SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning

Jan Kossmann
jan.kossmann@hpi.de
Hasso Plattner Institute
Potsdam, Germany

Alexander Kastius
alexander.kastius@hpi.de
Hasso Plattner Institute
Potsdam, Germany

Rainer Schlosser
rainer.schlosser@hpi.de
Hasso Plattner Institute
Potsdam, Germany

ABSTRACT

Well-chosen secondary indexes are highly relevant for database performance. For complex workloads, current index selection algorithms are either not fast or not highly competitive. Our approach, *SWIRL*, which is based on reinforcement learning, offers both. The necessary knowledge about index candidates is internalized during a preliminary training procedure. The training effort is then traded off against rapid runtimes while workloads containing known and unknown queries can be handled, which is relevant, e.g., in cloud-based business application scenarios.

In this work, we (i) explain how we model the index selection problem for reinforcement learning, (ii) enable our approach to generalize to unseen workloads, (iii) achieve efficient training for thousands of index candidates by relying on *invalid action masking*, and (iv) compare it to state-of-the-art and other RL index selection approaches on the analytical TPC-H, TPC-DS, and Join Order Benchmarks. The results show that overall *SWIRL* determines competitive index configurations and outperforms competitors in terms of runtime, some by orders of magnitude.

1 INTRODUCTION

Secondary indexes are indispensable for the performance of relational database systems [6]. Determining the right set of indexes is a challenging process that has been researched for the past 50 years [32, 36]. There are various sophisticated index selection approaches, but for complex workloads, current approaches produce either solutions of high quality or provide low selection runtimes. Yet, they fall short of striking the right balance between both and further metrics, e.g., *Extend* [50] and *DB2Advis* [56] in the schematic Figure 1. Our index selection approach based on reinforcement learning (RL), *SWIRL*, bridges this gap by incorporating knowledge of vast amounts of workloads during training.

Motivation. By 2022, more than 75% of all databases are estimated to run in the cloud [22]. The increasing share of database deployments in cloud environments, especially in Software-as-a-Service (SaaS) scenarios, shifts the responsibility for effective physical database design to cloud vendors that maintain these systems. This development and the cloud's promise to reduce the total cost of ownership [15] allow for reconsidering how physical database design challenges are approached. The sheer number of systems to be maintained and dynamically changing workloads [37], which demand fast reactions by reconfigurations, require that optimized configurations, e.g., for indexes, can be determined quickly and efficiently. In SaaS scenarios, thousands of customers run similar workloads on similar schemata because

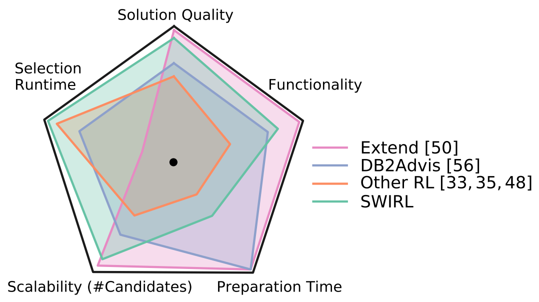


Figure 1: Index selection algorithms show weaknesses in different dimensions. *Functionality* means, e.g., supporting multi-attribute indexes or budget constraints.

such applications predefine schemata and workloads [2]. Therefore, the resulting physical database design decision problems are also – to some degree – similar.

However, knowledge about these characteristics is not utilized by state-of-the-art index selection approaches. These approaches determine solutions without relying on prior knowledge, often resulting in long runtimes or in degraded performance. Alternative concepts like reinforcement learning-based approaches can – if applied carefully – effectively exploit these characteristics and the fact that massive amounts of training data exist [18]: During training, an agent efficiently learns which indexes are beneficial under what circumstances for the predefined schemas. After training, and in contrast to state-of-the-art approaches, it does not need to enumerate possible solutions expensively but *infers suitable indexes almost instantaneously* based on the previously accumulated knowledge. While other approaches must account for complex effects, e.g., index interaction (see Section 2.1) by costly and iteratively testing multiple configurations, our approach *internalizes* such effects during training. Naturally, to gain this knowledge, extensive training is required, which is justifiable if efficient index configurations can be determined quickly later when the model is frequently applied.

Contributions. This paper offers the following contributions:

- We present an RL-based index selection approach matching the performance of the best competitors while its runtime outperforms the fastest (but not as good) algorithms.
- Our solution applies a sophisticated workload model to generalize to workloads with unseen queries and relies on *invalid action masking* to reduce training durations.
- We include the first survey and performance study for RL-based index selection and evaluate our approach with complex analytical database benchmarks on PostgreSQL in terms of training overhead, runtime, and performance.
- We provide an open-source implementation[1] that can be used to reproduce our results and adapted for further RL experiments with other physical database design challenges.

[1]SWIRL's source code: https://github.com/hyrise/rl_index_selection

2 BACKGROUND

This section introduces the index selection problem and its main challenges (Section 2.1) before we formalize it in Section 2.2. Afterward, we introduce the formal foundations of RL in Section 2.3.

2.1 The Index Selection Problem

Index selection describes the process of determining an optimal set of indexes for a workload under constraints, e.g., a storage budget or a specified maximum number of indexes [44]. We denote these constraints as *stop criteria*. In the following, we explain which factors make index selection a challenging problem. For more details, we refer to a recent experimental survey [32] on index selection.

Large solution space. For reasonably sized datasets and workloads, numerous options for indexation, i.e., index candidates, exist. The number of *relevant* index candidates depends on the number of attributes (real-world datasets can contain thousands of attributes [5]) accessed by the workload's queries and the maximal number of attributes per index (multi-attribute candidates are typically generated by permuting single-attribute candidates). Evaluating all candidate combinations is, in general, impractical as their number exceeds the number of attributes by orders of magnitude [59]. Hence, enumerating all solutions is infeasible.

Index interaction. During index selection, the candidates cannot be considered independent because indexes *interact* [51]: the existence of one index can affect the performance impact of another index. Thus, during every step of an index selection process, the currently existing indexes have to be taken into account. This fact requires frequent recomputations of the candidates' benefits because every index creation or removal might drastically impact another index candidate's potential benefit.

Quantifying index impact. Determining the potential performance impact of index candidates is essential for comparing them and choosing the most promising ones. Physically creating indexes and measuring their impact is theoretically possible, but long creation and execution times render this method infeasible. For this reason, index selection approaches typically rely on estimates instead of actual measurements. Some database systems offer *hypothetical* indexes that are not (entirely) physically created but only inexpensively simulated for such estimations. These hypothetical indexes are considered by the DBMS' optimizer (*what-if optimization* [13]) to generate query plans and cost estimations. Despite the relatively cheap simulation of hypothetical indexes, the cost estimation process is still a major contributor to the runtime of index selection algorithms [43]. While cost estimations may differ from actual execution costs to a large extent [7], they are typically the only feasible option for large workloads and still allow for comparing different index selection approaches [32].

2.2 Problem Formalization

We consider a workload characterized by N query templates (or query classes) and K involved attributes. Each query class n is represented by a set of attributes $q_n \subseteq \{1, ..., K\}$, $n = 1, ..., N$, that are accessed. Further, by I, we denote a given set of *index candidates*. A (multi-attribute) index $i \in I$ is characterized by an *ordered* set of attributes from $\{1, ..., K\}$. The width W of an index corresponds to the number of attributes it contains. W_{max} denotes the largest index width considered during index selection. The *required storage* of index i is denoted by m_i, $i \in I$. A *selection* of indexes is denoted by the subset $I^* \subseteq I$. The *costs* to execute a query of class n depend on the chosen selection of indexes I^* and are denoted by parameters $c_n(I^*)$. Assuming that queries of class n occur with *frequencies* f_n, $n = 1, ..., N$, the *total workload costs* C depend on I^* and amount to

$$C(I^*) := \sum\nolimits_{n=1,...,N} f_n \cdot c_n(I^*). \qquad (1)$$

The goal is to determine an index selection $I^* \subseteq I$ such that $C(I^*)$, cf. (1), is minimized and a given *storage budget* B (stop criterion) is not exceeded. Using binary variables x_i, which indicate whether a candidate index $i \in I$ is part of the selection I^* (1 yes, 0 no), i.e., $I^*(\vec{x}) := \bigcup_{i \in I: x_i = 1} \{i\}$, the total storage used by a selection $I^* = I^*(\vec{x})$ amounts to $M(I^*(\vec{x})) := \sum_{i \in I} m_i \cdot x_i$ and the *index selection problem* can be mathematically defined by:

$$\underset{x_i \in \{0,1\}, i \in I}{\text{minimize}} \quad C(I^*(\vec{x})) \quad \text{subject to} \quad M(I^*(\vec{x})) \leq B. \qquad (2)$$

In this context, a restriction on the number L of selected indexes would correspond to a cardinality constraint, $\sum_{i \in I} x_i \leq L$.

2.3 Reinforcement Learning

Reinforcement Learning (RL) covers a group of algorithms to solve decision problems. Those problems are characterized by processes that repeatedly allow an agent to perform an action a_t of its available actions A given a current state $s_t \in S$ [55]. The state describes the properties of the environment the agent is currently observing. Depending on the problem and the RL algorithm, A and S can be either discrete or continuous and have an arbitrary number of dimensions. After performing the chosen action, a new state s_{t+1} is reached, and the process repeats. To provide agents with feedback on whether the action was chosen well, they receive a feedback signal, the reward r_t after each decision. The simulation might end at some point, leading to episodes of finite length characterized by the states, the agent's decisions, and the following rewards. The problem is to find an optimal policy, which maps states to actions, concerning the discounted future long-term reward for a starting state at time t:

$$G_t = \sum\nolimits_{k \geq 0} \gamma^k \cdot r_{t+k}. \qquad (3)$$

Rewards are discounted to take into account that further progression in the decision process becomes less predictable. Low values of the discount factor $\gamma \in [0, 1)$ motivate the agent to act more greedily and consider possible long term rewards less.

