Automatic Index Creation using Deep Reinforcement Learning

Ivan Prymak, Milena Malysheva, Guzel Mussilova, Mahmoud Mohsen, Ali Hashaam, Celine Petrosian
University of Magdeburg
firstname.lastname@st.ovgu.de

***Abstract*—Data management is always challenging. With continuous changes in the scale of data, different formats supported and workloads, it becomes challenging to reach the appropriate system configurations that lead to good runtime. Database administrators are commonly expected to deal with these challenges. Given the complexity of the tasks, and their high error-proneness, the database community has worked for years developing tools to automate and help such tasks. In this project we evaluate the gains that new techniques from machine learning can bring to automating the index selection process. We propose and evaluate different models, some are based on the literature and some are novel propositions by us; testing them over a commercial database and with a publicly available benchmark. We find that learned models work well when heuristics fail (leading to 40% lower costs), but heuristics can still outperform learned models in several cases. We also find that supervised learning can feasibly become a novel way for supporting reinforcement learning.**

***Index Terms*—Automatic index selection, Deep reinforcement learning, Physical design, Q-Learning, DQN, Supervised learning**

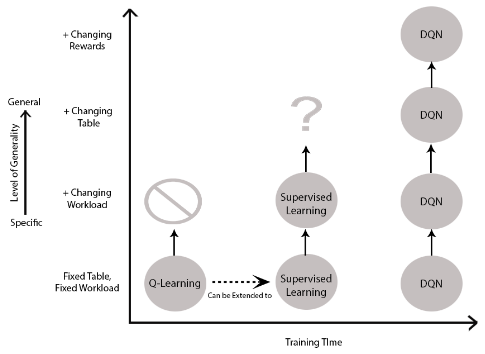

Fig. 1. Generalizability vs. Training time of reinforcement learning, deep reinforcement learning and supervised learning methods

I. Introduction

Physical database design is always among the most important data management concerns, since the right set of physical structures or configurations has a strong impact on query performance. This has influenced numerous DBMS vendors to ship their systems together with so-called *design advisory tools*, which enable a partly automated physical design process. These tools assist database administrators (DBAs) in finding an optimal or near-optimal configuration to improve performance; for example by recommending auxiliary structures (e.g., indices) that for a given workload help to minimize the query execution costs. From a more general perspective, automated physical designers aim to reduce the database administration efforts and help non-experts to exploit database systems.

Even though, these tools can still provide the anticipated performance improvements for some evaluations, the accuracy in their estimations for more general scenarios is often disregarded [4]. Studies [4] have experimentally proven that, in reality, advisory tools are often affected by errors that are propagated from the query optimizer (e.g. optimizers exploit principles like attribute value independence in their estimations, which can result in producing routinely large mistakes [1], [10]). Accurate estimations imply that DBAs can accept the recommendations with a higher certainty, whereas inaccuracies may lead to loss of trust in the advisory tools. Therefore, in order to reach the right decision, DBAs sometimes also have to rely on their experience and intuition.

Another core problem is that these tools work mostly offline, since their recommendations require the execution of an optimization algorithm, that evaluates the predicted gains from alternative indexes, by using a cost model [19]. Since the search space of possible combinations is large, even after pruning, this process can be time consuming. Therefore, for online usage (i.e., providing real-time recommendations) there is a need for a tool that can give responses in a short time.

Furthermore, the physical designers do not seem to learn from their previous experience. Such sense of intuition and experience are embedded in the area of machine learning (ML). A conceptual framework for experience-driven learning is *reinforcement learning* (RL) [7].

Despite of the all advantages that RL provides, its past approaches lack scalability and were limited to low-dimensional problems. By using neural networks within RL, deep learning advances RL and defines the field of “deep reinforcement learning” (DRL). Relying on the properties of neural networks (e.g., function approximation, representation learning, etc), DRL techniques seem to efficiently tackle the curse of dimensionality. Namely, they can automatically derive low-dimensional representations of high-dimensional data. They have shown good performance in practice, outperforming conventional ML approaches in learning complex tasks like playing games [6], [8]. Recent works on applying deep learning techniques in the database physical design have also demonstrated some promising early results [2], [3], [5], [9].

Capitalizing on the principles of DRL, in this paper we present a model, which aims to be accurate, trustworthy, generalizable and efficient enough to run on-line; and by learning from its experience enables automated adaptive indexing. We specifically consider a design space for agents that we sketch in Fig. 1. Here we illustrate our observation that Q-Learning, Supervised Learning and Reinforcement Learning Agents can be trained to deal with problems at different levels of generality. However simple Q-learning cannot scale to changes in the workload (which would require keeping several Q-tables), nor is it clear that supervised learning can scale beyond a single change. Per design deep reinforcement learning approaches should be able to scale towards more complex problems, but this comes at the cost of more training time than other approaches. In consideration of this design space we evaluate in this paper the use of different agents. Our work aims to be an addition to previous research in this topic (Sharma et al. [2]), since we consider a different evaluation and we compare with heuristics (rather than index all or no index), and agents other than DQN ones.

