Learning a Partitioning Advisor for Cloud Databases

Benjamin Hilprecht
TU Darmstadt

Carsten Binnig
TU Darmstadt

Uwe Röhm
The University of Sydney

Abstract

Cloud vendors provide ready-to-use distributed DBMS solutions as a service. While the provisioning of a DBMS is usually fully automated, customers typically still have to make important design decisions which were traditionally made by the database administrator such as finding an optimal partitioning scheme for a given database schema and workload. In this paper, we introduce a new learned partitioning advisor based on Deep Reinforcement Learning (DRL) for OLAP-style workloads. The main idea is that a DRL agent learns the cost tradeoffs of different partitioning schemes and can thus automate the partitioning decision. In the evaluation, we show that our advisor is able to find non-trivial partitionings for a wide range of workloads and outperforms more classical approaches for automated partitioning design.

ACM Reference Format:
Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2020. Learning a Partitioning Advisor for Cloud Databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3389704

1 Introduction

Motivation: Providing data solutions as a service is a growing field in the cloud industry. Cloud platforms such as Amazon Web Services or Microsoft Azure provide multiple ready-to-use scale-out DBMS solutions for OLAP-style workloads as a service. Using these services, customers can easily deploy a database, define their database schema, upload their data and then query the database using a cluster of machines. While the provisioning is usually fully automated, many design decisions which were traditionally made by the database administrator remain a manual effort. For example, in Azure's Data Warehouse but also in Amazon Redshift customers have to choose a partitioning attribute of a table to split large tables horizontally across multiple machines. Partitioning the database can dramatically improve the performance of analytical workloads since data-intensive SQL queries can be farmed out to multiple machines.

While partitioning a database in an optimal manner is a non-trivial task it has a significant impact on the overall performance. For example, analytical queries typically involve multiple joins over potentially large tables. If two tables are co-partitioned on the join attributes they can be joined locally on each node avoiding costly network transfers. Deciding for complex schemata with many tables and possible join paths which tables should be co-partitioned is a non-trivial task since this not only depends on the schema but also other factors such as table sizes, the query workload (i.e., which joins are actually important and how often tables are joined), or hardware characteristics such as network speed and of course the database implementation itself.

There exists already a larger body of work to automate the physical design of distributed DBMSs including the data partitioning [4, 24, 31]. These advisors formalize the problem as an optimization problem and thus rely on cost models to estimate the runtime of queries for different partitionings. However, this approach is unsuitable for a cloud providers: First, cloud providers typically allow customers to deploy their DBMS solutions on various hardware platforms which renders the problem of acquiring exact cost models a challenge on its own. Secondly, even if the cost model is tuned for a given hardware platform, optimizer cost estimates are still often notoriously inaccurate [16] resulting in non-optimal partitioning designs if existing automated design approaches are used as we show in our experiments.

Contributions: In this paper, we propose a different route and make the case to use Deep Reinforcement Learning (DRL) to realize a cloud partitioning advisor as a service that can be used for internal and external DBMS solutions. The advantage for DRL is that a DRL agent learns by trial and error, and thus they do not rely on the fact that an accurate cost model is available. Instead, by deploying different partitionings and observing query runtimes, they learn the tradeoffs for varying workloads. Once trained, our learned advisor can be queried to obtain a partitioning for the observed workload.

One could now argue that instead of learning a DRL agent, we could simply learn a neural cost model that predicts the runtime for different partitionings and then use an optimization procedure similar to [24] to select a suitable partitioning. Recently, learned cost models have also been used for query

