
ORDERS	1500000	9	2	7
LINEITEM	6001215	16	4	12
Totals	8661245	61	16	45

By summing the amount 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. We can think of it as a matrix of 45 columns by over 35 trillion lines containing all possible combinations. 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 manually compute which columns compose the ground truth optimal index configuration. We manually create each index possibility and check whether an index benefits at least one query within the 22 TPC-H queries. To check whether an index is used or not, we run the EXPLAIN command to visualize execution plan of each query. After this analysis, 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.

4.1.2 Baselines

The baselines comprise different indexing configurations using different indexing approaches, including commercial and open-source database advisors, 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:

1. **Default**: is the initial DBMS index configuration and contains no indexes except the standard on primary and foreign keys;

2. **All indexed**: is the configuration that contains all columns indexed;

3. **Random policy**: is the best performing configuration explored by an agent following a policy that selects indexing options randomly over the course of 1000 iterations;

4. **EDB** [EnterpriseDB, 2019]: is the index configuration suggested by EnterpriseDB, a commercial database advisor tool;

5. **POWA** [POWA, 2019]: is the index configuration suggested by the PostgreSQL Workload Analyzer, an open-source advisor tool;

6. **ITLCS** [Pedrozo et al., 2018]: is the *Index Tuning with Learning Classifier System* (ITLCS), which combines a learning classifier and genetic algorithms to explore rules for efficient indexing;

7. **GADIS** [Neuhaus et al., 2019]: is the Genetic Algorithm for Database Index Selection (GADIS), which uses an evolutionary algorithm to explore index configurations encoded as individuals;

8. **NoDBA** [Sharma et al., 2018]: is a system based on a cross-entropy deep reinforcement learning method applied to recommend indexes for given workloads; and

9. **rCOREIL** [Basu et al., 2016]: is a system based on a policy iteration reinforcement learning method, which estimates a database cost model and suggests indexes that decrease such cost.

The EDB [EnterpriseDB, 2019], POWA [POWA, 2019] and ITLCS [Pedrozo et al., 2018] index configurations are a result of a study conducted by Pedrozo, Nievola and Ribeiro [Pedrozo et al., 2018]. The authors [Pedrozo et al., 2018] employ these methods to compute the actual indexes suggested by each method to each of the 22 queries in the TPC-H workload, which constitute the respective index configurations we use in this analysis. The index configurations of GADIS [Neuhaus et al., 2019], NoDBA [Sharma et al., 2018] and rCOREIL [Basu et al., 2016] are a result of experiments we ran using source-code provided by the authors. We execute the author's algorithms without modifying any hyperparameter except configuring the database connection. The index configurations 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

We now detail the agent's training process. In our case, to approximate the value function, we use a simple multi-layer perceptron neural network with two hidden layers and an Adam optimizer with a mean-squared error loss, both PyTorch 1.5.1 implementations using default hyperparameters [Paszke et al., 2019]. The input and output dimensions depend on the amount of columns available to index in the database schema, as shown in Section 3.2.1. The neural network architecture is detailed in Table 4.2.

Table 4.2: Neural network architecture (sequential model).

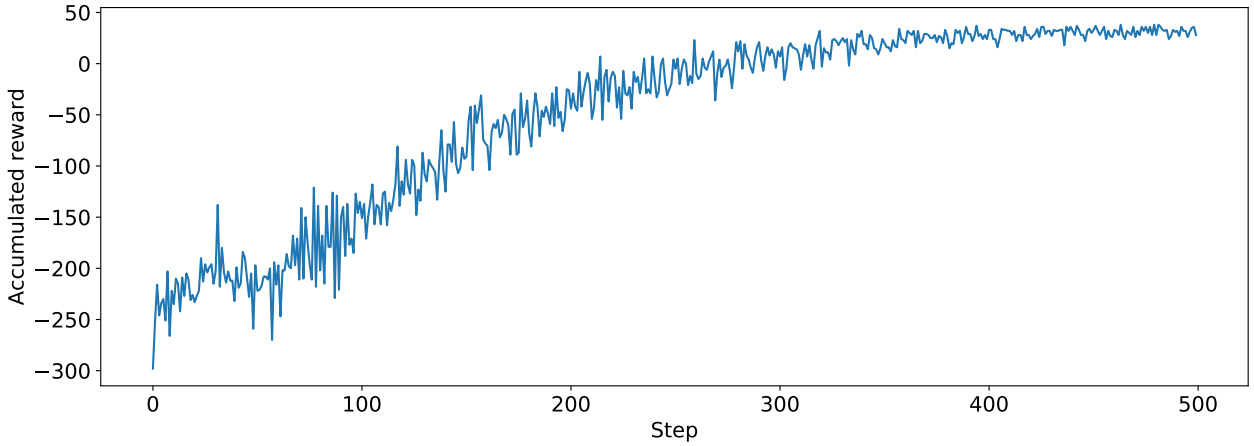
Layer	Dimension
Linear	(90, 128)
ReLU	
Linear	(128, 128)
ReLU	
Linear	(128, 46)