Our core contributions are:

- We introduce a new model that extends the work of Sharma et al. [2] by implementing different agents (namely, Q-Learning, Deep Q-Network and Supervised Learning) to address the problem of automatic index creation for a maximum of 3 indexes.
- We propose a more comprehensive evaluation of these agents by having different baselines. Among our findings we can confirm that DQN leads to cost improvements on the final step of 1.3x over the heuristic of selecting to index the 3 columns with the overall less selectivity for the workload.
- We propose a novel idea of using supervised learning for the function approximation component of RL.

The rest of the paper is structured as follows. In Section 2 we formulate a problem of automatic index creation, then provide brief background on RL and define the main components in form of an RL instance. In Section 3 we elaborate our design decisions and the preliminary implementation of our model. Further, in Section 4 we present an experimental setup along with the first results. Related work is discussed in Section 5. In Section 6 we conclude our work and discuss the open challenges.

II. Problem Formulation: Automatic Index creation using Reinforcement Learning

In this section we establish the core problem that our work tackles: automatic index creation.

A. Automatic index creation

Index creation aims to select attributes to create secondary indexes on, after which the processing of the workload will benefit the most. Simply creating indexes on all given attributes is not reasonable due to space constraints and index maintenance costs. Defining the problem more precisely: let us consider that our database schema S consists of n columns and k is the maximum amount of indexes which we can create where k $\leq$ n. Hence, our goal is to find a subset of S of maximum size k to decrease the runtime on a workload W.

In the past this problem was already addressed in numerous works. E.g., in one of the earliest works by researchers from AutoAdmin Project at Microsoft Research [15], where in their work on "What-if" indexes they proposed an index selection tool to enumerate and pick a recommended set of indexes by exploiting an index analysis tool which determines the "goodness" of the index sets based on a cost model.

B. Background

1) Reinforcement Learning: RL is a field of ML which stems from behaviorist psychology. The core idea revolves around learning through *interaction*, where an RL agent's behavior corresponds to a *reward-driven behavior* [7]. I.e., while interacting with its environment and by examining the outcome of its actions, the RL agent learns to adjust its behavior towards maximizing a cumulative *reward*.

The typical RL set-up is presented by Arulkumaran et al. [7], in it an *agent* is placed in an *environment* at time *t*, with the task of observing a *state* s_t, and then deciding on which action to perform. After performing the action a_t, the agent transitions to a new state s_{t+1}. If there is no state transition, the problem is called a multi-arm bandit problem, requiring techniques simpler than RL. When there is a state transition, these can be either deterministic or non-deterministic.

Each state ideally contains all information from the environment that is needed for the agent in order to take the best action. These are called observable environments. When the environment does not provide such characteristics they can be partially observable environments.

Maximal rewards derived from the environment define the best sequence of actions. After each transition of the environment to a new state, the agent receives a scalar reward r_{t+1} as feedback. Thus, the agent aims to learn *policy* π that maximizes future *return* (cumulative discounted reward), where an *optimal policy* is the one that maximizes this expected return. Additionally, the RL agent is always exposed to the "exploration-exploitation" dilemma, i.e., to obtain already gained knowledge or to aim for increasing the long-term rewards; and the learning formulation is also exposed to the credit assignment problem, where the value of rewards needs to be properly assigned to actions.

The set-up for RL can be formally described as Markov Decision Process (MDP), with:

- S : Set of states s.
- A : Set of actions a.
- $p(s_{t+1}|s_t, a_t)$: Set of transition dynamics that maps a state-action pair at a time t into a distribution of states at time t+1.
- $p(r_{t+1}|s_t, a_t)$: Reward function.

- γ : Discount factor which is set between 0 and 1, where the lower values put higher priority to immediate rewards.

Here, the agent-system interaction is represented by the policy π that maps states s to a probability distribution over actions a. This policy is evaluated by a value function that correlates certain cumulative discounted reward to each state. These value functions are state-action pair functions, they evaluate the goodness of a particular action in the given state by deriving an estimated long term return for that action. The value function of a given policy obeys an identity known as the Bellman equation, where the value function satisfies the linear evaluation equation and the optimal value function satisfies the non-linear optimality equation of Bellman. For more details we refer the reader to the work of Arulkumaran et al. [7].