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389704

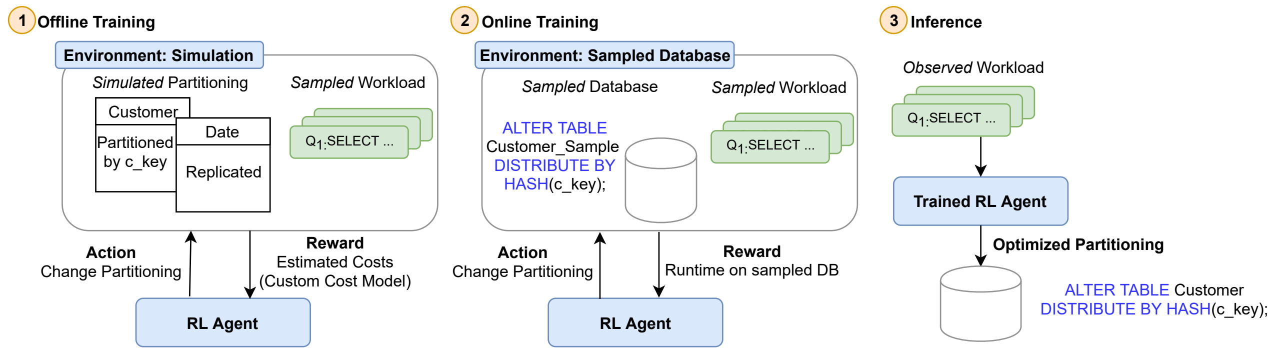


Figure 1: Overview of DRL-based approach to Learn a Cloud Partitioning Advisor.

optimization [20] or cardinality estimation [13]. The main reason why we use DRL for the partitioning problem is that it inherently addresses the exploitation vs. exploration tradeoff (i.e., it efficiently navigates the space of possible partitionings instead of relying, for example, on naïve random sampling from the space of possible solutions to collect training data). This is especially important for learned cost models, where collecting training data can be immensely expensive since a representative set of queries has to be executed over a potentially large database. In our case, the high training costs are amplified since we need to run the same set of queries over a representative set of different partitionings of the database where repartitioning itself is a costly operation. The exploration/exploitation behavior of DRL thus helps us concentrating on "promising" partitionings by exploitation and repartitioning intelligently if we explore (i.e., we try out promising partitions in the neighborhood first). This general advantage of RL was exploited in the data management community for similar problems as well [21, 36].

To summarize, we make the following contributions:

(1) We first formalize a framework that translates the partitioning problem to a DRL problem.
(2) We present a two-step learning procedure to efficiently reduce the training time of our DRL agent that first bootstraps a DRL agent offline (with a simple network-centric cost model) and then refines the agent online by actually running real workloads.
(3) Moreover, we propose an extension that makes use of a committee of DRL agents to improve the adaptivity of our approach for dynamic workloads. This also allows us to extend the advisor using an incremental approach if new queries are added to the workload or the database schema changes.
(4) In our evaluation, we show that our approach can handle a variety of different database schemata and workloads. We also compare our approach to classical optimization-based approaches and show that our approach is able to find non-obvious solutions that outperform classical optimization-based approaches even if accurate cost estimates would be available.

Outline: The remainder of this paper is organized as follows: First, in Section 2 we provide an overview of our approach to use DRL to learn a partitioning advisor. In Section 3, we formalize the partitioning problem as a DRL problem before we introduce our training procedures in Section 4. We then explain how to obtain partitionings at inference time in Section 6, explain optimizations for workload changes in Section 5 and present the results of our experimental evaluation in Section 7. Finally, we conclude with related work in Section 8 and a summary in Section 9.

2 Overview

The basic idea of this paper is to train a Reinforcement Learning (RL) agent for each cloud customer that learns the tradeoffs of using different partitionings for a given database schema for different workloads. Learning these tradeoffs is appealing since cost models are known to be notoriously inaccurate [16] and would thus over- or underestimate the benefits of certain partitionings. To this end, we propose to train a DRL agent that learns the tradeoffs of using different partitionings and thus can be used to suggest a partitioning for a given customer workload. An overview of our approach is depicted in Figure 1.

In order to make use of our learned partitioning approach, the customer only needs to provide the DBMS (schema and data) and a sample workload that reflects the set of typical queries in a production workload. Based on this information, we train a DRL agent in an offline- and online phase (step 1 and 2). After training, the DRL agent can then be used in the production DBMS to decide which partitionings to deploy by monitoring the actual workload (observed workload). Changes in the workload might then trigger the DRL agent to suggest new partitionings that are more suited for a given workload (without retraining the agent).

In the following, we describe the high-level design of our training procedure for the DRL agent which is the core contribution of this paper. A detailed discussion about the training procedure can be found in Section 4. In general, DRL agents learn by interacting with an environment by choosing actions and observing rewards which they seek to maximize. In our setup, the environment is the DBMS which the agent manipulates with actions that change the partitioning of individual tables. During the training phase, the agent learns to minimize the runtime of a given workload consisting of a mix of representative queries. In the training phase, the agents thus learns the effects of different partitionings on individual query latencies.