The hyperparameters we use for training the agent are shown in Table 4.3. The first two hyperparameters, *learning rate* and *discount factor*, are used in the update equation of the value function. The next two hyperparameters are related to experience replay, where *replay memory size* defines the amount of experiences the agent is capable of storing, and *replay batch size* defines the amount of samples the agent uses at each time step to update the value function. The other four hyperparameters are related to the epsilon-greedy exploration function, where we define an *epsilon initial* as maximum epsilon value, an *epsilon final* as epsilon minimum value, a percentage in which *epsilon decays*, and the interval of time steps at each decay. Finally, the last hyperparameter is the query horizon in which we keep track of the last H queries in the agent state representation, influencing on how fast the agent sees a workload shift and adapts the index configuration. All values of DQN's hyperparameters are a result of an informal search, such as employed by the original authors [Mnih et al., 2015], and the newly introduced query horizon H can be adapted if needed.

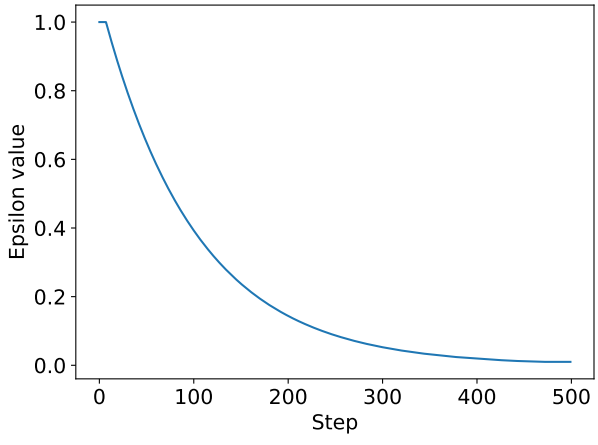
We train our agent for the course of 64 thousand time steps in the environment. Training statistics are gathered every 128 steps and are shown in Figure 4.1. Sub-figure 4.1a shows the total reward accumulated by the agent at each 128 steps in the environment, which consistently improves over time and stabilizes after the 400th x-axis value. Sub-figure 4.1b shows the corresponding epsilon value per 128 steps during training, which starts at 1.00 and decays towards its minimum value of 0.01. Sub-figure 4.1c 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.

Table 4.3: Training hyperparameters.

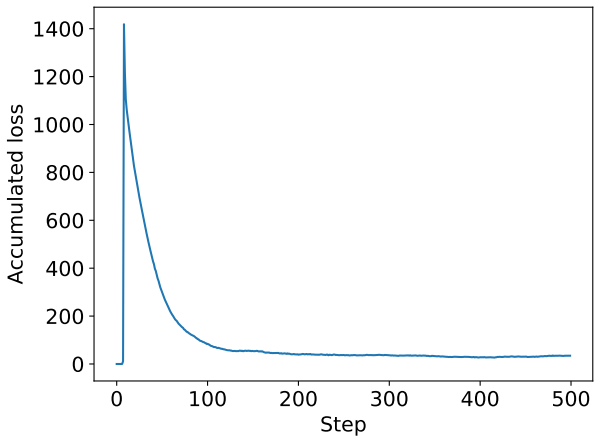
Hyperparameter	**Value**
Learning rate (α)	0.0001
Discount factor (γ)	0.9
Replay memory size	10000
Replay batch size	1024
Epsilon initial	1.00
Epsilon final	0.01
Epsilon decay	1%
Decay interval	128
Query horizon (H)	40



(a) Accumulated reward per 128 steps.



(b) Epsilon value per 128 steps.



(c) Accumulated loss per 128 steps.

Figure 4.1: Training statistics.

To evaluate the agent behavior and the index configuration in which the agent is converging to, we plot in Figure 4.2 data of each index configuration 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. Both the total amount of indexes *and* the total optimal indexes converge towards the ground truth optimal indexes. That is, the agent not only learns to keep the optimal indexes in the configuration, but it also drops the indexes that do not benefit the workload.