C. The problem formulation

Given the provided background, the problem of an automatic index creation as an RL instance in the form of MDP can be formulated as follows:

- *Environment:* The environment is represented as the set of possible states s, actions a and the rewards p.
  - *States:* States are formulated the same for all agents, but agents have different start states. The state for all agents is expressed as the boolean array of columns with indexes in them, in addition to the workload given (which is represented as a matrix, with each row being a query, and each entry in the row being the selectivity factor (SF) of the query on each column C_j. The starting state for our agents is a configuration without indices.
  - *Actions:* The set of actions a in our case is narrowed down to a single action A of creating an index on a specific column C_j, if that index does not exist:

    $$A = \{create\_index\_on(C_j)\}$$

    We do not evaluate the action of dropping indices.
  - *Rewards:* In order to obtain the rewards for the agents we take the reciprocal of the estimated cost provided by our tested database (PostgresSQL) query optimizer. Thus, the higher cost the lower reward is:

    $$p(r_{t+1}|s_t, a_t) = \frac{1}{estimated\ query\ cost}$$

  Agents: In our problem formulation the agent is used for learning the long term value of a sequence of actions (adding indexes) during a number of training iterations, given an initial state and a workload passed as the parameters, with the workload being fixed per episode. For our current research and implementation we selected Q-Learning, Deep Q-Learning and Supervised Learning (XGBoost) agents.

III. Implementation

Before delving into the implementation details we explain the design decisions of the selected agents.

A. Design Decisions

1) Q-Learning: Q-Learning is an RL method which is used when the agent initially is only aware of a set of possible states and actions. It is able to improve its behavior through learning from the history of its interactions with the environment [9]. Since policy-based methods can require more training for large action spaces, we have chosen for our problem Q-Learning as a value-based method which in order to converge to a reasonably optimal solution does not require a complete exploration of all possible policies. Although, it still requires a sufficient training in order to converge, storing Q-values results in a memory consumption overhead, it does not generalize to unvisited states nor does it manage changing workloads, it can work well as a baseline approach for a reinforcement learning method.

2) Deep Q-Learning: Deep Q-Learning generally exploits the same premise as the regular Q-Learning but extends it for larger domains by approximating the Q-function instead of finding an optimal one. It operates under the assumption that it is highly inefficient to store a direct mapping between the states, actions and reward (what Q-Learning does in a table form) and exploits a neural network (NN) (i.e., Deep Q-Network (DQN)) to estimate these Q-Functions instead. Moreover, DQN can approximate unvisited states. Particularly given the state as an input and by storing the information about the reward for visited states, it predicts the outcomes for each action taken from the given state. During the stage of exploration the DQN is trained to produce better estimates, whereas during the exploitation stage it chooses the optimal actions. Among the drawbacks of this approach, Deep Q-Learning requires more training time than Q-Learning; first because the neural network requires training, and second because in practice more instances of successful episodes need to be visited such that the agent learns.

3) Supervised Learning (XGBoost): While Deep Q-Learning, when trained for a given workload can produce satisfactory results, without prior knowledge of the state transition and reward models, for our problem it still requires some amount of training time. Having the results from several different Q-Learning executions (each finding the optima given a limited number of iterations, over different workloads), with the selectivities as features and the selected indexes as labels, we can consider using supervised learning algorithms to learn the task of predicting indexes, framing it as a multi-label classification task. Here for each input of selectivities, the supervised learning classifier should be able to predict the best k indexes. This is learned by using the output of several Q-Learning executions. Considering that it is a typical classification algorithm task, algorithms such as XGBoost, kNN or SVM were evaluated in terms of F1 score, where XGBoost shows the best score. Moreover, this score can be used later to measure and compare the "goodness" of our agents.

B. Implementation

We present our implementation of automatic index creation using the reinforcement learning agent, DQN-agent and XG-

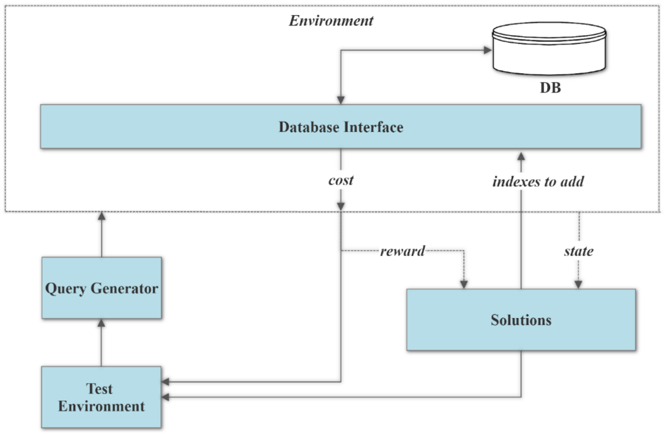

Fig. 2. Core Components

Boost agent. The core components of our proposal along with their interactions are illustrated in the Figure 2.