To implement an RL system, the agent needs to estimate the best expected value of G_t correctly. The Q-value is the expected value of G_t given a certain state s_t and the chosen action a_t, i.e.,

$$Q(s, a) = \mathbf{E}[G_t | s_t = s, a_t = a]. \qquad (4)$$

The Q-value, as specified in (4) can be reformulated iteratively, as it incorporates the Q-value of the following state and its long term reward, G_{t+1}. This allows to learn an estimator for the Q-value using the Bellman-update, given an observed state s_t, a performed action a_t, the reward r_t, and the follow-up state s_{t+1}:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta \cdot (r_t + \max_{a_{t+1} \in A} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)),$$

where $\eta \in (0, 1)$ is the learning rate. Higher η values increase the update size but decrease the stability of the estimation. In this setup, the agent keeps a matrix to store and update the Q-value for each observed combination of s_t and a_t. This representation allows it to derive a policy from the Q-estimation, by greedily choosing the action a that maximizes $Q(s, a)$ in the current state s. Actions are randomly chosen with a specified probability ε to ensure that the agent does not always choose the same actions (and leaves beneficial states unobserved). Further, instead of using tabular Q-values, a generic function approximator, such as an artificial neural network (ANN), can represent Q. In this setup, the difference between the network's estimation for the Q-value and the computed target value $r_t + \max_{a_{t+1} \in A} Q(s_{t+1}, a_{t+1})$ is minimized at each learning step.

This concept can be further expanded with policy gradient methods, which do not derive a policy from the learned value estimations but instead keep a parametric policy $a_t = \pi_\Phi(s_t)$. By adjusting Φ, the mapping from s_t to a_t is changed. Adjusting Φ usually relies on the policy gradient theorem, which allows to improve expected rewards via Φ only based on past observations. Our experiments rely on Proximal Policy Optimization (PPO) [52]. PPO offers the advantage of adjusting the learning rate automatically. Correctly adjusting the rate stabilizes the learning process by avoiding drastic changes in the agent's behavior and improves overall performance.

The index selection problem can cause states in which not all actions within A are applicable, i.e., not all indexes can be created, for instance, due to budget constraints, cf. Equation (2). Such invalid actions could be modeled by assigning large negative rewards to such actions. We will incorporate another feedback mechanism: action masking [28]. In this approach, the agent receives the allowed actions as input and is structurally enforced to select an element within this set. This technique shortens the learning process, as this element of the decision does not have to be represented within the agent's policy anymore. In addition, peak performance can be increased as well [28]. Such efficiency considerations are essential if the overall action space consists of many actions but only a few of these actions are allowed in a given state. If action masking were omitted, the agent would have to explore many of the invalid actions to recognize which are allowed (depending on the state).

3 RELATED WORK

In the following, we first introduce the state-of-the-art index selection approaches that are later used for evaluations. Afterward, we discuss existing RL-based approaches and their differences to our solution before we conclude with requirements for a competitive RL-based index selection approach in Section 3.3.

3.1 Existing Index Selection Approaches

There is a multitude of existing index selection algorithms, the first dating back to 1971 [36]. Most techniques either iteratively *add* indexes to an empty start configuration or *reduce* a comprehensive start configuration step by step. Reductive approaches [9, 57] often result in very long runtimes because many iterations are necessary to comply with the specified constraints [32]. For this reason, we chose three additive algorithms for comparison that showed either the fastest runtimes or were able to determine the best (or even optimal) solutions during a recent experimental evaluation study [32]: *AutoAdmin* by Chaudhuri and Narasayya [12], *DB2Advis* by Valentin et al. [56], and *Extend* by Schlosser et al. [50]. We refer the interested reader to [32] for brief descriptions of these algorithms. The referenced performance study has shown that none of the existing approaches meets both performance criteria - high quality *and* fast solutions - for complex analytical workloads.

Even though adaptive indexing [30] or database cracking [29] are valuable indexing techniques, they are not considered in this work. These techniques typically target column stores while we investigate generally applicable index selection approaches. Further, they are not available for database systems that offer publicly available hypothetical index interfaces, such as PostgreSQL, which would hinder a fair comparison and distract from this work's main focus, i.e., comparing existing with RL-based index selection approaches.

3.2 RL-based Index Selection Approaches

Lately, a couple of RL-based index selection approaches have been presented as an alternative to existing rule- and enumeration-based heuristics. The RL approaches show promising results with regards to some aspects but also have different limitations. Next, we describe these approaches, including their limitations, and highlight differences to our work. Table 1 compares the approaches along different dimensions: (i) whether multi-attribute indexes are supported, (ii) the index selection's stop criterion (see Section 2.1), (iii) the availability of an open-source implementation for further experiments and the reproduction of results. Further, (iv) we examined how the listed publications incorporate the representation of the workload at hand. We compare whether or not they can generalize to to (v) unknown workloads that include completely unseen query classes. Lastly, (vi) we mention how the approaches were evaluated.

Sharma et al. were the first to present an RL-based index selection approach, *NoDBA*, capable of creating single-attribute indexes in 2018 [53]. They evaluate their ideas with queries that filter TPC-H's `LINEITEM` table on multiple attributes. The model represents the workload as a matrix that contains the selectivity of every attribute for every query if the query is filtered on this attribute. This model makes the generalization to unknown queries theoretically possible even though it is not discussed in the publication. Varying frequencies of the queries are not considered but could be modeled by repeatedly adding the same query to the state matrix, which would be unfeasible for larger workload sizes. Their approach does not consider other operators (apart from selection predicates) for index selection. Naturally, this is a significant limitation as other operators, e.g., joins and aggregates, are responsible for a large amount of the overall runtime in typical database workloads [19, 40]. The authors provide an open-source implementation[2] of their work.

Sadri et al. present *DRLinda* for cluster databases [48, 49]. While multi-attribute indexes are not supported, considering multiple instances in a database cluster sets a different focus, further complicates the problem, and is a differentiator to all other – including our – approaches. The workload is represented in three ways: (i) an access matrix that encodes for every attribute whether or not it is accessed in a query, (ii) an access vector that counts how often every attribute is accessed in total, and (iii) a selectivity vector that holds selectivity $= \frac{\text{\# unique values}}{\text{\# of rows}}$ for each attribute. This model could enable the agent to generalize to unknown workloads which is not discussed or evaluated in the publication. There is neither a public implementation nor an evaluation that compares with state-of-the-art index selection approaches available. For our evaluations (Section 6), we have reimplemented *DRLinda* and included it in our open-source repository, see Footnote 1 on Page 2.

Lan et al. propose another RL-based solution that allows identifying multi-attribute indexes [33]. With increasing index widths (W), the number of candidates increases drastically; for workloads with hundreds of attributes, thousands of relevant 3-attribute

[2]Source Code for NoDBA [53]: https://github.com/shankur/autoindex

Table 1: Comparison of different reinforcement learning-based index selection approaches.

		NoDBA [53]	DRLinda [48]	Lan et al. [33]	SmartIX [35]	DRLISA [58]	SWIRL
(i)	Multi-attribute indexes	No	No	Yes	No	Unspecified	Yes
(ii)	Stop criterion	# Indexes	# Indexes	# Indexes[5]	# Steps	No improvement	Budget
(iii)	Implementation available	Yes	No	Yes	Yes	No	Yes
(iv)	Workload representation	Yes	Yes	None	None	Unspecified	Yes
(v)	Generalization new queries	+	++	–	–	Unspecified	+++
(vi)	Evaluation	TPC-H scans	TPC-H partly	TPC-H	TPC-H	YCSB	TPC-H/DS, JOB

indexes exist [32]. The set of available actions usually comprises one action per index candidate for RL-based approaches. The authors propose five heuristic rules that serve as a preselection to reduce the number of index candidates (and consequently actions) and enable the selection of multi-attribute indexes. Excluding index candidates in advance may limit the potential solution quality [50]. The model does not implement workload representation. For this reason, it cannot generalize and is only suitable for the exact training workload. The evaluation is conducted on 14 TPC-H queries and the implementation[3] is publicly available.

Licks et al. present SmartIX: A database indexing agent based on reinforcement learning [35], provide an open-source implementation[4], and an extensive introduction to RL in the context of index selection. Their implementation is not capable of creating multi-attribute indexes and does not include workload representation. In consequence, generalization is not possible. While the approach is evaluated with the full TPC-H benchmark and compared to a multitude of other approaches, the best-performing state-of-the-art approaches [32] are not included. Their training procedure trades off long trainings (days) against avoiding inaccuracies of cost estimations: SmartIX derives query runtimes from actual query executions instead of what-if estimations.

Yan et al. target NoSQL databases with *DRLISA* [58], which is a differentiator to all other approaches. Further, it is the only approach that stops independently of the number of indexes or a storage budget. Instead, it terminates if no further performance improvement can be realized. According to the paper, the RL model takes a workload representation as input, but we could not find further details regarding this representation. Consequently, generalization may or may not be possible. The authors do not mention any publicly available implementation and evaluate their approach with the YCSB (Yahoo! Cloud Serving Benchmark) [14].

3.3 Research Gap: Requirements for Competetive RL-based Index Selection

Based on the motivation stated in Section 1, the limitations of state-of-the-art (Section 3.1), and RL-based approaches (Section 3.2), we derive the following requirements for a competitive RL-based index selection approach. In terms of performance indicators, a potential approach should determine solutions that are competitive *(R-I)* with the solutions of the best state-of-the-art algorithms, e.g., *AutoAdmin* or *Extend*, while the computation of such solutions should be significantly faster *(R-II)*, for instance, comparable to the computation times of *DB2Advis*. At the same time, the training duration of the proposed model should not outweigh *(R-III)* the advantage gained during application time.