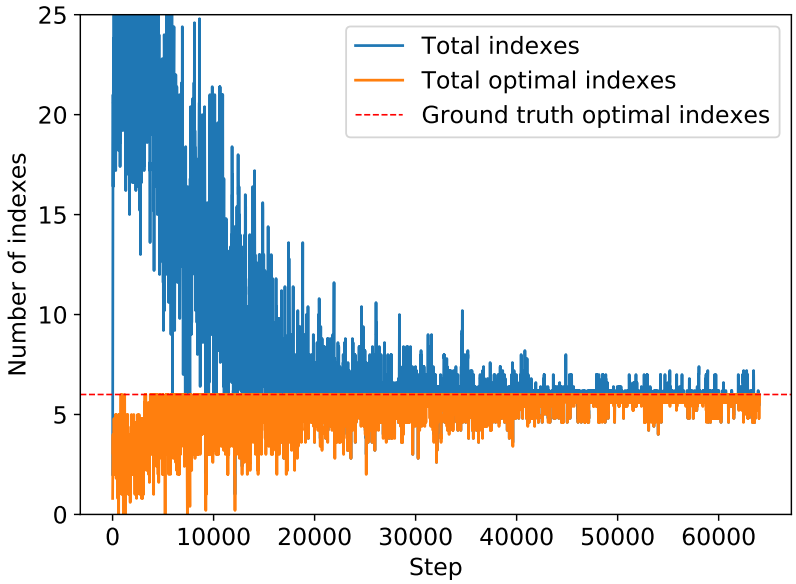
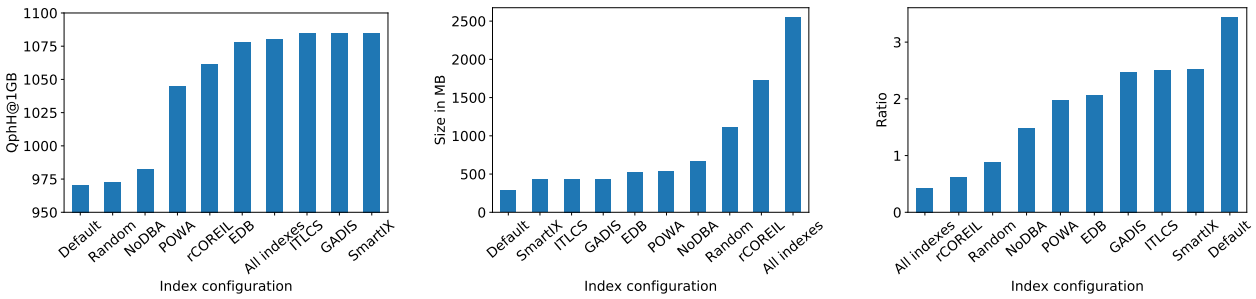


Figure 4.2: Indexes configuration at each step while training.

4.3 Static configurations

We now evaluate each baseline index configuration in comparison to the one in which our agent converged to. We show the TPC-H performance metric (QphH, i.e. the query-per-hour metric) and the index size of each configuration. Figure 4.3a shows the query-per-hour metric of each configuration (higher values denote better performance). The plotted values 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. Figure 4.3b 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. Therefore, in Figure 4.3c, we plot a ratio of query-per-hour over index size. A detailed table containing results of each benchmark execution and exact index sizes is shown in Appendix B, and a detailed list of each index configuration in Appendix C.



(a) Query-per-hour metric

(b) Index size (in MB)

(c) Ratio of performance and index size

Figure 4.3: Static index configurations results.

While SmartIX achieves the best query-per-hour-metric, the two genetic algorithms [Neuhaus et al., 2019] and [Pedrozo et al., 2018] have both very similar query-per-hour and index size metrics in comparison to our agent. GADIS [Neuhaus et al., 2019] 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 [Ramakrishnan and Gehrke, 2003, Ch. 8, p. 290-291]. It has the lowest ratio value due to the storage it needs to maintain indexes.

While rCOREIL [Basu et al., 2016] is the most competitive reinforcement learning method in comparison to SmartIX, 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 [Sutton and Barto, 2018, Ch. 4, p. 87], this can become a problem in databases that contain a larger number of columns to index.