Naïvely, we could train the agent on the customer database directly but this would require a high effort to collect the training data. For instance, repartitioning a large database table can take several minutes to complete. During the training phase the agent requires several of these actions to learn the effects. We therefore separate the training process into two phases: (1) *offline* and (2) *online* training. In the offline training phase, the agent solely interacts with a "simulation" of the customer database. Since the network is typically the bottleneck of distributed joins, we developed a simple yet generic cost model focused on the network overhead required to answer a query given a certain partitioning. In combination with the metadata (schema and table sizes) about the customer database, we can estimate the query costs given a partitioning in our simulation. These estimates are used as rewards for the agent. Though not precise, this bootstraps the agent and enables it to already find a reasonable partitioning given a production workload (i.e., a mix of SQL queries). In our experiment, we show that a DRL agent using this approach is already able to find partitionings that are on par with traditional optimization-based partitioning advisors that rely on DBMS internal cost models.

In an optional online training phase, the agent then does not just interact with a simulation but with a real database. However, instead of using the complete database we only use a sample of the data to speed-up this step of the training phase. The benefit of this phase is that it does not depend on the accuracy of our simple network-centric cost model anymore. Instead, we can simply measure the runtimes of queries on the sampled database to compute the rewards of the agent. Consequently, the agent learns the effects of partitionings more accurately.

Once the training is completed, we finally use the agent to make actual partitioning decisions. As input, it requires a workload, i.e. which queries were submitted in a certain time window. Based on this workload, the agent suggests partitionings which we deploy on the actual customer database. In many cases, the agent can be used directly to suggest an optimized partitioning without any further training. However, in case the database schema changes or completely new classes of queries occur in a workload our advisor needs to be retrained. In order to optimize for this case, we provide an incremental training procedure that we discuss in Section 5.

3 Partitioning as a DRL Problem

As discussed before, in this paper we use DRL to tackle the partitioning problem of databases. While DRL is typically used for sequential decision making, it has successfully been applied to solve classical combinatorial optimization problems [5, 12, 26] as well. The intuition of this paper is similar since the partitioning problem is indeed a combinatorial optimization problem. The main reason why RL has proven to be beneficial when applied to optimization problems is that it efficiently tackles the exploitation vs. exploration tradeoff (i.e., it more efficiently navigates the space of possible solutions instead of relying, for example, on naïve greedy search). This is especially important in our domain where collecting training data can be extremely expensive since a representative set of queries has to be executed over a potentially large database. We now discuss the required background on Deep Reinforcement Learning (DRL) before we show how the partitioning problem can be formulated as a DRL problem including how we featurize the DBMS schema and a SQL workload.

3.1 Background on DRL

In Reinforcement Learning (RL), an agent interacts with an environment by choosing actions. Specifically, at each discrete time step t, the agent observes a state s_t. By choosing an action $a \in A$, it transitions to a new state s_{t+1} and obtains a reward r_t. Mathematically, this can be modeled as Markov decision process. The way the agent picks the actions depending on the state is called policy π. The goal of the agent is to maximize the rewards over time. However, the greedy policy, i.e. selecting the action with the highest immediate reward, might not be the best strategy. Instead, the agent might select an action that enables higher rewards in the future. Consequently, when selecting actions the agent should always keep the long-term rewards in mind [34].

One approach to solving this problem is Q-learning. With the Q-function, the expected discounted future rewards are approximated as follows if we pick action a at state s: $Q(s, a) = \mathbb{E}\left(\sum_{t=0}^{\infty} r_t(s_t, a_t)\gamma^t | s_0 = s, a_0 = a\right)$. In Q-learning, the rewards are discounted with a factor $\gamma < 1$ to account for a higher degree of uncertainty for future states. The Q-function is learned during training. Note that if the approximation is good enough we can choose an optimal action for a state s as $\operatorname{argmax}_{a \in A} Q(s, a)$.

During training we also have to select random actions such that there is a tradeoff between exploration and exploitation what we have learned so far. Usually, exploration is realized by picking a random action with probability ϵ.

This probability is decreased over time [34] by multiplication with a factor called *epsilon decay*.