None of the existing RL-based approaches offers all the functionality that is commonly expected from index selection approaches, cf. Table 1. Multi-attribute indexes are widely deployed in real-world environments [21]. Thus, we require a powerful new RL-based solution concept to support multi-attribute indexes *(R-IV)*. Besides, the proposed solution should accept storage budgets *(R-V)* as a stop criterion as they allow more fine-grained solutions compared to criteria targeting a fixed number of indexes [32]. Further, an index selection algorithm based on RL should be able to generalize *(R-VI)* to unknown workloads, at least to a reasonable extent. Otherwise, retraining the agent for every workload change would be necessary, which is – given the training durations for RL approaches – unrealistic and would make the approach much less attractive.

There is currently no RL-based index selection approach that competes with state-of-the-art algorithms in terms of *R-I* to *R-III* and unifies the functionality demanded by *R-IV* to *R-VI*. Lastly, meeting the requirements should be evaluated on multiple complex analytical workloads against competitive state-of-the-art approaches, which has not been done for existing RL approaches.

4 SWIRL: SELECTION OF WORKLOAD-AWARE INDEXES USING RL

In this section, we detail our approach (*SWIRL*) that tackles index selection with RL. We start with an overview in Section 4.1. Afterward, in Section 4.2, we explain how we model the index selection problem in an RL-compatible fashion and handle the resulting complexity; therefore, we answer how the environment's state, i.e., the queries, the budget, and the active indexes, is represented. Lastly, we clarify the applied training procedure that enables efficient training and determining solutions of high quality.

4.1 Overview

Figure 2 depicts an overview of the entities involved in the RL-based index selection process and how they interact. The process is divided into three phases: (i) preprocessing: training and testing workloads are generated, index candidates are determined, and the workload representation model is prepared; (ii) training: the agent learns which indexes are valuable for the provided queries as well as how these indexes interact, and (iii) application: the agent applies the trained model to determine indexes for certain workloads. During (ii) and (iii), the agent iteratively selects indexes for a given workload. This process forms the Markov decision process that the RL algorithm observes, cf. Section 2.3.

Preprocessing. ① The user, e.g., a database administrator (DBA), can specify a set of *representative*[6] *queries*. The potential impact of the specified query set is discussed later in Section 4.2.2.

② Afterward, *index candidates* are generated based on the input *schema's* attributes and the set of representative queries. Restricting the set of index candidates to relevant ones is crucial [32, 33] since index candidates correspond to the agent's actions, and too large action spaces complicate the agent's process

[3]Implementation of Lan et al. [33]: https://github.com/rmitbggroup/IndexAdvisor
[4]SmartIX's [35] experimental setup: https://doi.org/10.5281/zenodo.3254967
[5]Due to space restrictions, budget results were not reported by the authors.

[6]Relying on representative queries is in accordance with other learned approaches [27].

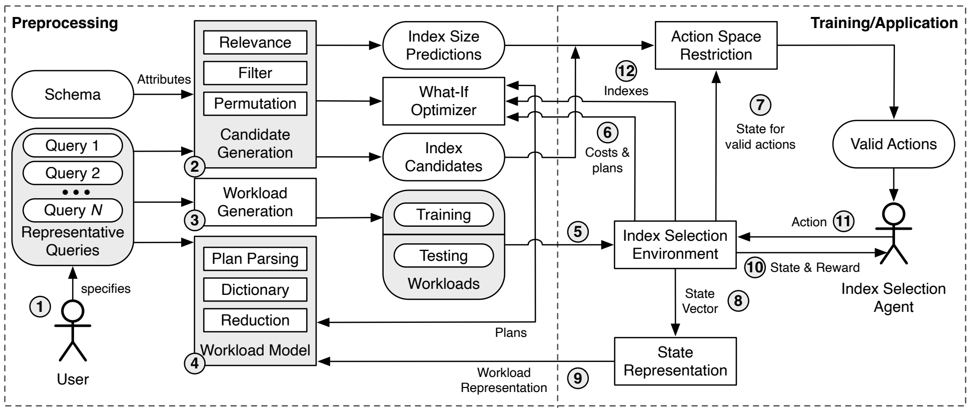


Figure 2: Overview of RL-based index selection process. Divided into preprocessing, training, and application phase.

of determining reasonable solutions and can increase training durations. At the same time, candidates should not be limited too much; otherwise, solutions of high quality cannot be determined [50]. For this reason, not all but most attributes of the schema (and their permutations) should become index candidates. By default, our system generates all *syntactically relevant* index candidates (except for indexes on very small tables, $n < 10\,000$). *Permutations* are generated according to a user-specified admissible index width (W_{max}). The candidate generation also prepares predictions of the index sizes for every candidate based on the estimates of a *what-if optimizer.*

③ Based on the representative queries, random *workloads are generated* as follows. A workload consists of (a subset of) the representative queries and assigns a random frequency to each query. Thereby, we create variability and ensure that the agent has to handle different query-frequency pairs during training to anticipate a wide variety of workloads later during application.

The created workloads are split into training and test sets, and it is guaranteed that the test set contains only workloads that are not part of the training set. Besides, it is possible to specify that a certain number of the representative queries are not part of any training but only of test workloads to guarantee pure out-of-sample predictions. By doing so, we can investigate that agent's capability to generalize to unknown queries and workloads.

④ Machine learning models are usually provided with numerical features. The *workload model* is responsible for creating workload representations, i.e., transforming information about the queries of the current workload to a numerical representation that can be passed to neural networks. This process is crucial because, without a representation, unknown queries cannot be handled. Details are presented in Section 4.2.2.

Training. During training, the agent learns which indexes are beneficial in which situations by efficiently trying out many index configurations for different workloads. Thereby, it is also able to implicitly discover and *internalize* complex effects like the relationship between different candidates, i.e., index interaction, without these effects being explicitly specified or modeled.

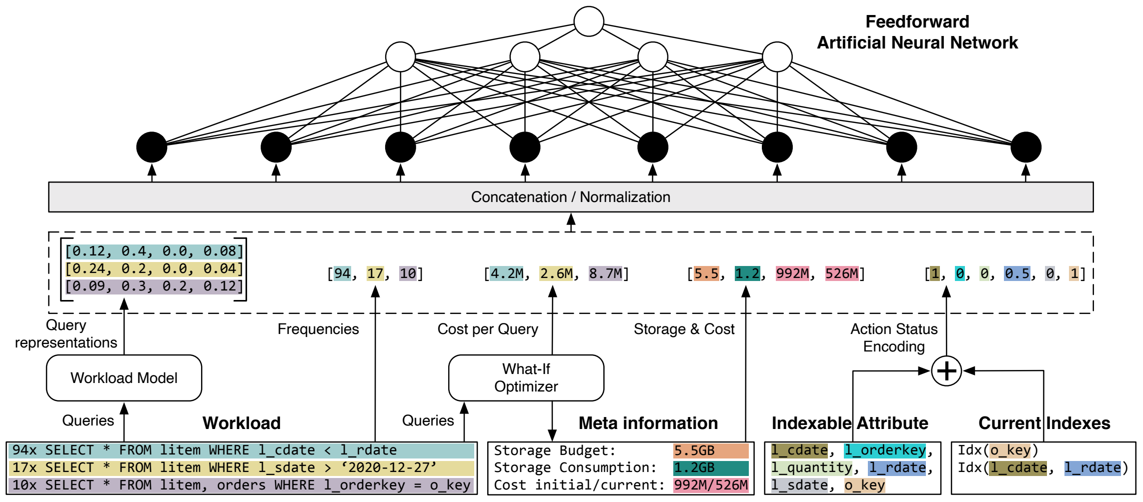
The central components of the RL process are the *index selection environment* and the *index selection agent.* The agent is stateless and provides numerical actions that correspond to the creation of indexes in the environment. The stateful environment encapsulates the DBMS: it translates and implements the agent's actions in the DBMS, determines their consequences, rewards the agent (Section 4.2.4), informs it about the environment's state (Section 4.2.1), and abstracts further parameters, e.g., the budget.

⑤ The environment retrieves a new workload (③) for every training episode. Within one episode, the workload is constant. ⑥ Subsequently, the costs and plans for every query of the current workload are requested from the what-if optimizer given the current index configuration, which is usually empty for the first step of an episode. ⑦ Then, the agent's action space is restricted to contain only actions that are valid for the environment's current state. These restrictions are a major factor for converging as quickly as possible and allowing thousands of index candidates without limiting the candidate set a priori. Actions can be limited based on the current workload, the remaining budget, or previous actions; see Section 4.2.3 for details. This procedure is a main differentiator to existing RL-based index selection approaches. ⑧ For the state representation, the current state of the environment, e.g., the remaining budget, current costs, and active indexes, is translated to numerical features so that it can be passed to the agent. This process includes the retrieval ⑨ of the workload representation from the workload model. ⑩ The environment's state and, if available, a reward are passed to the agent, which, in return, ⑪ reacts with an action under consideration of the currently valid actions. ⑫ After the environment implemented the agent's actions by creating indexes via the what-if utilities, the process is continued at step ⑥ until there are no valid actions left, e.g., if the budget is exceeded or a user-specified maximum number of iterations has passed. After a configurable number of iterations, the ANN (cf. Section 2.3) is updated to reflect the observations collected during the passed iterations.