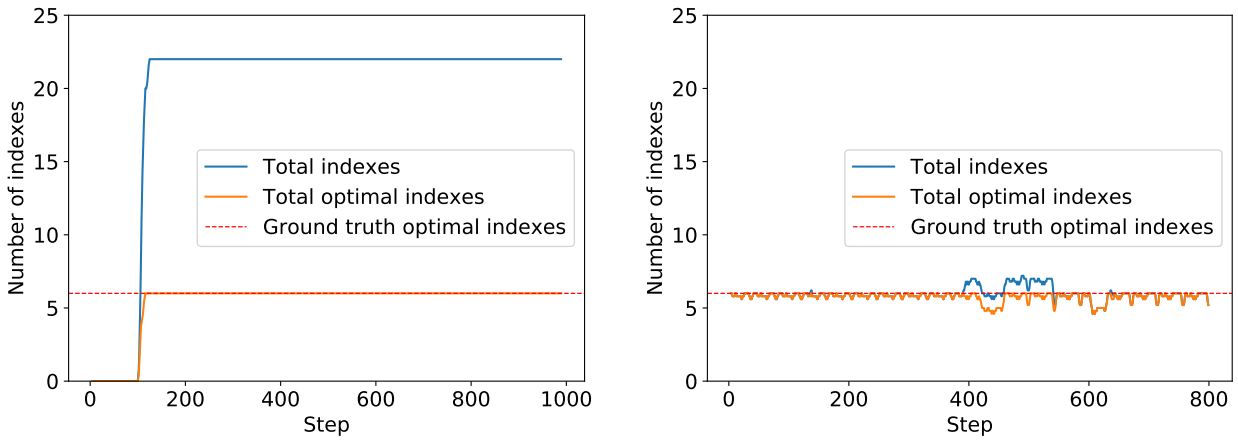
Among the database advisors, the commercial tool EDB [EnterpriseDB, 2019] achieves the highest query-per-hour metric in comparison to the open-source tool POWA [POWA, 2019], while its indexes occupy virtually the same disk space. Other baselines and related work are able to optimize the index configuration 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, 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.

4.4 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 Chapter 5). 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 Chapter 5). Therefore, we do not include NoDBA in the following analysis, and focus the discussion on rCOREIL and SmartIX.

4.4.1 Fixed workload

We now evaluate the index configuration of rCOREIL and SmartIX over time while the database receives a fixed workload of queries. Figure 4.4 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 that maintains all ground truth optimal indexes, however 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.

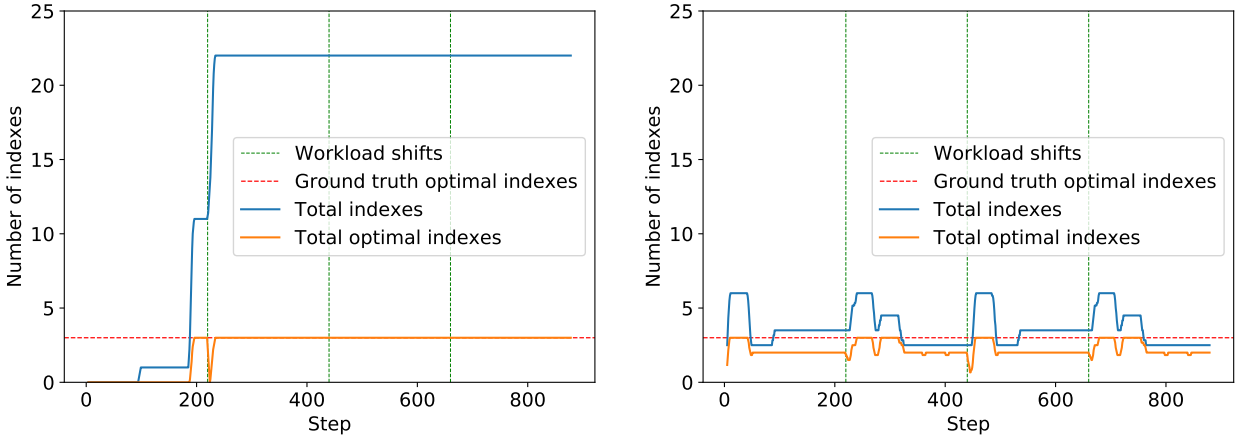


(a) rCOREIL (b) SmartIX

Figure 4.4: Agent behavior with a fixed workload.

4.4.2 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.