There are different ways of realizing the Q-function. For Deep Q-learning [23] (or Deep Reinforcement Learning), a neural network $Q_\theta(s, a)$ with weights θ is used for the approximation. Having observed a state s_t and an action a_t, the corresponding immediate reward r_t and the future state s_{t+1} the network is updated via Stochastic Gradient Descent (SGD) and the squared error loss $(r_t + \gamma \operatorname{argmax}_{a \in A} Q(s_{t+1}, a_t) - Q(s_t, a_t))^2$. The intuition is that the expected discounted future rewards when selecting action a_t in step t should be the immediate reward r_t together with the maximum expected discounted future rewards when selecting the best action a in the next step $t+1$ discounted by γ, i.e. $\operatorname{argmax}_{a \in A} Q(s_{t+1}, a_t)$.

3.2 Problem Modeling

In order to formulate the partitioning problem as a DRL problem we model the database and the workload as state and possible changes in the partitioning as actions. Rewards correspond to the gain in performance for a given workload. During training, the agent thus learns the impact of different partitionings on the workload. Figure 2 shows an example of our encoding for a simple database with three tables and a workload with two queries.

Partitioning State: The most important part of the state is to model a partitioning for a given database. For simplicity, we assume that only one partitioning scheme is used (e.g., hash-partitioning) that horizontally splits a table into a fixed number of shards (which is equal to the number of nodes in the database cluster). Moreover, replicated tables are also copied to all nodes in the cluster. In fact, these are the partitioning / replication options supported by the two DBMSs we used in our evaluation. However, in general our approach can easily be extended to more complex partitioning schemes as well. Following the assumptions that a table T_i can either be replicated or alternatively partitioned by one of its attributes $a_{i1}, a_{i2}, \ldots, a_{in}$, we can encode the state as a binary vector using an one-hot encoding $s(T_i) = \left(r_i, a_{i1}, a_{i2}, \ldots, a_{in}\right)$, where r_i encodes whether a table is replicated and the remaining bits indicate whether an attribute is used for partitioning. For instance, if the `part` table in Figure 2a is replicated, its state vector is $(r_3, a_{31}) = (1, 0)$ whereas the `customer` table is partitioned by the attribute a_{21} and the resulting vector is $(r_2, a_{21}) = (0, 1)$ (as shown in in Figure 2b).

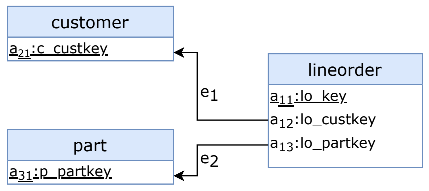
To reduce the exploration of sub-optimal partitionings, we further extend the state representation making it explicit which tables are co-partitioned, i.e., the partitioning attributes of the tables match their join attributes. For instance, if the `customer` and `lineorder` table in Figure 2 are partitioned by the attributes `lo_custkey` and `c_custkey` respectively, we can join them locally on each node without shuffling. To explicitly encode co-partitioning we introduce the concept of edges; i.e., if an edge between a pair of join attributes a_{ir} and a_{js} of the corresponding tables T_i and T_j is activated, it guarantees co-partitioning. For instance, since the edge e_1 in Figure 2b is active the `customer` and `lineorder` tables are co-partitioned. The fixed set of possible edges E can easily be extracted from the given schema and workload (i.e., all possible join paths). Since every edge can either be active or inactive, the edge states can be represented as a fixed-size binary vector. To represent the features for the partitioning of a database with multiple tables as input for our Q-Network, we append the state vectors of all tables. For instance, the edge vectors and individual table vectors of Figure 2b are appended in Figure 2c and fed into the Q-network. Since this input is of fixed length, we are able to use a feed-forward neural network to predict the Q-value.

Workload State: Moreover, we need to model the workload as part of the state since for the same database schema, different workloads result in different partitioning strategies that should be selected. Formally, a workload is a set of SQL queries $Q_1, Q_2, \ldots, Q_n$. One way to model the workload is to encode each query using different one hot encoded vectors, i.e., one vector for the set of tables, join predicates, where conditions etc., similar to [13, 33]. However, this modeling approach assumes that only queries of a typical pattern occur (e.g., queries without nesting) and thus this approach is not suited for our approach since a partitioning advisor should be trained on arbitrary workloads where the query patterns are not known in advance and complex queries involving nested queries and complex predicate conditions appear.