Application. After training, our model is applied as follows. At ⑤, the actual workload is received instead of training workloads. Starting with an empty index set, the agent repeatedly *evaluates* the fitted ANN to subsequently select the action a_t ⑪ with the highest estimated reward (the best index) for a state s_t ⑩. Choosing a_t leads to a new state s_{t+1} for which a_{t+1} is determined until the budget is exceeded. This procedure can particularly be applied to any *unknown* workload as the ANN can even be evaluated for unseen s_t. Note, the application of our model is fast compared to state-of-the-art non-RL approaches since (i) interactions with the what-if optimizer are not necessary and (ii) due to the trained ANN, only simple evaluations remain to be performed.

4.2 Model

Below, we explain how we model the index selection problem for RL. We elaborate on how information about the environment, e.g., the workload and current index configuration, is transformed into a vector-based representation, i.e., how we represent the



```
94x SELECT * FROM litem WHERE l_cdate < l_rdate
17x SELECT * FROM litem WHERE l_sdate > '2020-12-27'
10x SELECT * FROM litem, orders WHERE l_orderkey = o_key
```

Figure 3: State representation for a simplified example workload. The numbers are example values for demonstration.

environment's *state*. Afterward, we discuss how the workload is represented, how changes of the index configuration are modeled as actions of the agent, and how the agent is rewarded.

4.2.1 State Representation. Figure 3 shows how we encode the index selection problem with a simplified sample workload. The sample contains 28 features distributed over 7 vectors, enclosed by a dashed box. The number of necessary features to effectively represent a particular instance of the index selection problem largely depends on the number of query classes in the workload, their complexity, and the number of indexable attributes; details are covered later in this section.

The cost and storage information contained in the state representation can be based on actual measurements or on estimates obtained from a what-if optimizer. While the latter option offers only estimates, it is much faster. As state representations must be updated during training for each of the agent's steps, we rely on the what-if-based estimations; actual execution time measurements are simply impractical. The state representation can be divided into three aspects: the workload, meta information (e.g., budget), and the current index configuration. All of these aspects influence which indexes of all possible indexes are beneficial: for two different workloads, completely different index configurations might lead to the best performance.

Workload representation. Workloads can either contain the same query classes, but their frequencies can differ, or the query classes themselves can differ. The workload representation must reflect both cases. For a workload with N query classes and a specified representation width of R (in Figure 3, $N = 3$, $R = 4$), the workload representation consists of (i) N numerical vectors of length R that represent the queries' contents, (ii) a vector of length N with a numerical value for each query's frequency, and (iii) another vector that contains the estimated execution cost per query (N values) given the currently active index configuration. Representing the queries' contents is crucial for our approach and a main differentiator to other approaches; without it, the agent cannot learn about the structure of queries, recognize similarities, and, in the end, generalize to unseen queries, see Section 4.2.2.

Even though the neural network structure is fixed, a model that was trained with workload size N can always be utilized to determine index configurations even if the workload size is different, e.g., $\tilde{N}$, during inference time. If $\tilde{N} < N$, *padding* can be applied, i.e., query representation, frequency, and cost per query are set to 0 for $N - \tilde{N}$ queries. Otherwise, if $\tilde{N} > N$, a representative set of the workload with size N must be created. Such a set can always be found, e.g., by focusing on the most relevant queries and summarizing similar queries; *workload compression* [11, 16] has been effectively used for index selection in the past. Also, query clustering approaches exist to reduce the total query count [37]. Choosing N to be sufficiently large in the beginning can avoid the need for workload compression altogether or, at least, decrease the possible information loss caused by it. Choosing N sufficiently small allows controlling the model's complexity.

The meta information contains four scalar features regarding storage and workload cost: (i) a value for the currently specified[7] storage budget (B), (ii) the current storage consumption based on the what-if optimizer's index size predictions, and (iii) the initial (without any indexes) and (iv) current cost (C) for executing the entire workload, cf. Equation (1).

The current index configuration encodes for every indexable attribute (see Section 4.1) whether an index is present or not. In the simplest case, with a maximum index width of $W_{max} = 1$, this information can be represented by a binary vector as every index can exist once or not at all. Encoding the index configuration is more challenging if multi-attribute indexes are admitted because there can be millions of index candidates, e.g., for TPC-DS with $W_{max} = 4$, $|I| \approx 1.3$ million [32]. If we used a binary vector as above, we would increase the number of (very sparsely populated) features by the number of index candidates which would be infeasible. Since wide indexes occur in real-world systems [21] and limiting W_{max} can harm performance [50], decreasing the dimensionality by limiting the number of candidates is also not an option. Hence, we encode the information on the current index configuration for each indexable attribute separately to avoid large feature spaces: The value in the vector is incremented by $1/p$ for every index that contains the corresponding attribute. p refers to the position in the index. For example, for `Idx(l_cdate, l_rdate)` `l_cdate`'s p is 1 (vector value 1/1) and `l_rdate`'s p is 2 (value 1/2). If a further index `Idx(c1, c2, c3, l_cdate)` would exist, `l_cdate`'s vector value would be $1.25 = 1 + 1/4$. Modeling the current index configuration like that – in contrast to a binary vector – implicates some loss of information: instead of encoding which exact indexes exist, we encode to which degree attributes

[7]Storage budgets are externally specified, e.g., by DBAs or external meta models.

are covered by indexes. However, according to our evaluations, the agent is able to handle this encoding. In addition, the *index selection environment* still maintains the full information that is, e.g., necessary for applying action masking (Section 4.2.3).

Concatenation and normalization. Before the presented state information is passed to the neural network, the vectors are concatenated, and the contents are normalized with StableBaseline's `VecNormalize` class, which normalizes values X to $\tilde{X}$ using their moving average $\bar{X}$ and the variance $\sigma^2(\cdot)$ as follows ($\varepsilon := 10^{-8}$ prevents possible divisions by zero):

$$\tilde{X} = (X - \bar{X})/(\sigma^2(\bar{X}) + \varepsilon)^{0.5}.$$

Normalization is applied to improve the network's learning behavior. The used activation function `tanh` suffers from vanishing gradients on large inputs. This effect can be avoided by normalization with zero mean and a variance of one [23].

Number of features. The *number of features F* passed to the model, except for the meta information (*MI*), is not fixed and varies with the problem instance at hand and the configuration. It amounts to:

$$F = N \cdot R + N + N + MI + K. \tag{5}$$

The number of query classes in the workload (N) largely affects the size of the frequency, cost per query, and query representation vectors. Also, the representation width (R) is important for large workloads as it is multiplied with N. If the workloads that are passed to the model contain complex, unlike queries, R must be chosen large enough to capture different queries and their similarities properly. Lastly, large database schemas can result in hundreds of thousands of indexable attributes. Hence, we only consider attributes (cf. K) that are accessed by at least one query. Otherwise, it could result in too many features only to represent the current index/action status. For a TPC-DS scenario with a workload size of 30 query classes and a representation width of 50, there are $30 \cdot 50 + 30 + 30 + 4 + 186 = 1\,750$ features for 186 indexable attributes, cf. Equation (5).

4.2.2 Workload Modeling and Query Representation. One of the main goals of our approach – and a major differentiator to existing approaches – is to be able to handle query templates that were not part of the training workloads. Of course, these templates should not differ entirely but be reasonably similar to the query templates used during training. Thus, the set of training queries should roughly capture the workload expected at application time.

The desired capability to handle unknown queries requires us to set such queries into context with known ones, which adds complexity to the model: details of the queries must be encoded such that the agent can incorporate them into its decision making, i.e., a detailed representation of the workload respectively of its queries is necessary. This representation must be compact enough to avoid feature explosion and contain enough detail to distinguish queries properly. Besides, the task of creating the representation must not be too complex as it would further increase training durations.

Figure 4 depicts how our representation model is built and how representations are inferred. We build representative plans from the set of representative queries by utilizing the what-if optimizer and index candidates. For every query, the what-if optimizer is repeatedly invoked to generate various plan alternatives based on different index configurations. Theoretically, the query representation could be built entirely on the queries' SQL strings, but execution plans contain more details, information about index usage and might change with the agent's actions, i.e., index decisions. The representative plans are passed to the representation model.

The operators of every plan that are relevant[8] for index selection are transformed into a text representation. For example, under the presence of an index on `TabA.Col4`, a text representation `IdxScan_TabA_Col4_Pred<` might be generated. The text representations for all representative plans are stored in the *operator dictionary*, which assigns an ID to every distinct operator's text representation. These IDs are used in the next step to construct a *Bag Of Operators (BOO)* (cf. bag-of-words model [10, 25]), i.e., a numerical representation of the operators of a query.

Dimensionality Reduction. The BOO could be made part of the state representation. However, using the BOO without further processing would result in numerous, very sparsely populated features per query. For the TPC-DS benchmark's query templates, we count 839 distinct relevant operators, which we would need to incorporate for every query of the workload, i.e., N times. Consequently, we apply a dimensionality reduction step. Based on the BOO representations of all representative plans, we build a Latent Semantic Indexing [17] model to reduce the feature count. Other approaches, e.g., random projections [3] or principal component analysis, could also be used to reduce dimensionality.

Choosing the number of features representing a query – the representation width R – is a tradeoff decision. Larger values for R increase the model's size and training times. Smaller values can lead to insufficient workload representations. The library used for the LSI model (see Section 5) indicates how much information is lost when the model is built. We experimented[9] with varying values for R and found that for the examined workloads and $R = 50$, approximately 10% of information is discarded. Additionally, since the agent's performance does not improve much when choosing higher R values, $R = 50$ seems to be a reasonable choice.