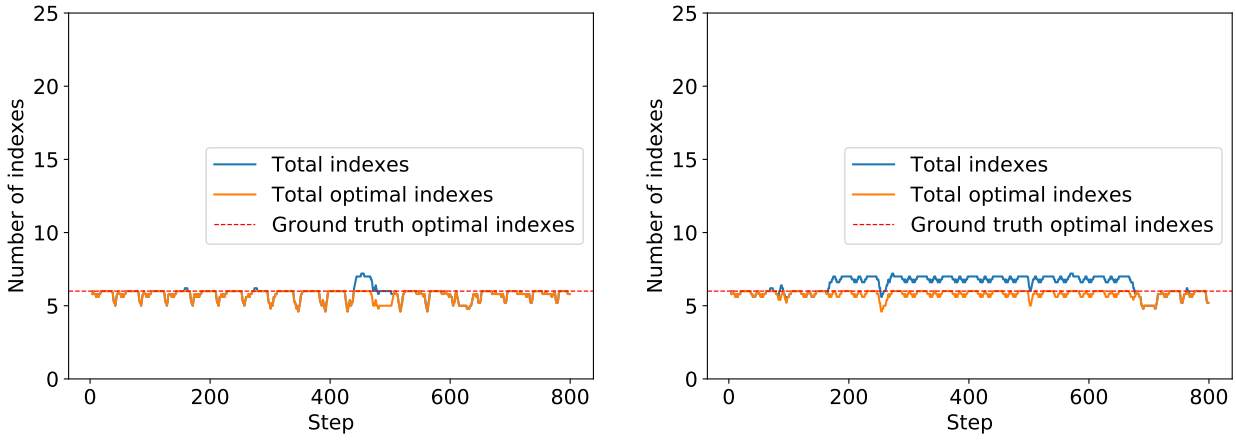
(a) rCOREIL (b) SmartIX

Figure 4.5: Agent behavior with a shifting workload.

The behavior of each algorithm is shown in Figure 4.5. 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 for 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.

4.5 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 4.6.



(a) 10GB TPC-H database.

(b) 100GB TPC-H database.

Figure 4.6: Agent behavior in larger databases.

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

5. RELATED WORK

Machine learning techniques are used in a variety of tasks related to database management systems and automated database administration [Van Aken et al., 2017]. One example is the work from Kraska et al. [Kraska et al., 2018], 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 [Pavlo et al., 2017] 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. [Basu et al., 2016] 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 [DB Group at UCSC, 2019] 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. [Sharma et al., 2018] 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 [Sharma et al., 2018]. 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 [TPC, 1998], 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. [Pavlo et al., 2017] 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. [Marcus and Papaemmanouil, 2018] proposed a proof-of-concept to determine the join ordering for a fixed database; Ortiz et al. [Ortiz et al., 2018] 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 dissertation, 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 its 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 less resources. Importantly, our learning approach proves to be practical in realistic scenarios of shifting workloads and very large database sizes.

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 informative in discovering whether the agent learns to avoid indexes in write-intensive tables.

Concerning the limitations above, future work can consider to: (1) investigate techniques that allow us to deal with composite indexes, which incurs a factorial increase in the state space; (2) improve the reward function to provide feedback in case of write-intensive workloads (the reward function needs continuous improvement, and inf fact there exists a whole research field on reward shaping); (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).

Finally, 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 less resources. As a result of this research, we published a paper at the Applied Intelligence journal [Licks et al., 2020]. At the same time of the development of this research, other papers in different subjects were co-authored [Amado et al., 2020b] [Amado et al., 2020a] [Amado et al., 2019].

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 [Giardino et al., 2016]. 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 baseline indexing recommendations.

REFERENCES

Amado, L. Aires, J. P. Pereira, R. F. Magnaguagno, M. C. Granada, R. Licks, G. P. Marcon, M. and Meneguzzi, F. (2020a). Latrec+: Learning-based goal recognition in latent space. In: *AAAI Workshop on Plan, Activity, and Intent Recognition (PAIR)*, vol. 34, pp. 1–2, New York, USA. AAAI.

Amado, L. Licks, G. P. Marcon, M. Pereira, R. F. and Meneguzzi, F. (2020b). Using self-attention LSTMs to enhance observations in goal recognition. In: *IEEE International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, Glasgow, UK. WCCI.

Amado, L. R. Aires, J. P. Pereira, R. F. Magnaguagno, M. C. Granada, R. Licks, G. P. and Meneguzzi, F. (2019). Latrec: Recognizing goals in latent space. In: *Demonstrations of The International Conference on Automated Planning and Scheduling (ICAPS)*, vol. 29, pp. 1–2, Berkeley, USA. ICAPS.