1) Q-Learning: In our implementation of the Q-Learning agent we used an open source toolkit OpenAI Gym. In OpenAI Gym, we implement an episodic reinforcement learning, where each episode stands for a transition from an initial configuration to a final configuration with no indexes at all in the beginning and the maximum number of indexes k at the end. In general, each new episode is formed by starting with the initial configuration, incrementing by one the number of indexes until k is reached, then the episode is terminated.

In our implementation Q-Learning is provided with the results from the heuristic as a starting point for its evaluation (i.e., the indexes selected by the heuristic correspond to the steps chosen in the first episode).

Considering an increasing memory consumption with an increasing number of indexes, in our implementation we set k to 3, this means that there are 3 indexes to be selected by the end.

As was mentioned above the input for the Q-Learning is a state which is basically an index configuration. Each Q-value of state s and action a can be expressed using the Bellman equation in terms of the Q-value of the next state as follows:

$$Q(s,a) = r + \gamma max_a Q(s', a')$$

The maximum future reward for this state and action is computed here as the immediate reward plus the maximum future reward of the next state. In Q-learning the Q-function can be iteratively approximated using the Bellman equation, where as its simplest case the Q-function can be implemented as a table, with states and actions as rows and columns, respectively, and the values stored in the middle. They can also be stored in a hash table where the state-action pair is used as a hashing key. It is also worth mentioning that with each new workload the Q-Table is to be reseted as it is being used in a workload-specific manner. To generalize to different workloads means that the table will have to learn different values for the actions, creating more entries in the table, since each combination of index configuration (a boolean array indicating what indexes are being used) plus a workload represents different states.

$$Q(s,a) = Q(s,a) + \alpha\{r + \gamma max_{a'} Q(s', a') - Q(s,a)\}$$

Here, γ is a learning rate that is used to control the difference between the previous Q-value and the newly derived Q-value. Considering α = 1 we obtain two $Q(s,a)$ that cancel each other and get the same Bellman equation as described above. We use $max_{a'} Q(s', a')$ as an approximation in order to update $Q(s,a)$. Even though this approximation may be erroneous at the early stages of the learning, it gets more accurate iteratively. After some time when this update is performed enough times, the Q-function eventually converges and provides the true Q-value.

As for the policy for choosing the next action we exploit the static ϵ-greedy along with the Boltzmann. Using ϵ-greedy an action will be obtained with a probability ϵ to be a random action (exploration) and a probability $1 - \epsilon$ to be an action that maximizes the expected Q-value for the learning process (exploitation). However, since ϵ is a static value and we want to achieve a certain adaptiveness, so that the probability will not to depend on the constant ϵ and will change with a growing number of episodes, we use Boltzmann that provides us these dynamics. With Boltzmann actions are selected through a stochastic process that assigns a probability that is proportional to the learned Q-value for the action, hence a balance between exploration and exploitation can also be achieved.

2) Supervised learning: In addition to the traditional approaches, in this project we propose and evaluate a novel supervised learning model as a new perspective to predict the indexes for unseen workloads, based on the experience generated by Q-Learning agents. The idea is similar than approaches that use deep neural networks, but we propose to reframe the problem as a classification instance. Firstly, given a workload we derive the selectivity for each of the queries and then we aggregate those selectivities in order to obtain the cumulative selectivity. The aggregation of the selectivities is performed using pair-wise multiplication. We propose this in consideration that the signal of low selectivities should be inflated, in contrast to the signal of higher selectivities. In this aggregation method the selectivity values range from 0 to 1, therefore no further normalization is needed.
In our case we perform a multi-label classification (using the cumulative selectivities of the workload as features, and the k indexes selected as output), though XGBoost predicts only for a single output, as a workaround we have used scikit-learn OneVsRest classifier. Figure 3 illustrates the tree structure of XGBoost. The leaves values are used to calculate the probability of predicting an index on the first column of the LineItem table. F0,f11 etc, represent the features or the commutative selectivities for column0 and column11, respectively. We have used the default parameters of XGBoost. Additionally, as this model overfits the received workload, we assume that tuning the hyper parameters may provide better results and can be as a future work.