Encoding nested queries with the featurization as proposed in [13, 33] would be in general possible but result in an overly complex encoding with many more input vectors and a neural network structure which requires an extensive training. However, a more complex encoding is still only able to represent a fixed class of queries. Moreover, more complex encodings typically require orders of magnitude more training data.

We thus take a different route to featurize the workload based on the observation that OLAP workloads are typically composed of complex but *recurring* queries. We assume that a representative set of possible queries q_i in a workload of queries Q is known in advance which is not uncommon in OLAP workloads. To encode a specific workload, we use a vector where an entry encodes the current normalized frequency f_i of a query q_i: $s(Q) = \left(f_1, \ldots, f_m\right)$. That way, the input state can represent different query mixes. For example, since the query q_2 occurs twice as often as the query q_1 the frequency vector becomes $(0.5, 1)$ in Figure 2b.

Moreover, completely new queries can be supported in our state encoding without the need to train a new DRL agent



q1: SELECT * FROM customer c, lineorder l
WHERE l.lo_custkey=c.c_custkey;

q2: SELECT * FROM part p, lineorder l
WHERE l.lo_partkey=p.p_partkey;

(a) Database and Workload

Foreign-Key Edges:
Edge e_1 for `lo_custkey`$\rightarrow$ `c_custkey`: active
Edge e_2 for `lo_partkey`$\rightarrow$ `c_partkey`: inactive
$s(E) = (e_1, e_2) = (1, 0)$

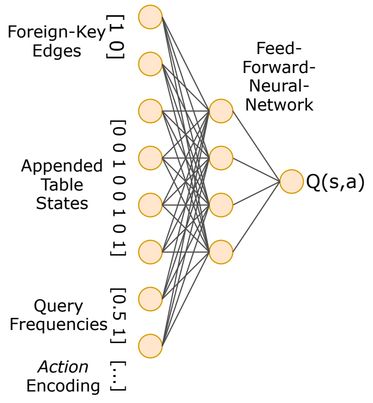
Table States:
`lineorder` partitioned by `lo_custkey`
$s(lineorder) = (r_1, a_{11}, a_{12}, a_{13}) = (0, 0, 1, 0)$

`customer` partitioned by `c_custkey`
$s(customer) = (r_2, a_{21}) = (0, 1)$

`part` replicated
$s(part) = (r_3, a_{31}) = (1, 0)$

Query Frequencies:
q_2 occurs twice as frequently as q_1
$s(Q) = (f_1, f_2) = (0.5, 1)$

(b) State Representation



(c) Q-Network with Encoded State

Figure 2: State Representation of Simplified SSB Schema and Workload.

from scratch. One case that we typically see in analytical workloads is that the same query is used with different parameter values resulting in different selectivities. In order to support this case, we bucketize queries into classes with different selectivity ranges and use different entries in $s(Q)$; i.e., one for each bucket. That way, if a query is used with a new set of parameter values, it is supported by finding the corresponding entry in $s(Q)$ and increasing the query frequency f_i. For supporting completely new SQL queries and not just new parameter values of existing queries in the workload, we provide entries in $s(Q)$ that are initially set to 0 (i.e., no query of this type occurs in the workload) and use those entries for new queries if they occur. We support this case in our approach by using a committee of DRL agents that we can extend incrementally. As we show in our experiment in Section 7, the time required for this is only a small fraction of the original training time.

Actions: A small state space is essential to apply Q-learning because we have to compute the Q-values for all possible actions to decide which action to execute in a state. We designed the actions to affect at most the partitioning of a single table. More precisely, we support two types of actions: (1) partitioning a table by an attribute or (2) replicate a table. During training, the RL agent can only select one of these actions at each step. This reduces the repartitioning costs during training since similar partitionings are observed successively.

In addition, we provide an action for (de-)activating edges as a short-cut to change the partitioning. Intuitively, activating an edge co-partitions two tables while the de-activation of edges allows follow-up actions to choose a new strategy (e.g., replication discussed above). It is important that the set of edges to be activated is conflict-free. For this, we solely allow to activate an edge if there are no two edges which require a table T_i to be partitioned by different attributes a_{ir} and $a_{ir'}$. For example, edge e_2 cannot be activated in Figure 2 because e_1 is already active. First, the conflicting edge e_1 would have to be deactivated.