Alternative workload modeling approaches. In the literature, alternative workload modeling approaches [18, 54] exist. These approaches differ because different use cases guided their design. For example, the workload modeling of Ding et al. [18] builds the input for a classifier that determines which of two

[8]We focus on operators that are affected by indexes. Source code: https://github.com/hyrise/rl_index_selection/blob/main/swirl/bag_of_operators.py

[9]Representation width experiments: https://github.com/hyrise/rl_index_selection/tree/main/experiments/representation_width

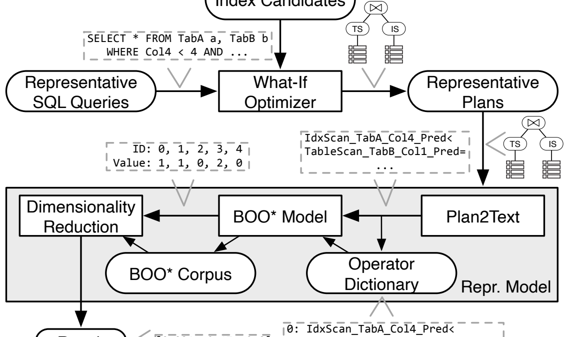


Figure 4: Workload representation example. *BOO refers to Bag Of Operators.

query plan alternatives is cheaper. The modeling approach featurizes query plans based on physical operators, too. However, the featurization is schema-agnostic without explicitly referring to accessed tables or attributes. Instead, it encodes and aggregates different information, e.g., the amount of work done per operator type. Ding et al.'s workload modeling is reasonable for their use case. In contrast, our approach is specifically designed for an RL-based index selection agent. Our agent's actions are directly related to particular operators and attributes. Therefore, the explicit knowledge of how attributes and operators occur in the workload is necessary for good index selections.

Simplifications and extensions. If unknown queries are not required to be handled, the modeling can be simplified. For example, Hilprecht et al.'s approach [27] for RL-based partitioning models the workload by only encoding the frequency share of known queries in the current workload. We argue that even in cloud scenarios (see Section 1), with a predefined set of standard queries, the assumption that no unknown queries will occur is invalid because customers usually formulate additional queries.

The exact values for query template parameters can influence the execution costs and resulting query plan. If only the costs differ, the workload model's query representation is identical, but the corresponding value in the query cost vector (cf. Section 4.2.1) reflects the difference. Query plans (for the same query with different parameters) that contain different physical operators can be expressed by our BOO model. However, semantically equivalent but structurally different (e.g., operator order) query plans could result in the same BOO. This issue could be avoided by also encoding the structure of the plan, e.g., by including the preceding operator(s) in the representation of subsequent operators. Since we did not encounter any such issues, the current version of our model does not encode the preceding operators.

4.2.3 Actions. The action space determines how the agent can act, i.e., for index selection, which indexes the agent can create. Our model employs a discrete action space A, where every action is a unique (multi-attribute) index candidate, i.e., we set $A := I$. The index candidates are determined during preprocessing, cf. Section 4.1; the existence of thousands of multi-attribute index candidates is not rare for real workloads and datasets [50].

Carefully designing and handling the action space is crucial for two reasons. (i) As stated in Section 2.3, the training efficiency depends on the number of available actions. For complex combinatorial problems, more available and dependent actions (index interaction) further complicate the problem. Simply limiting the index candidates a priori might reduce the size of the action space but can also negatively impact the quality of the determined index configurations as shown by Schlosser et al. [50]. In addition, (ii) particular actions might be invalid at particular states. Comparable to the rules of chess where moving pawns across three squares is forbidden, the index selection process also follows specific rules: we can consider the repeated creation of an existing index or exceeding the storage budget as a breach of the rules. In RL, rules are usually enforced by large negative rewards to teach the agent the invalidity of certain actions. However, everything that must be learned can potentially increase the training duration and harm performance.

We utilize *invalid action masking* [28] to (temporarily) disable actions given the current state, thereby guiding the agent to only consider a subset of all available actions. Valid actions must be updated before every step, and there are four reasons why actions could be marked invalid, which is also demonstrated in Figure 5.

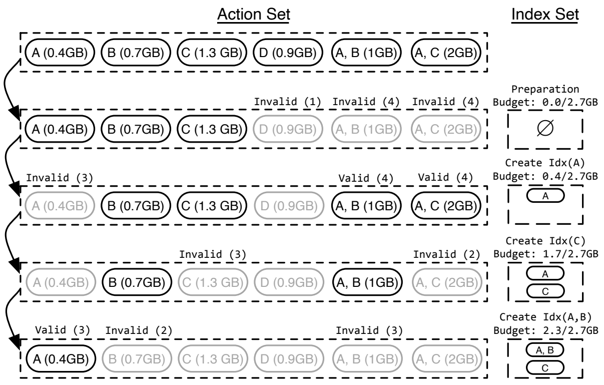


Figure 5: Example for invalid action masking. Numbers in braces on (in)validation actions indicate the reasons for the change. Creating an index (A,B) drops the index (A).

(1) **Index candidate irrelevant for workload**. Before the agent's first step, we check for every index candidate whether it is syntactically relevant for the workload at hand, i.e., whether all of the index's attributes occur in the workload.
(2) **Index would exceed the budget**. Before every step, we calculate for every index candidate whether its creation would exceed the storage budget, given the current consumption.
(3) **Index already existing.** After choosing an action a, it is marked invalid such that it cannot be chosen again. Later, action a can be marked valid again, e.g., due to choosing actions that are associated with multi-attribute indexes.
(4) **Invalid precondition.** Before the first step, all multi-attribute indexes are masked as invalid. Only after the agent chose an index (A), all multi-attribute indexes with A as the first attribute, e.g., indexes (A, B), (A, C), are made valid. We follow the intuition of Chaudhuri et al. "that for a two-column index to be desirable, a single-column index on its *leading* column must also be desirable" [12, p. 151] and the procedure of the *Extend* index selection algorithm by Schlosser et al. [50]. Three- and n-attribute indexes are masked accordingly.

DBAs might favor preventing the model from handling indexes that were manually created based on domain knowledge or that guarantee service-level agreements (SLAs). Such indexes can be made inaccessible by invalidating actions affecting them. The effect of action masking on the number of available actions in realistic circumstances is demonstrated later in Section 6.3.

4.2.4 Reward. For each action, the agent receives a reward, r_t, cf. Equation (4), that incentivizes beneficial actions and guides the learning process. There are multiple options for building reward functions for index selection that could consider relative or absolute cost impacts of indexes, their storage consumption, and their validity (see Section 4.2.3). Absolute cost impacts have the disadvantage that these might largely differ for similar actions for different workloads and do not account for the required storage. For this reason, in line with *Extend* [50], to consistently optimize the usage of storage in each step/state t, we consider the additional relative benefit (reduction of workload costs, cf. Equation (1)) of an index selection I_t^* per additional utilized storage as reward, cf. Section 2.2:

$$r_t(I_t^*) = \frac{(C(I_{t-1}^*) - C(I_t^*))/C(\emptyset)}{M(I_t^*) - M(I_{t-1}^*)}.$$

In contrast to other approaches, it is unnecessary to punish invalid actions with negative rewards due to action masking. Our implementation allows defining alternative reward functions.

4.2.5 Miscellaneous. To prevent overfitting, we monitor the model's performance with workloads that are different from training and testing set every few thousand steps. If the moving average of the performance stops improving, we record the model's current state. Also, we employ extensive caching of cost requests (cf. Section 5), which largely impacts the training duration, see Section 6.3.

5 IMPLEMENTATION

In this section, we detail the implementation of *SWIRL*. Our open-source implementation, see Footnote 1 on Page 2, to train, evaluate, and adapt the presented approach is written in Python 3 and should facilitate further experiments with RL-based approaches for index selection or physical database design in general.

Database. Throughout the paper, we use the database system PostgreSQL and HypoPG [47] for what-if optimization. We rely on Kossmann et al.'s [32] index selection evaluation framework's cost evaluation component that encapsulates the retrieval of query plans, query costs, index information and enables us to create and drop hypothetical indexes. For this reason, the proposed approach is not tightly coupled to PostgreSQL, but support for other database systems that also offer what-if optimization could be added. Usually, cost estimation requests are responsible for the majority of the runtime of index selection algorithms. Therefore, the cost evaluation component implements a cache for such requests, which is indispensable for efficient training procedures, see Section 6.3.

Model. For the implementation of the RL algorithm (PPO), we use Stable Baselines [26] versions 2 and 3 that rely on Tensorflow, respectively PyTorch. The agent interacts with a database environment that is implemented according to OpenAi's gym [8] interface. The latent semantic indexing model used for workload representation (Section 4.2.2) is built with Gensim [46].

The model's hyperparameters that are displayed in Table 2 were experimentally determined. The gamma value appears low compared to other problems that are traditionally solved via RL. A value in the lower range increases the agent's greediness, while it is still chosen high enough to allow long term considerations in the relatively short episodes of the index selection problem. We have demonstrated in Section 4.2.1 that the number of features depends mainly on the workload size and the used representation width. Therefore, it might be necessary to adapt the network architecture for larger problem instances before training; the displayed size was sufficient for the evaluated workloads.

Flexibility. Our implementation is divided into multiple components, e.g., the reward determination, state representation, or the maintenance of the action space are handled by components. Hence, alternative implementations for these components can be added to evaluate different problem modeling strategies and design decisions experimentally. Most parameters, e.g., the workload size, maximum index width, or utilized reward function, can be configured via JSON configuration files. Furthermore, the concept of our approach is not tailored to index selection but could be extended to other physical database design problems, e.g., automated compression selection or partitioning, as long as the impact of varying configurations can be determined and transformed to a reward.

Table 2: Hyperparameters for our PPO (Section 2.3) model.