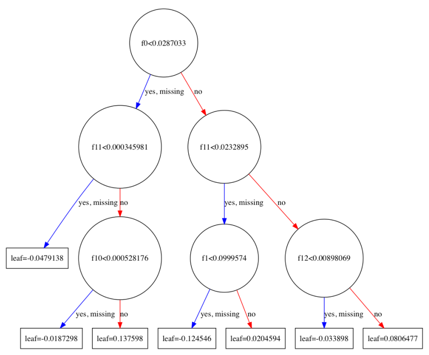

Fig. 3. Tree structure of XGBoost

In terms of training, the supervised learning proceeded as follows:

1) First, we ran 500 instances of the Q-learning process, each with a random workload. We stored the selected indexes for each process, and the cumulative rewards, forming a table of features and labels. At the end of each process we discarded the Q-tables.
2) Second, we ran a classifier on this table, such that the based on workload features we could predict the best indexes.

We should note that the supervised learning differs from Q-learning and from deep Q-learning in the fact that it does not consider the rewards of a sequence of actions, but instead it only focuses on the complete set of actions. Furthermore, the focus is not on the long-term reward of an action, but on mimicking the decisions of an informed process, learning from an amount of such processes.

3) DQN: DQN enhances basic Q-Learning for larger domains by approximating the Q-Function instead of finding an optimal, its main goal is to minimize as good as possible the distance between the Q-Function and approximated function.

$$\hat{Q}_\theta(s,a|\theta) \sim Q(s,a)$$

Since for our implementation we learn the Q-value given a state and an action. Thus, the Bellman's equation is to be updated in the following way:

$$Q(s,a) = r_t + \gamma max_{a'} \hat{Q}(s',a'|\theta)$$

In the Design Decisions subsection we already mentioned that for the DQN method the state as an input is fed into the NN. This input consists of two parts, namely the workload I_w and the current index configuration I_i [2]. I_w is basically a $n \times m$ matrix with the values for selectivities of a query batch in the interval of [0;1], with n as the number of queries in the given workload and m as a number of columns. Thus, a matrix entry i,j represents a selectivity $Sel(Q_i, C_j)$ of a column C_j given a query Q_i as follows:

$$I_w = \begin{bmatrix} Sel(Q_0, C_0) & \cdots & Sel(Q_0, C_{m-1}) \\ \vdots & \ddots & \vdots \\ Sel(Q_{n-1}, C_0) & \cdots & Sel(Q_{n-1}, C_{m-1}) \end{bmatrix}$$

where SF for each query batch is derived in the following way:

$$Sel(Q_i, C_j) = \begin{cases} \frac{\#\ selected\ tuples\ of\ Q_i\ on\ C_j}{\#\ total\ tuples\ of\ C_j} & if\ C_j\ is\ in\ Q_i \\ 1 & if\ C_j\ is\ not\ in\ Q_i \end{cases}$$

If C_j is presented in a query Q_i, then $Sel(Q_i, C_j)$ is returned as the fraction of selected tuples to the total number of tuples. I.e., the higher the result, the smaller the selectivity and the other way around. Whereas if C_j is not presented in a query Q_i at all, then $Sel(Q_i, C_j)$ = 1. As a result, the worst selectivity will encourage the system not to build an index on this column.

The second part of the input is the current index configuration I_i which is represented as an array of size m for the index existence per column (i.e., 0 for non-indexed column, otherwise 1). Hence, I_i can be formulated as follows:

$$I_w = [isIndexed(C_0) \cdots isIndexed(C_{m-1})]$$

The output is an array of size m with the returned Q-values that will be used then to choose an action.

As for the next parameters of NN, our neural network consists of 4 layers: 1 input layer with 6×16 nodes (where this corresponds to 16 columns of the table, times 1 for the index configuration array plus and 5 for a set of 5 queries that we trained for), 2 hidden layers with 2×16 nodes each, and 16 output nodes, corresponding to the value of each action (adding an index on a column). In order to store previous experiences of the learning we use an $experience\ replay$ for the batches of 32 previously experienced iterations at each training step. For network implementation we use Keras, ReLu activation functions and select Adam as an optimizer. The agent itself uses Boltzmann's Q-Policy to select the actions from Q-value predictions.

The environment on which the DQN agent learns assigns a reward for every action taken, the action being the selected column to index. The reward function is obtained as follows:

$$p(r_{t+1}|s_t, a_t) = \begin{cases} \frac{1}{query\ batch\ execution\ time} & if\ C_j\ was\ not\ indexed \\ -1 & if\ C_j\ was\ indexed \end{cases}$$

If the action was taken for the column C_j that had not been indexed, then the reward is returned as reciprocal to the query batch execution time (cost given by the optimizer), otherwise -1. Thus, the agent will maximize the total reward by minimizing the query batch execution time and avoid indexing the same columns. Furthermore, the environment itself changes the query batches to make the agent able to generalize over any query batch. For our training, we have used 50000 training steps (actions taken) with query batch changing every 1000 steps.