An action a is encoded similarly to the partitioning and workload state: we use appended one-hot encoded vectors to capture the information required for an action, i.e., the kind of action (replicate, partition, (de-) activate an edge etc.), the affected table and attribute as well as the (de-)activated edge. Both the state s and an action a are then used as input for the neural network to predict the Q-value $Q(s, a)$.

Rewards: The overall goal of the learned advisor is to find a partitioning that minimizes the runtime for the workload mix (queries and their frequencies) modeled as part of the input state. This objective has to be minimized by the DRL agent and can be used as a reward. Estimates of the simple network-centric cost model $c_m(P, q_i)$ for the queries q_i given a partitioning P are used for the offline training and actual runtimes $c_r(P, q_i)$ for the online training. Since the DRL agents seeks to maximize the reward, we use negative costs in the reward definition resulting in $r = -\sum_{j=1}^{m} f_j c(P, q_j)$.

We decided to exclude the costs of repartitioning the database as rewards into our learning procedure since we aim for setups where we expect that repartitioning does not happen that often and can be executed in the background especially for OLAP workloads and thus does not have a negative effect on the actual workload execution. In case repartitionings should be used more frequently, these cost should be included into the rewards to prefer repartitionings that can be applied with less cost.

4 Training Procedure

In the following, we discuss the details of the offline and the online phase of our DRL-based training procedure. At the end of this section, we further discuss optimizations of our approach that allow us to provide a higher accuracy for changing workloads (i.e., if the frequency of queries change) and to incrementally add new unseen queries (and tables).

4.1 Phase 1: Offline Training

For training a partitioning advisor, the DRL agent interacts with the state reflecting the current partitioning by selecting different actions and observing rewards as described before in Section 3. During the offline training phase, the database partitioning is simulated and the runtimes are estimated using our network-centric cost model $c_m(P, q_i)$ approximating computation and network transfer costs of a given query q_i for a partitioning strategy P. In particular, similar to an optimizer the cost model enumerates different join orderings. For each individual join in the plan, it estimates the optimal join strategy (symmetric repartitioning join, symmetric repartitioning join, broadcast single table or co-located join) and the resulting network and computation costs. The sum of the costs is finally returned as cost estimate for the query. In our experiments, we show that based on this simple network-centric cost model, we can already train an DRL agent that is able to suggest reasonable partitionings.

Using our simple network-centric cost model as well as the state/action representation introduced before, we can train the DRL agent as described in Algorithm 1. The training is divided into sequences of states and actions of length $t_{\max}$ called episodes. The selected actions change the partitioning used for the cost estimation $c_m(P, q_i)$. Similar to typical RL implementations, the DRL agent returns to the state s_0 at the end of every episode. To guarantee that the DRL agent can find a suitable partitioning we have to make sure that it can reach any other partitioning within $t_{\max}$ steps starting from the initial partitioning s_0. Since for every table we need just one action to partition it by any attribute or to replicate it, any state can be reached within at most $|T|$ actions (where $|T|$ denotes the number of tables in the schema). Hence, we need to set $t_{\max} \geq |T|$. However, as $t_{\max}$ influences the training time it is also a hyperparameter and can similarly be tuned.

4.2 Phase 2: Online Training

In contrast to offline training, the idea of online training is to deploy the partitionings P_i on a database cluster and measure the true runtimes $c_r(P_i, q_i)$ to compute the reward. However, the naïve approach is way too expensive to be used in practice. Imagine for example that we need 1200 episodes for training each having $t_{\max} = 100$ steps. Assume we have just a few queries and a small schema such that the total workload takes around 20 minutes and the repartitioning takes another 20 minutes on average. If we simply executed every action, i.e., we repartition the tables and measure the workload runtimes on a cluster, we would end up with a runtime of $(20mins + 20mins) * 1200 * 100 \approx 9years$.