Hyperparam	Value	Hyperparam	Value
Learning rate η	$2.5 \cdot 10^{-4}$	Discount γ	0.5
Batch size	2048	Clip Range	0.2
ANN Layer Structure for Q and π	256-256	Policy	MLP

6 EVALUATION

This section evaluates our approach, assesses whether it fulfills the requirements formulated in Section 3.3, and compares its performance to state-of-the-art index selection algorithms.

After describing the experimental setup in Section 6.1, we evaluate the solution quality of the index configurations identified by *SWIRL* (Section 6.2) and compare the necessary training durations in different contexts (Section 6.3), which also includes an assessment of the effectiveness of *invalid action masking*.

6.1 Experimental Setup

All experiments were executed on an AMD EPYC 7F72 with 24 cores with Python 3.7 and PostgreSQL 12.5. Each experiment was repeated with multiple random seeds to ensure stable results. While neither the experiments nor the RL agent, in general, are limited to a particular index type, the following experiments use non-covering B-trees, the default index type of PostgreSQL.

Competitors. Based on the findings of a recent performance study by Kossmann et al. [32], we compared *SWIRL* with three state-of-the-art approaches: *DB2Advis* (fastest), *Extend* (best), and *AutoAdmin* (well-tried); cf. Section 3.1. The implementations[10] were obtained from the study. Note that "the re-implemented algorithms do not fully reflect the behavior and performance of the original tools, which may be continuously enhanced and optimized." [32, p. 2387]. We have chosen *DRLinda* [48] and Lan et al. [33] for RL comparisons. Lan et al.'s solutions were shown to be on par with state-of-the-art approaches [33] for known workloads. *DRLinda* is the only competitive algorithm that seeks to generalize to unseen workloads [48, 49]. Originally, *DRLinda* does not support storage budgets but a maximum number of indexes to create selections of different sizes. To evaluate selections for a given budget, we subsequently select indexes according to the order associated with *DRLinda*'s solutions for increasing numbers of indexes as long as permitted by the experiment's budget. To achieve better and more fine-grained solutions, we also check whether subsequent (potentially smaller) indexes can be added.

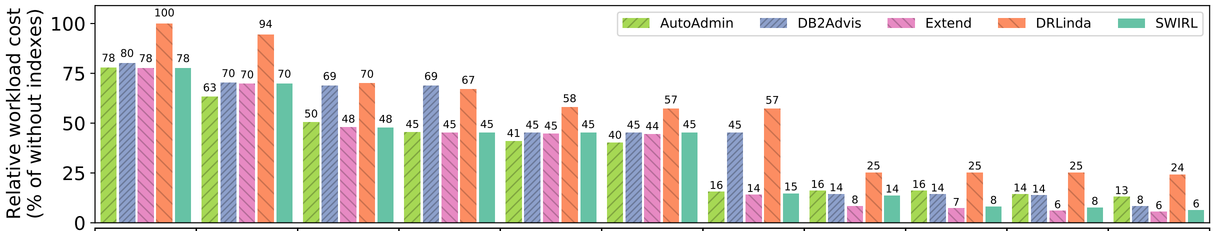
Attempts to combine *DRLinda* with Lan et al.'s [33] solution to support multi-attribute indexes were not successful.[11] Their index representation approaches – attribute-based for *DRLinda* vs. index candidate-based for Lan et al. – might not be compatible.

Benchmark workloads. We chose the three benchmarks: TPC-H [45], TPC-DS [41], and Join Order Benchmark (JOB) [34] for evaluation because they contain complex queries that challenge index selection approaches [32] and differ in dataset size and origin as well as workload complexity (number and intricacy of queries). The JOB is based on data from the Internet Movie Data Base (IMDB), while the TPC benchmarks utilize synthetically generated data. All benchmark-defined primary and secondary indexes are removed for the following experiments.

For a better assessment of index selection algorithms during the following measurements, and in line with a previous evaluation study [32, p. 2389 and Figure 4], we exclude the TPC-DS queries 4, 6, 9, 10, 11, 32, 35, 41, 95 and the TPC-H queries 2, 17, 20. The study states that "these queries dominate the costs of the

[10]Source code of competitors: https://git.io/index_selection_evaluation

[11]DRLinda multi-attribute attempt: https://github.com/hyrise/rl_index_selection/tree/main/experiments/drlinda_multi_attribute



	0.5 GB	1.0 GB	1.5 GB	2.0 GB	2.5 GB	3.0 GB	4.0 GB	5.0 GB	6.0 GB	7.5 GB	10.0 GB
AutoAdmin	167.72s	187.57s	270.06s	409.41s	167.88s	210.88s	532.60s	582.03s	682.55s	700.65s	765.96s
DB2Advis	9.57s	9.61s	8.19s	7.50s	6.91s	7.73s	7.50s	8.32s	8.29s	8.19s	7.84s
DRLinda	1.78s	3.42s	2.62s	3.73s	6.56s	5.23s	7.79s	6.95s	6.07s	9.75s	8.53s
Extend	63.27s	112.30s	98.98s	136.68s	155.97s	168.05s	157.39s	153.59s	216.54s	246.65s	324.16s
SWIRL	**0.71s**	**2.24s**	**0.37s**	**1.14s**	**2.57s**	**4.45s**	**3.01s**	**2.79s**	**2.08s**	**5.72s**	**7.13s**

Figure 6: Comparison of state-of-the-art approaches vs *SWIRL* for a Join Order Benchmark workload ($N = 50$; 20% of templates are unknown to *SWIRL*) on PostgreSQL. Chart: cost relative to processing w/o indexes; table: selection runtime.

entire workload, thereby rendering the index selection problem less complex because an index that decreases the cost of at least one of these queries would always outperform indexes for other queries by orders of magnitude" [32, p. 2389]. Further, we use an index width W_{max} of at most 3 as larger values are not beneficial for the considered workloads for PostgreSQL, cf. [32]. The representation width (Section 4.2.1) is set to $R = 50$ for all experiments.

6.2 Performance

For the following experiments, we create workloads of size N by randomly choosing query templates from all available templates of a selected benchmark and assign random frequencies according to a uniform distribution. In addition, we define a certain number of query templates that are withheld during training and the share that these withheld queries make up in the test workloads used for evaluation. Independent of the unknown templates, we always ensure that test workloads are not used for training. Thus, evaluated workloads differ from training workloads in three dimensions: (i) the exact combination of query templates and (ii) the exact frequency-template-combination have not been seen during training. (iii) Also, the evaluated workloads contain the unknown templates withheld during training.

First, we evaluate the performance in terms of achieved solution quality *(R-I)* (cf. Section 3.3) and observed runtime *(R-II)* for determining the evaluated solutions. Therefore, *SWIRL* is compared to the competitors in Figure 6 for a single JOB-based workload for budgets from 0.5 to 10GB. We used a workload size of N=50. Of the JOB's 113 query templates, 10 were withheld during training, all of these are included in the workload evaluated in Figure 6. Hence, 20% of the workload's query templates are unknown to the agent. The bar chart depicts the estimated[12] workload processing costs for different budgets while the table displays the runtime for these solutions. The figure demonstrates that a set of adequately chosen indexes decreases the costs significantly. Also, it becomes apparent that *SWIRL* is competitive with state-of-the-art approaches and outperforms *DRLinda*: in terms of cost, its performance is roughly on par with the best-performing algorithms (or even better for large budgets). Regarding selection runtime, it outperforms all competitors in all cases. *AutoAdmin* and *Extend*, which were among the best in previous performance studies [32], are often orders of magnitude slower.

After evaluating one workload comprehensively, we now evaluate *SWIRL*'s performance across many different cases. For each of the three benchmarks, we trained a model and generated 100 random evaluation workloads with different frequencies and 20% of query templates that were withheld during training. For each of the 100 evaluation workloads, we determined index configurations with all competitors for random budgets between 0.25GB and 12.5GB, calculated average performances as well as runtimes, and present the results in Figure 7. In contrast to our *train-once-apply-often* approach, Lan et al. has to determine a solution via their RL algorithm for every problem/workload instance, cf. Section 3.2. The non-parallel RL algorithm causes large training/solution times, > 12 hours for some TPC-DS workloads. Therefore, we could evaluate only TPC-H workloads with the provided implementation (Footnote 4 on Page 4) for Lan et al.

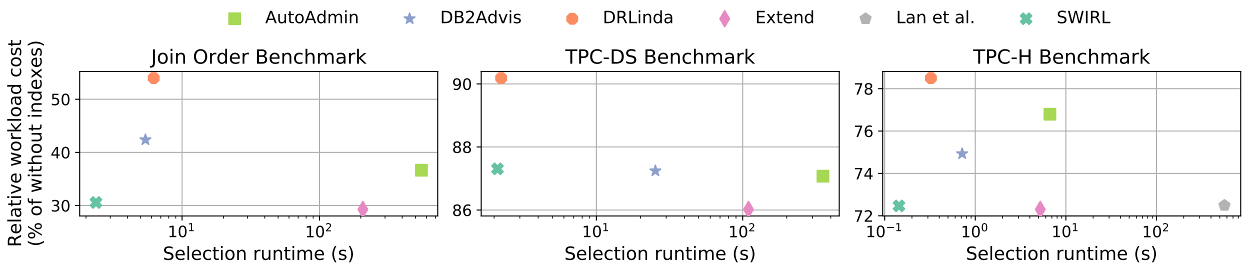
Figure 7 depicts the averages (means) of the relative workload cost and the index selection runtime for 100 evaluation workloads for each of the three benchmarks. While *Extend* determines the best solutions across all benchmarks (regarding the relative total workload costs compared to using no indexes: $RC := C(I^*)/C(\emptyset)$, Equation (1)), *SWIRL*'s solutions are, on average, close to the best solutions. For instance, for TPC-DS, *SWIRL*'s performance is on average only 1.3pp worse than *Extend*'s. In addition, the performance is superior over *DB2Advis*, *AutoAdmin*, *DRLinda*, and Lan et al.'s solution. The large overall differences to *DRLinda* can be explained by its approach to representing workloads and the lack of multi-attribute index support. Lan et al.'s performance is close to the best that we have observed, but its required solution time (see above) is also the highest.