4) DB Integration: For our implementation we performed an integration with DBMS PostgreSQL, where from the DB interface we exploit the functions for adding real indexes and getting query execution time. Since PostgreSQL devises a query plan for each query received, therefore in order to get a query execution time the EXPLAIN command was applied. A typical output of this command contains a basic query information with respect to the query issued. For our evaluation we use the values of the Total Cost output as execution time estimation.

IV. Evaluation and Results

A. Experiment Setup

We conducted our experiments on a PC, equipped with a CPU Octa core Intel Xeon E5-2630 v3s- 2014 with 8 cores 2.4 GHz and a NVIDIA GK110BGL [Tesla K40c]. For learning, we use keras-rl [12], which uses python on top of keras for implementing a general reinforcement learner, that is itself a NN API on top of a deep learning backend like Tensorflow, Theano, or Microsoft CNTK. We chose Tensorflow with GPU support as a backend to keras in this work. To describe the problem environment, we use the library from the OpenAI project called Gym [13] and pass then this environment to keras-rl to abstract away the database interactions. In order to avoid the network to get stuck in local minimum by getting trained on the current experience, we use the optimization of experience replay. For the database part, we use the schema and data of the TPC-H benchmark [11] and PostgreSQL to run the queries. We use scale factor 1. In order to connect the network to the DB the PostgreSQL adapter for Python psycopg2 is used. As for workloads we query only the LineItem table, which consists of 16 columns.

B. Test Environment

For our tests we chose a random baseline, which means that we create indexes on a number n of random columns. We designed 3 tests with a varying number n of random columns participating in a query for each, i.e., n=2, n=4 and n=10, respectively. In addition, we decided to have 1 test for 5 TPC-H queries that use the line item table (Q1, Q4, Q6, Q15, Q18).

In a process of trial and error, guided by the use of a verification of variance with a t-test, we designed each test to have 50 iterations.

C. Results

Fig. 4 shows the results of our experimental evaluation, reporting the resulting cost given by the query optimizer for running the workloads after selecting 3 indexes. The indexes are selected by the different trained agents. The first observation is that a random selection of indexes is consistently the worst choice, leading to performance deterioration.

Next we can consider the heuristics and Q-Learning. Our heuristic consists on adding up the selectivities on each

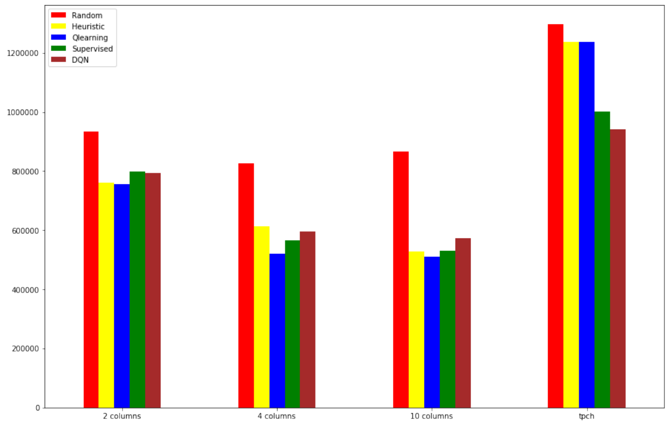

Fig. 4. Evaluation of Different Indexing Agents, with Different Number of Random Columns Participating in the 5-Query Workloads

columns across all queries, and then finding the columns that have the lowest total selectivity.

From the results we see that Q-Learning is always at least as good as the heuristic (since the heuristic is the starting point of the exploration). In our observations we see that heuristics performs overall well, but fails when two conditions happen:

1) *Skew in Column Selection:* When queries in the columns participating are randomly distributed (cases 2, 4, and 10), the heuristic we describe can work well. However, when a single column is used in several queries (TPC-H, where, for example a single column can appear in 80% of the queries), then even if the individual selectivities of the column per query are high (e.g. 0.8), the sum (or product) across all queries can low, hiding the low selectivties from columns that are used in less queries (e.g. if a column has a selectivity of 0.01 only on one query and of 1 on the rest).
2) *Skew in Query Runtime:* Since the heuristic is agnostic to the real cost of the query, which is fed to the environment. When a query has a higher runtime than others (TPC-H case), Q-Learning and other techniques that interact with the environment are in a better position to learn the best actions, whereas the heuristic would fail to capture this aspect.