Basu, D. Lin, Q. Chen, W. Vo, H. T. Yuan, Z. Senellart, P. and Bressan, S. (Sep, 2016). Regularized cost-model oblivious database tuning with reinforcement learning. *Transactions on Large-Scale Data- and Knowledge-Centered Systems (TLDKS)*, vol. 28, pp. 96–132.

Bellman, R. (Apr, 1957). A markovian decision process. *Journal of Mathematics and Mechanics*, vol. 6, pp. 679–684.

DB Group at UCSC (2019). DBTune. Retrieved from URL https://github.com/dbgroup-at-ucsc/dbtune. October 2019.

Duan, S. Thummala, V. and Babu, S. (Aug, 2009). Tuning database configuration parameters with iTuned. *Very Large Data Base Endowment (VLDB Endownment)*, vol. 2, pp. 1246–1257.

Elfayoumy, S. and Patel, J. (2012). Database performance monitoring and tuning using intelligent agent assistants. In: *International Conference on Information and Knowledge Engineering (IKE)*, pp. 1–5, San Diego, USA. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

EnterpriseDB (2019). Enterprise Database. Retrieved from URL https://www.enterprisedb.com. October 2019.

Giardino, C. Paternoster, N. Unterkalmsteiner, M. Gorschek, T. and Abrahamsson, P. (Jun, 2016). Software development in startup companies: the greenfield startup model. *IEEE Transactions on Software Engineering (TSE)*, vol. 42, pp. 585–604.

Kraska, T. Beutel, A. Chi, E. H. Dean, J. and Polyzotis, N. (2018). The case for learned index structures. In: *International Conference on Management of Data (SIGMOD)*, pp. 489–504, New York, USA. ACM.

Licks, G. P. Couto, J. C. de Fátima Miehe, P. De Paris, R. Ruiz, D. D. and Meneguzzi, F. (Mar, 2020). SMARTIX: A database indexing agent based on reinforcement learning. *Applied Intelligence (APIN)*, vol. 50, pp. 2575–2588.

Marcus, R. and Papaemmanouil, O. (Jun, 2018). Deep reinforcement learning for join order enumeration. In: *1st International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*, pp. 1–4, New York, USA. ACM.

Mnih, V. Kavukcuoglu, K. Silver, D. Graves, A. Antonoglou, I. Wierstra, D. and Riedmiller, M. A. (Dec, 2013). Playing Atari with deep reinforcement learning. *arXiv preprint CoRR*, vol. abs/1312.5602, pp. 9.

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. (Feb, 2015). Human-level control through deep reinforcement learning. *Nature*, vol. 518, pp. 529–533.

Neuhaus, P. Couto, J. Wehrmann, J. Ruiz, D. and Meneguzzi, F. (Jul, 2019). GADIS: A genetic algorithm for database index selection. In: *31st International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pp. 39–42, Pittsburgh, USA. KSI Research Inc. and Knowledge Systems Institute.

Olofson, C. W. (2018). Ensuring a fast, reliable, and secure database through automation: Oracle autonomous database. White paper, IDC Corporate USA.

Ortiz, J. Balazinska, M. Gehrke, J. and Keerthi, S. S. (Mar, 2018). Learning state representations for query optimization with deep reinforcement learning. *arXiv preprint CoRR*, vol. abs/1803.08604, pp. 1–5.

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: Wallach, H. Larochelle, H. Beygelzimer, A. d'Alché-Buc, F. Fox, E. and Garnett, R., editors, *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, pp. 8026–8037. Curran Associates, Inc., 1 ed..

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: *Conference on Innovative Data Systems Research (CIDR)*, pp. 1–6, Chaminade, USA. CIDRDB, CIDR.

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: *International Conference on Hybrid Artificial Intelligence Systems (HAIS)*, pp. 716–729, Basel, Switzerland. Springer.

Popovic, J. (2017). Automatic tuning – SQL Server. Retrieved from URL https://docs.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning. June 2019.

POWA (2019). PostreSQL Workload Analyzer. Retrieved from URL http://powa.readthedocs.io. October 2019.

Ramakrishnan, R. and Gehrke, J. (2003). *Database Management Systems*. McGraw-Hill, New York, USA, 3 ed..

Rummery, G. A. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. White paper TR 166, Cambridge University Engineering Department, Cambridge, England.