Regarding the mean selection runtime (cf. $\varnothing t$), *SWIRL* outperforms all five competitors after training. It undercuts *AutoAdmin*, *Extend*, and Lan et al. by orders of magnitude. For example, for the TPC-DS, *SWIRL* determines index configurations in 2.1s on average, while *DB2Advis* takes 12×, *Extend* 52×, and *AutoAdmin* 168× as long. *DRLinda* is only marginally slower. The difference might be due to *SWIRL*'s efficient handling of index candidates via action masking and the native support for budget constraints, see Section 6.1. As a result, *SWIRL* dominates all competitors in terms of selection runtime and the fast competitors (*DRLinda* and *DB2Advis*) in both dimensions, performance and runtime.

6.3 Training Duration & Effort

RL index selection approaches pay the price for determining efficient configurations upfront: short runtimes at application time are exchanged for long training durations. Table 3 shows the training time *(R-III)* and the number of cost requests that occurred during training (with 16 parallel environments) along with the number of features and actions for different scenarios.

[12]The underlying index selection evaluation framework [32] also allows reporting actual workload execution times. This functionality can be used to, e.g., assess cost estimation accuracy. Note, such an assessment is not the focus of this paper.



	Ø RC	Ø t	Ø RC	Ø t	Ø RC	Ø t
AutoAdmin	36.64%	552.4s	87.07%	353.5s	76.79%	6.7s
DB2Advis	42.38%	5.4s	87.24%	25.3s	74.93%	0.7s
DRLinda	53.98%	6.2s	90.19%	2.2s	78.50%	0.3s
Extend	**29.27%**	207.0s	**86.02%**	109.5s	**72.30%**	5.2s
Lan et al.	-	-	-	-	72.49%	563.0s
SWIRL	30.57%	**2.4s**	87.31%	**2.1s**	72.47%	**0.1s**

Figure 7: Comparison across 100 random workloads of the TPC-H (SF10), TPC-DS (SF10), and Join Order Benchmark on PostgreSQL. $RC := C(I^*)/C(\emptyset)$ denotes the relative workload costs; Ø the arithmetic mean, t the selection time.

The training duration refers to the time needed for convergence until no further improvements are realized. Obtaining the cost for a query given a particular index configuration is denoted as a cost request. The number of requests is crucial for assessing index selection approaches because even though a single request takes only milli- or microseconds, it is not uncommon that millions of such requests are issued during selection processes [32]. In addition, the training duration also contains the time required for computing the state representation, applying action masking, creating hypothetical indexes, updating the weights of the neural network, and using it for predictions during training.

Several dimensions influence the problem complexity and, thereby, the training durations: (i) the workload size N influences the number of features and large values increase the time necessary for estimating the workload's execution costs as more queries have to be converted to query plans by the optimizer. (ii) The complexity of the queries: more complex queries cause longer optimization times. (iii) The number of index candidates: many candidates lead to large action spaces and the agent requires more time to determine efficient actions; also, the existence of more indexes can increase query optimization time [32, p. 2392]. For these reasons, the evaluated scenarios cover workloads of different sizes from all three supported benchmarks and different admissible index widths. Table 3 shows that training durations increase with the workload and index candidate complexity and range from multiple minutes to a couple of hours. Even for large workload sizes with 100 query templates and 3-attribute indexes, the training duration is reasonable. To set this into context, *Extend* needs ≈ 9 minutes to determine a solution for a scenario for which *SWIRL* trains 331 minutes, or 37x as long. As shown in Figure 6 and Figure 7, *SWIRL*'s runtimes only amount to a few seconds. Hence, if dozens or hundreds of systems must be tuned (repeatedly), short runtimes compensate for long training durations. The training time of our reimplementation of *DRLinda* is in the same range. While *DRLinda*'s model is simpler, the utilized DQN [39] is, in Stable Baselines, not as efficient as PPO.

Table 3 also demonstrates that a large fraction of the time required for training is caused by cost requests, even though most of these requests can be cached. This effect can be observed for most state-of-the-art index selection approaches, too [32, 43]. Interestingly, training durations can vary significantly for similar numbers of cost requests, which can be observed for the two JOB experiments: the first causes more cost requests but requires only 47% of the training duration. This effect is probably due to the much lower cost request cache rate, which, in turn, is caused by more index options and possible actions (due to a larger W_{max}).

Effectiveness of action masking. As explained in Section 4.2.3, we rely on *invalid action masking* to provide state-dependent action sets, thereby, assisting the agent during training by reducing the action space. In the following, we investigate the effectiveness of applying this technique to index selection. Figure 8 gives an intuition on the effectiveness by depicting the share of valid actions at any given point of a single training episode for a JOB scenario. The figure also shows how many valid actions refer to indexes of widths 1, 2, or 3 and the fraction of these actions that are invalidated because they do not fit the remaining budget.

Figure 8 demonstrates that in the beginning, only ≈ 8%, and at no point, more than 12% of all actions are valid; with a decreasing remaining budget, more indexes are invalidated because they are too large. In addition, the majority of valid actions refer to indexes of widths 1 and 2. For these reasons, invalid action masking appears as an effective technique to restrict action spaces. Consequently, this technique also decreases the required training duration for convergence. According to our experiments, the duration for a non-masking variant increases by 8× for a TPC-H scenario with a maximum index width, $W_{max} = 1$, to achieve comparable performance. This effect is even more pronounced for more realistic problems with larger action spaces: for TPC-H with W_{max}=3, which comes with significantly more index candidates ($|I|$=3 532 vs $|I|$=46), the solution quality of the non-masking version was not close to *SWIRL*'s, even after training more than 10× as long. We observe that without action masking, the overall performance might be worse even with extended training durations, which is in line with [28].

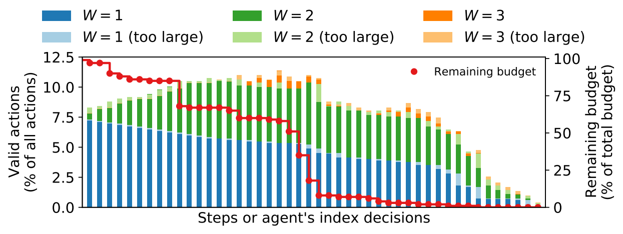


Figure 8: Impact of invalid action masking on the number of valid actions for a JOB scenario (storage budget B=10GB, maximum index width W_{max}=3, $|A|$ = 819 candidates).

Table 3: Training duration and problem complexity metrics for different scenarios. W_{max}: admissible index width.

Benchmark	N	#Features	W_{max}	#Actions ($A = I$)	#Episodes	Training duration breakdown			
						Total	Costing (% of Total)	#Cost requests (%cached)	∅ Episode time
TPC-H	19	468	1	46	2 272	0.07h	20.2%	1 829 088 (95.9%)	0.1s
TPC-H	19	468	3	3 532	768	0.19h	32.6%	1 802 016 (71.2%)	0.9s
TPC-DS	30	1 750	1	186	751	0.42h	22.0%	3 002 850 (92.5%)	2.0s
TPC-DS	30	1 750	2	3 174	512	0.77h	22.4%	2 995 680 (84.6%)	5.4s
TPC-DS	60	3 310	2	3 174	512	1.31h	23.4%	5 991 360 (86.9%)	9.2s
JOB	100	5 265	1	61	1 616	2.58h	37.4%	10 097 600 (83.3%)	5.7s
JOB	100	5 265	3	819	560	5.52h	47.5%	9 990 400 (63.4%)	35.5s

7 DISCUSSION & INTERPRETATION

The goal of our RL-based index selection approach was to fill the gap between algorithms that identify close-to-optimal configurations and algorithms that determine solutions quickly.

By comparing *SWIRL* to four state-of-the-art competitors with complex analytical benchmarks, we have shown that it fulfills the requirements stated in Section 3.3: it determines comparably good *(R-I)* (and sometimes better) multi-attribute *(R-IV)* index configurations also for partly unknown workloads *(R-VI)* faster than others *(R-II)* for different storage budgets *(R-V)* as shown in Section 6.2. Due to its stochastic nature, there is no guarantee for close-to-optimal solutions, and rarely, *SWIRL* performs worse than its competitors. We argue that this drawback is acceptable for the advantage gain in solution times, which is due to the different modus operandi compared to state-of-the-art approaches. Most of these approaches reevaluate the benefit of index candidates every time a candidate was chosen to incorporate index interaction (see Section 2.1) effects. In contrast, *SWIRL*, *internalizes* the knowledge about dependencies of index candidates (index interaction) during training and does not require costly reevaluations during application.

The handling of unknown query templates would not be possible if the agent only learned which indexes are suitable for which exact query templates. Instead, the agent must know about the queries' contents, i.e., its operators and about their influence on performance and indexes. Therefore, we featurize the workload's query plans (Section 4.2.2) to enable the agent to understand which operations of a query benefit from which index. Naturally, this approach is influenced by the data, i.e., query classes, seen during training: if unknown queries contain too many previously unseen operators, it is harder for the agent to associate them with index candidates and optimize decisions. In such cases, the training data should be improved. Furthermore, *invalid action masking* has been proven to be effective (Section 6.3) in reducing the number of applicable actions. Thereby, it enables acceptable training times *(R-III)*. Compared to the runtimes of state-of-the-art algorithms, the observed training durations are still significant. Consequently, there is a clear tradeoff between long a priori training durations and low runtimes during application. For this reason, the use of RL-based index selection approaches is not reasonable in all scenarios but only where the repeated determination of index configurations in similar scenarios is necessary, e.g., in cloud environments (cf. Section 1).