Therefore, online training is intended to (only) serve as refinement in addition to offline training. This has no effect if we use the same degree of exploration, i.e. if we choose

Algorithm 1 Offline Training

```
 1: Randomly initialize Q-network $Q_\theta$
 2: Randomly initialize target network $Q_{\theta'}$
 3: for e in 0, 1, ..., $e_{\max}$ do                          ▷ Episodes
 4:     Reset to state $s_0$
 5:     for t in 0, 1, ..., $t_{\max}$ do                   ▷ Steps in Episode
 6:         Choose $a_t$ = $\text{argmax}_a\, Q_\theta(s_{t+1}, a)$ with
              probability $1 - \varepsilon$, otherwise random action
 7:         Execute action $a_t$ (i.e., simulate what the next
              state $s_{t+1}$ and partitioning $P_{t+1}$ would be)
 8:         Compute reward with cost model $c_m$:
              $r_t = \sum_{j=1}^{m} f_j c_m(P_{t+1}, q_j)$
 9:         Store transition $(s_t, a_t, r_t, s_{t+1})$ in B
10:         Sample minibatch $(s_i, a_i, r_i, s_{i+1})$ from B
11:         Train Q-network with SGD and loss
        $\sum_{i=1}^{b}(r_i + \gamma\, \text{argmax}_{a \in A}\, Q_{\theta'}(s_{i+1}, a) - Q_\theta(s_i, a_i))^2$
12:     Decrease $\varepsilon$
13:     Update weights of target model: $\theta' = (1 - \tau)\theta' + \tau\theta$
```

random actions with the same probabilities $1 - \varepsilon$. Note that ε is multiplied with a certain factor called *epsilon decay* after every episode to decrease it over time. For online training, we start with the ε value that we would reach after 600 episodes (i.e. half of the usual amount of episodes) in the offline phase. This already significantly reduces the training costs as we will show in our experiments. However, this does not suffice to effectively reduce the time of the online phase in practice. We therefore use further optimizations which aim to minimize the online training time of the DRL agent as discussed next.

Sampling: Instead of using all tuples of a database, we just use a sample for every table. This speeds up both the runtime of the queries and the time needed to repartition or replicate any table. In addition, we found it useful not to use the runtimes of a query $c_r(P, q_i)$ directly but to multiply this with a certain factor for every query. The intuition is that some queries scale better than others on the full dataset. Hence, runtime improvements of queries that scale better and thus also run fast on the full dataset should weigh lower than improvements of queries which are very slow on the full dataset. To this end, we measure the runtimes of each query q_i for the partitioning $P_{offline}$ found in the offline phase once for the full dataset $c_{full}(P_{offline}, q_i)$ and once for the sample $c_{sample}(P_{offline}, q_i)$. Afterwards, we scale the costs for each query q_i with the corresponding factor $\mathcal{S}_i = \frac{c_{full}(P_{offline}, q_i)}{c_{sample}(P_{offline}, q_i)}$.

One question is how many tuples have to be sampled per table, i.e. how the sampling rate is chosen. Higher sampling rates result in a longer runtime of the online phase since both the query runtimes as well the repartitioning times will increase. In contrast, smaller sampling rates might lead to suboptimal partitionings. This can happen if partitionings

P' have shorter weighted runtimes $\mathcal{S}_i c_{sample}(P', q_i)$ on the sample than a superior partitioning P^* for the full dataset. We can account for these cases by selecting several partitionings $P_1, \ldots, P_n$ and measure their runtime both for the sample and for the full dataset. If partitionings with shorter weighted runtimes on the sample also lead to shorter runtimes on the full dataset size the sampling rate is sufficient. If not, the sample size has to be increased. As a simple heuristic one can empirically determine a threshold below which table sizes should not fall below after sampling. This guarantees that tables have a certain minimum size. If this threshold is large enough, optimal partitionings on the samples will also be optimal on the full dataset with high probability. A cloud provider could empirically determine this threshold for every database and hardware setup.

Query Runtime Caching: If the DRL agent visits two states s_i and s_j during training which have the same corresponding partitioning P, the runtimes do not have to be measured twice. Hence, we can cache query runtimes to faster compute recurring reward values. Additionally, if the partitionings of the states s_i and s_j differ only for a certain set of tables $\{T_{i1}, T_{i2}, \ldots, T_{in}\}$ we only have to measure the runtimes of queries q_i that contain at least one of these tables. In particular, the runtime of every query q_i containing the tables $\{T_{i1}, T_{i2}, \ldots, T_{in}\}$ depends only on the states of these tables, i.e. $s(T_{i1}), s(T_{i2}), \ldots, s(T_{in})$. Hence, for every query we can maintain a table containing the different state combinations $s(T_{i1}), s(T_{i2}), \ldots, s(T_{in})$ and the runtime of the query on the sample dataset. In summary, when visiting a new state we examine the state of every table $s(T_i)$; i.e. whether it is replicated or hash-partitioned by a certain attribute, and run only the queries q_i for which we do not have a runtime entry for the state combination of relevant tables $s(T_{i1}), s(T_{i2}), \ldots, s(T_{in})$.