Based on this observation we can see that for the TPC-H case the impact of the conditions we described is notable, making the heuristic perform worst than other approaches. Given that Q-Learning is only given a brief number of iterations, it can fail to find the optima in our study, and hence it at least is able to find a local optima close to the starting point given by the heuristic. Hence when the heuristic fails Q-Learning cannot improve over it as much as the other methods. For the case 4, the improvement of Q-Learning reaches a 20% better cost than the heuristic.

DQN and supervised learning have a competitive edge over Q-learning and other techniques, in the fact that they are trained for more cases and for a longer time. However in our results we find that when the heuristic performs well, both the heuristic and Q-Learning perform better than these specialized techniques. These results reinforce the observation that though machine learning methods hold a lot of promise, a skillful training and tracking of cases visited is required.

DQN and supervised approaches perform comparably across all approaches, suggesting that indeed the supervised technique is applicable to the task. This is specially interesting since the supervised technique can be trained in less time, and it suggests that we might be able to propose this approach as a novel application in the field.

Finally, we should make the observation that the evaluation that we report was designed to be able to include supervised learning, with its assumptions of giving a set of indexes at once, rather than one step after the other. Because of this we did not evaluate the cumulative reward that in a real reinforcement learning study we could have evaluated (i.e. the cost of the workload after adding each successive index, or after each step that the agent has learned). Instead we evaluated only at the last step. This change might have affected the observations that we have made with regards to Q-Learning and DQN.

V. Related Work

Automatic Index Selection: The AutoAdmin Project at Microsoft Research [14] was a pioneer in addressing the problem of automatic index selection. Authors aimed for more self-tuning and self-managing database systems with the key ideas that formed the basis of the Index Tuning Wizard (ITW). In particular, in their work they offered a "What-if" interface to a query optimizer, such that an index selection tool or a DBA was able to ask on the cost of running queries when using an index, in spite of the index not existing yet. Building on this interface authors proposed an index selection tool to enumerate, pick and recommended a set of indexes, by exploiting an index analysis tool which determines the "goodness" of the index sets and prunes the search space.

According to Schnaitter and Polyzotis [16], in the domain of index selection, commonly techniques are divided into online and offline ones. In the offline techniques, the DBA represents the workload, and the technique gives as result recommendations which the DBA can then use to choose indices. Consequently, the quality of the results can be effected by the selected workloads. In dynamic environments, in which the pattern of workloads is evolving, the offline approach can bring some challenges. To overcome these challenges [16] provided a semi-automated solution which falls between an ofline and online technique. They introduce WFIT, a Work Function algorithm for Index Tuning, which aims to keep a balance between online workload analysis and the DBAs feedback. In their proposal the online analysis enables the adaptation of recommendations to changes. These changes can be caused by the DBAs preferences or by workload evolution. Through feedback, the DBA can help the algorithm become more informed on the domain.

Later, Basu at al. [17] proposed to substitute the role of the DBA in index selection by an artificial intelligence agent based on reinforcement learning. Authors proposed an algorithm they called COREIL, which does not have a pre-defined cost model and through a dynamic programming approach that uses policy iteration, is able to learn and estimate the cost of individual queries. In their model the agent learns what index to suggest for one query at a time, and it also learns the penalty of building an index. In their work authors also propose ideas to deal with the dimensionality of the search space, by discarding indexes that will not reduce the cost.

Recently Sharma et al. [2] proposed a reinforcement learning complement to the work of Basu at al.. They consider a novel deep reinforcement learning solution, an idea that serves as a basis for our work; but authors do not add any specialization in terms of pruning the search space. Authors report performance gains, however their tests are only comparing against the extremes of indexing all or not indexing at all, which seemed unrealistic and motivated our experiments with a heuristic and other solutions.

Reinforcement Learning in Databases

The work of Basu at al. [17] mentioned previously was pioneering in employing reinforcement learning for database optimizations.

Currently there is early, ongoing work in using deep reinforcement learning, as exeplified in the work of Sharma et al. [2].

In this section we discuss two other works in the field.

One of the other investigated applications of deep reinforcement learning in the area of query optimization in relational databases is the identification of a good join order. The challenge is to search a large candidate space of join enumerations and finding the most cost-effective order. Existing query optimizers mostly rely on static strategies without learning from their previous experiences. Preliminary results of ReJOIN's [5] (learning based optimizer) reveals that a DRL-based approach can be made as good and often better than the Postgres SQL optimizer in terms of finding the optimal cost of join orderings.