Sharma, A. Schuhknecht, F. M. and Dittrich, J. (Jan, 2018). The case for automatic database administration using deep reinforcement learning. *arXiv preprint CoRR*, vol. abs/1801.05643, pp. 1–9.

Sutton, R. S. (Aug, 1988). Learning to predict by the methods of temporal differences. *Machine learning*, vol. 3, pp. 9–44.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press, Cambridge, Massachusetts, 2 ed..

Thanopoulou., A. Carreira., P. and Galhardas., H. (2012). Benchmarking with TPC-H on off-the-shelf hardware: An experiments report. In: *14th International Conference on Enterprise Information Systems (ICEIS)*, pp. 205–208, Wroclaw, Poland. INSTICC, SciTePress.

TPC (1998). Transaction Performance Council (TPC). Retrieved from URL http://www.tpc.org. June 2020.

Tsitsiklis, J. N. and Van Roy, B. (May, 1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control (TACON)*, vol. 42, pp. 674–690.

Van Aken, D. Pavlo, A. Gordon, G. J. and Zhang, B. (2017). Automatic database management system tuning through large-scale machine learning. In: *International Conference on Management of Data (SIGMOD)*, pp. 1009–1024, New York, USA. ACM.

Wang, J. Liu, W. Kumar, S. and Chang, S.-F. (Dec, 2015). Learning to hash for indexing big data: a survey. *Proceedings of the IEEE*, vol. 104, pp. 34–57.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Ph.D. Thesis, King's College, Cambridge, UK.