Training data influence. The results presented in Section 6 raise the question of how *SWIRL*'s generalization capabilities depend on the training data. We conducted two experiments[13]: (i) We investigated how the number of unknown queries during training impacts the agent's performance. As expected, the performance decreases if a higher number of query templates is unknown during training. (ii) We examined whether the performance depends on the particular set of query templates that are unknown during training. The specific selection of query templates appears to be of minor importance if the workload size N is sufficiently large.

Limitations. Our model is trained for a particular schema. Workloads that are totally different from the training workloads might cause suboptimal results. However, techniques as transfer learning [42] to speed up the retraining of existing models and more sophisticated workload representations, e.g., with word2vec [31, 38] approaches, could overcome these challenges. Also, even though there are no conceptual limitations, we have not evaluated *SWIRL* for transactional workloads yet. While their queries are usually less complex [1], such workloads pose different challenges, e.g., lock contention and index maintenance costs [24, 32].

8 CONCLUSIONS & FUTURE WORK

We presented a novel index selection approach based on reinforcement learning that identifies competitive solutions orders of magnitude faster than state-of-the-art approaches. In contrast to other RL solutions, *SWIRL* supports multi-attribute indexes and generalizes to (partly) unknown workloads. While *invalid action masking* appears to be an effective technique for efficient training, training can take up to a few hours and be up to 50× higher than solution times of state-of-the-art algorithms. Hence, it is most reasonable to apply *SWIRL* in scenarios where many index selection problems must be solved, e.g., managed cloud scenarios, because *SWIRL* trades off fast solution runtimes against elevated training durations.

For future work, our model could be extended to integrate further aspects of physical database design, e.g., automatic compression [4] or partitioning [27] selection. Also, the impact of alternative workload representation techniques, e.g., as presented by Sun et al. [54], on the model's performance could be investigated. Lastly, there are opportunities to reduce training times: *SWIRL* could be provided with expert-based [20] index configurations as a starting point. Such decisions could be derived from state-of-the-art algorithms, e.g., *Extend.* Additionally, *transfer learning* [42] could be applied. First, *SWIRL* would be trained on a wide variety of workloads (Phase 1). Later (Phase 2), when the particular application scenario is known, *SWIRL*'s training from Phase 1 is continued with more specific workloads. Each Phase 2 (for different scenarios) benefits from the Phase 1 training.

ACKNOWLEDGMENTS

We thank M. Boissier, F. Borges da Silva, S. Halfpap, D. Ho, F. Naumann, T. Rabl, and Z. Sadri for their feedback and support.

[13]Experiments investigating the impact of the training data: https://github.com/hyrise/rl_index_selection/tree/main/experiments/training_data_influence

REFERENCES

[1] Alberto Abelló and Oscar Romero. 2018. Online Analytical Processing. In *Encyclopedia of Database Systems, Second Edition.*

[2] Stefan Aulbach, Dean Jacobs, Alfons Kemper, and Michael Seibold. 2009. A comparison of flexible schemas for software as a service. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 881–888.

[3] Ella Bingham and Heikki Mannila. 2001. Random projection in dimensionality reduction: applications to image and text data. In *Proceedings of the International Conference on Knowledge discovery and data mining (SIGKDD).* 245–250.

[4] Martin Boissier. 2022. Robust and Budget-Constrained Encoding Configurations for In-Memory Database Systems. *PVLDB* 15, 4 (2022), 780–793.

[5] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. 2018. Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements. In *Proceedings of the International Conference on Data Engineering (ICDE).* 209–220.

[6] Philippe Bonnet and Dennis E. Shasha. 2018. Index Tuning. In *Encyclopedia of Database Systems, Second Edition.*

[7] Renata Borovica, Ioannis Alagiannis, and Anastasia Ailamaki. 2012. Automated physical designers: what you see is (not) what you get. In *Proceedings of the International Workshop on Testing Database Systems (DBTEST).*

[8] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *CoRR* abs/1606.01540 (2016).

[9] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 227–238.

[10] José Camacho-Collados and Mohammad Taher Pilehvar. 2018. From Word To Sense Embeddings: A Survey on Vector Representations of Meaning. *Journal of Artificial Intelligence Research* 63 (2018), 743–788.

[11] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek R. Narasayya. 2002. Compressing SQL workloads. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 488–499.

[12] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Databases (VLDB).* 146–155.

[13] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 367–378.

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC).* 143–154.

[15] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 666–679.

[16] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey F. Naughton, and Stratis Viglas. 2020. Comprehensive and Efficient Workload Compression. *PVLDB* 14, 3 (2020), 418–430.

[17] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. 1990. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science* 41, 6 (1990), 391–407.

[18] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 1241–1258.

[19] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *PVLDB* 13, 8 (2020), 1206–1220.

[20] Eyal Even-Dar, Sham M. Kakade, and Yishay Mansour. 2004. Experts in a Markov Decision Process. In *Advances in Neural Information Processing Systems (NIPS).* 401–408.

[21] Martin Faust, Martin Boissier, Marvin Keller, David Schwalb, Holger Bischoff, Katrin Eisenreich, Franz Färber, and Hasso Plattner. 2016. Footprint Reduction and Uniqueness Enforcement with Hash Indices in SAP HANA. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA).* 137–151.

[22] Gartner: Donald Feinberg, Merv Adrian and Adam Ronthal. 2019. *Gartner Says the Future of the Database Market Is the Cloud.* https://www.gartner.com/en/newsroom/press-releases/2019-07-01-gartner-says-the-future-of-the-database-market-is-the

[23] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning.* MIT Press.

[24] Goetz Graefe. 2006. B-tree indexes for high update rates. *SIGMOD Record* 35, 1 (2006), 39–44.

[25] Zellig S. Harris. 1954. Distributional Structure. *WORD* 10, 2-3 (1954), 146–162.

[26] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. 2018. Stable Baselines. https://github.com/hill-a/stable-baselines, visited 2022-02-22.

[27] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 143–157.

[28] Shengyi Huang and Santiago Ontañón. 2020. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. *CoRR* abs/2006.14171 (2020).

[29] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR).* 68–78.

[30] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB* 4, 9 (2011), 585–597.

[31] Shrainik Jain and Bill Howe. 2018. Query2Vec: NLP Meets Databases for Generalized Workload Analytics. *CoRR* abs/1801.05613 (2018).

[32] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *PVLDB* 13, 11 (2020), 2382–2395.

[33] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An Index Advisor Using Deep Reinforcement Learning. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM).* 2105–2108.

[34] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[35] Gabriel Paludo Licks, Júlia Mara Colleoni Couto, Priscilla de Fátima Miehe, Renata De Paris, Duncan Dubugras A. Ruiz, and Felipe Meneguzzi. 2020. SmartIX: A database indexing agent based on reinforcement learning. *Applied Intelligence* 50, 8 (2020), 2575–2588.

[36] Vincent Y. Lum and Huei Ling. 1971. An Optimization Problem on the Selection of Secondary Keys. In *Proceedings of the 26th Annual Conference (ACM '71).* 349–356.

[37] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 631–645.

[38] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. arXiv:cs.CL/1301.3781

[39] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013).

[40] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-Efficient Aggregation: Hashing Is Sorting. In *Proceedings of the International Conference on Management of Data (SIGMOD).* 1123–1136.

[41] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the International Conference on Very Large Databases (VLDB).* 1049–1058.

[42] Sinno Jialin Pan and Qiang Yang. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 22, 10 (2010), 1345–1359.

[43] Stratos Papadomanolakis, Debabrata Dash, and Anastassia Ailamaki. 2007. Efficient Use of the Query Optimizer for Automated Database Design. In *Proceedings of the International Conference on Very Large Databases (VLDB).* 1093–1104.

[44] Gregory Piatetsky-Shapiro. 1983. The Optimal Selection of Secondary Indices is NP-Complete. *SIGMOD Record* 13, 2 (1983), 72–75.

[45] Meikel Pöss and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Record* 29, 4 (2000), 64–71.

[46] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks.* 45–50.

[47] Julien Rouhaud. 2015. HypoPG - Hypothetical Indexes for PostgreSQL. https://github.com/HypoPG/hypopg, visited 2022-02-22.

[48] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. DRLindex: deep reinforcement learning index advisor for a cluster database. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS).* 11:1–11:8.

[49] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. Online Index Selection Using Deep Reinforcement Learning for a Cluster Database. In *Proceedings of the International Conference on Data Engineering (ICDE) Workshops.* 158–161.

[50] Rainer Schlosser, Jan Kossmann, and Martin Boissier. 2019. Efficient Scalable Multi-Attribute Index Selection Using Recursive Strategies. In *Proceedings of the International Conference on Data Engineering (ICDE).* 1238–1249.

[51] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. 2009. Index Interactions in Physical Design Tuning: Modeling, Analysis, and Applications. *PVLDB* 2, 1 (2009), 1234–1245.

[52] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017).

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

[54] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *PVLDB* 13, 3 (2019), 307–319.

[55] Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement learning - an introduction.* MIT Press.

[56] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *Proceedings of the International Conference on Data Engineering (ICDE).* 101–110.

[57] Kyu-Young Whang. 1985. Index Selection in Relational Databases. In *Proceedings of the International Conference on Foundations of Data Organization (FoDO)*. 487–500.

[58] Yu Yan, Shun Yao, Hongzhi Wang, and Meng Gao. 2021. Index selection for NoSQL database with deep reinforcement learning. *Information Sciences* 561 (2021), 20–30.

[59] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *Proceedings of the International Conference on Very Large Databases (VLDB)*. 1087–1097.