Modern data processing applications can also be a challenge for query optimizers. The query plan of these applications contains different operators from different domains. Each of these operators has different physical implementations which performs differently. These diversity of implementations is a challenge to the query optimizer who must provide a good physical plan by choosing the best physical implementations for each operator. One general solution for this task is proposed by Kaftan et al., in their library called Cuttlefish [18]. The idea is to represent the selection of operators a reinforcement learning problem where the action never changes the state. Such problem can be solved with approaches easier than reinforcement learning, which are called multi-arm bandits. These solutions do not need to learn the long term value of an action, but instead learn the distribution of expected rewards from choosing an action given some simple context and no other precondition. The solution provided by Cuttlefish is shown to be adaptive to data characteristics without any predefined optimization rules or cost models. Also it supports different execution models, with and without the data context.

VI. Conclusions and Open Challenges

In this paper we introduced our proposal for using a supervised learning algorithm as a complement to deep reinforcement learning for automating physical database design tasks like the index creation process. Our goal was to contribute towards an automatic index creation as a learning problem in a generic way for future developments in DBMS.

We specifically design and train several models using the Postgres database to provide rewards based on the costs given by the query optimizer. We train on a set of randomly generated workloads. We observe that for TPC-H queries DQN can outperform all models, leading to 40% improvement over a random approach, and 30% over a heuristic. However, our work also shows that in random cases when heuristics are applicable, they can still outperform learned models. This suggests that a more careful engineering is required in the building and training of the models. Tracking and guiding the training process are important aspects that need to be considered for building a high quality solution. Furthermore the need for comprehensive evaluation remains a significant challenge in building these models, since they might prove to be good in one case but be insufficient for other cases.

Our observations also suggest that there could be an arrangement where heuristics coexist with learned models.

In future work we will compare against algorithmic implementations of the techniques behind index advisory tools. We will compare the inference time of our trained model with that of the tool, studying how feasible it is to use our solution on a live system. In addition we will develop a C++-based solution that uses SIMD operations or GPUs, to support a faster inference process, to make our tool more competitive. We will also work on improving the building process for the models. For this we seek to propose a more comprehensive evaluation (considering the intermediate steps) which could aid in the training.

References

[1] V. Leis, A. Gubichev, A. Mirchev, et al. How good are query optimizers, really? PVLDB, 9(3):204-215, 2015.

[2] Ankur Sharma, Felix Martin Schuhknecht, Jens Dittrich. "The Case for Automatic Database Administration using Deep Reinforcement Learning". CoRR 2018.

[3] Bogdan Rducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro adaptivity in Vectorwise. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13). ACM, New York, NY, USA, 1231-1242. DOI: http://dx.doi.org/10.1145/2463676.2465292

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

[5] Ryan Marcus and Olga Papaemmanouil. "Deep Reinforcement Learning for Join Order Enumeration". CoRR 2018.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, et al. Playing atari with deep reinforcement learning. CoRR, abs/1312.5602, 2013.

[7] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. A Brief Survey of Deep Reinforcement Learning. IEEE Signal Processing Magazine, 34(6):26-38, Nov. 2017.

[8] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. Nature, 529(7587):484489, 2016.

[9] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Broneske, Gunter Saake, Maya S Sekeran, Fabin Rodriguez, and Laxmi Balami. Gridformation: Towards self-driven online data partitioning using reinforcement learning. In Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, page 1. ACM, 2018.

[10] Guy Lohman. Is query optimization a solved problem. In Proc. Workshop on Database Query Optimization, p. 13. Oregon Graduate Center Comp. Sci. Tech. Rep, 2014.

[11] http://www.tpc.org/tpch/.

[12] https://github.com/keras-rl/keras-rl.

[13] https://gym.openai.com.

[14] Nicolas Bruno, Surajit Chaudhuri, Arnd Christian Knig, Arnd Christian Knig, Vivek Narasayya, Ravi Ramamurthy, Manoj Syamala, Nico Bruno. AutoAdmin Project at Microsoft Research: Lessons Learned. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering IEEE Computer Society December 1, 2011

[15] Surajit Chaudhuri, Vivek Narasayya. AutoAdmin What-If Index Analysis Utility. SIGMOD Association for Computing Machinery, Inc. June 1, 1998

[16] Karl Schnaitter, and Neoklis Polyzotis. "Semi-automatic index tuning: Keeping dbas in the loop." Proceedings of the VLDB Endowment 5.5 (2012): 478-489.

[17] Basu, Debabrota et al. "Cost-model oblivious database tuning with reinforcement learning." International Conference on Database and Expert Systems Applications. Springer, Cham, 2015.

[18] Kaftan, Tomer, et al. "Cuttlefish: A Lightweight Primitive for Adaptive Query Processing." arXiv preprint arXiv:1802.09180(2018).

[19] Chaudhuri, Surajit, and Vivek Narasayya. "Self-tuning database systems: a decade of progress." In Proceedings of the 33rd international conference on Very large data bases, pp. 3-14. VLDB Endowment, 2007.