APPENDIX A – TPC-H DATABASE SCHEMA

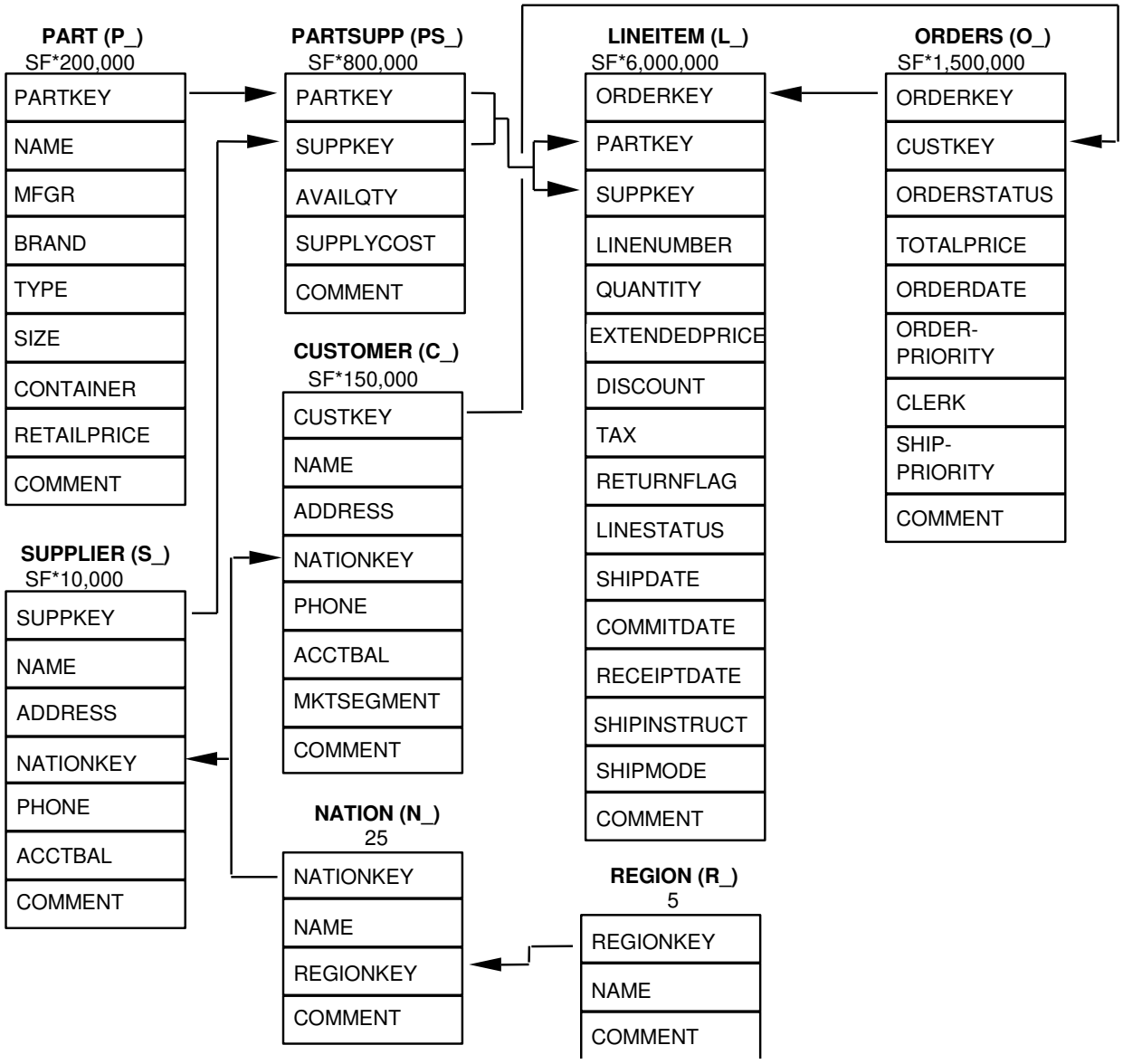


Figure A.1: TPC-H database schema.

APPENDIX B – BENCHMARK RESULTS

In the Table B.1, we show the benchmark results of each index configuration. Below, in Table B.2, we show de index size of each configuration.

Index config.	Power@1GB	Throughput@1GB	QphH@1GB	QphH std. dev.
No indexes	1489.3522	632.8253	970.7003	10.7479
Random	1481.6559	638.4742	972.5285	10.2342
NoDBA	1529.7047	631.4268	982.7456	14.0786
POWA	1573.4915	693.9250	1044.8518	11.9582
rCOREIL	1622.1830	695.1131	1061.8330	11.0346
EDB	1667.6451	696.5483	1077.7452	5.6812
All indexes	1668.3812	699.1538	1079.9548	9.0501
ITLCS	1659.7305	708.8918	1084.6054	13.6867
GADIS	1667.7764	705.8360	1084.9475	8.4136
SmartIX	1659.7895	709.4642	1085.0884	7.4550

Table B.1: TPC-H benchmark results for each configuration.

Index config.	Index size (in MB)
No indexes	282.62
SmartIX	429.89
ITLCS	433.85
GADIS	438.40
EDB	521.84
POWA	531.03
NoDBA	663.66
Random	1107.65
rCOREIL	1725.40
All indexes	2547.79

Table B.2: Index size of each configuration.

APPENDIX C – INDEX CONFIGURATIONS

In Table C.1, we detail each index configurations and annotate whether an index is among the optimal index set or not.

Table C.1: Index configurations

Index config.	Columns indexed	Index used?
SmartIX	c_acctbal	Yes
	p_brand	Yes
	p_size	Yes
	o_orderdate	Yes
	l_shipdate	Yes
GADIS	c_phone	No
	l_shipdate	Yes
	o_orderdate	Yes
	p_container	Yes
	ps_availqty	No
ITLCS	l_shipdate	Yes
	p_type	Yes
	o_orderdate	Yes
	ps_partkey	No
EDB	c_mktsegment	No
	o_orderdate	Yes
	p_type	No
	l_returnflag	No
	l_shipdate	Yes
	s_name	No
POWA	c_mktsegment	No
	l_returnflag	No
	l_shipdate	Yes
	o_orderdate	Yes
	p_name	No
	p_type	No
rCOREIL	l_returnflag	No
	l_quantity	No
	l_extendedprice	No
	l_discount	No
	l_receiptdate	No
	l_shipdate	Yes
	l_tax	No
	l_shipmode	No
	l_shipinstruct	No
	c_acctbal	Yes
	c_mktsegment	No
	c_name	No
	p_name	No
	p_brand	Yes
	p_type	No
	p_size	Yes
	p_container	Yes
	r_name	No
	s_name	No
	n_name	No
	o_totalprice	No
	o_orderstatus	No
	o_orderdate	Yes
	o_orderpriority	No
	o_shippriority	No
NoDBA	l_discount	No
	l_quantity	No
	l_shipdate	Yes
Random	l_extendedprice	No
	l_returnflag	No
	l_commitdate	No
	l_shipmode	No
	o_orderstatus	No
	o_clerk	No
	o_comment	No
	ps_supplycost	No
	ps_comment	No
	c_mktsegment	No
	c_phone	No
	s_address	No
	s_comment	No
	s_phone	No
	p_mfgr	No
	p_comment	No
	p_retailprice	No
	n_comment	No