Lazy Repartitioning: The approach of lazy repartitioning is to keep track of the partitioning deployed on the database P_{actual} and the partitioning P_t of the state s_t the agent is currently at. Every time the agent chooses an action and we reach a new state we first check which queries $\{q_{j1}, \ldots, q_{jn}\}$ have to be executed on the database. Especially in later phases of training this will be significantly fewer queries than the full set Q since many runtimes will be in the Query Runtime Cache. For this set we determine the set of tables $\{T_{i1}, \ldots, T_{im}\}$ which are contained in these queries. Only if P_{actual} and P_t do not match for one of the tables, we actually repartition the table.

Timeouts: The idea of this optimization is that a partitioning where a single query exceeds a certain time limit cannot be optimal. Hence, we can safely abort the query execution and move on with training. Recall that the reward for a partitioning P for online training is defined to be $r = -\sum_{j=1}^{m} f_j \mathcal{S}_j c_{sample}(P, q_j)$. We can similarly compute the (online) reward $r_{offline}$ of the partitioning $P_{offline}$ found in the offline phase. If a query q_i takes longer than $-r_{offline}/(\mathcal{S}_i \cdot f_i)$ we can safely abort it since the corresponding partitioning will definitely result in a lower reward. If we are aware of a partitioning with an even higher reward r', the timeout can further be reduced to $-r'/(\mathcal{S}_i \cdot f_i)$.

5 Optimizations for Workload Changes

In the following, we discuss two enhancements for training the partitioning advisor: (1) using a committee of experts rather than a single agent to further increase the capacity for workloads with many tables and queries, (2) using incremental training to adjust a learned advisor if new queries occur in the workload.

Committee of Experts: The main goal of our approach is to train a DRL agent just once such that it generalizes over different workload mixes (i.e., different query frequencies). If the workload mix changes, we want to use inference of the trained DRL agent and obtain a new partitioning that works better for the new workload mix.

A more advanced approach enabling more accurate results for a wide variety of workloads (i.e., large query sets) is not to train only a single RL agent to suggest partitionings for all possible frequency vectors but to use several expert models for subsets of all possible workload mixes. Using more models allows experts to specialize on certain aspects of the problem and moreover increases the overall capacity of the model. The related ensemble approach is a common optimization in machine learning to optimize the model performance. The question is how the workload space can be partitioned efficiently into different expert models. In the following, we explain our approach called *DRL subspace experts*.

The main idea of DRL subspace experts is to first obtain so called reference partitionings $\tilde{P}_1, \cdots \tilde{P}_n$ which are optimized for certain workloads. To find these, we use the inference procedure of the naïve model (i.e., the RL agent which was trained for the whole workload space) and ask this agent for the optimal partitioning using m frequency vectors where one query q_i is over-represented: $f_1, \ldots, f_{i-1}, f_i, f_{i+1}, \ldots, f_m$ with $f_j = f_{low}$ for $j \in \{0, 1, \ldots, i-1, i+1, \ldots, m\}$ and $f_i = f_{high}$. The main intuition is that individual queries might favor opposing partitioning strategies that we aim to simulate by "extreme" frequency vectors. Since many queries share the same reference partitioning, the number of distinct partitionings n is much smaller than the number of queries m (i.e., $n << m$). The distinct partitionings resulting from this step is the set of reference partitionings $\tilde{P}_1, \cdots \tilde{P}_n$.

For example, for a given workload with 10 queries we would sample 10 frequency vectors each representing a workload were one query is over-represented. We then use these to obtain the reference partitionings from a naïve model with only one agent. Based on these 10 frequency vectors, we might end up having just three different reference partitions $\tilde{P}_1, \tilde{P}_2$ and $\tilde{P}_3$.

Once we determined the reference partitionings, we can separate the workload space, i.e. the set of different frequency vectors. We say that a frequency vector $(f_1, \ldots, f_m)$ belongs to the frequency subspace of one of these reference partitionings $\tilde{P}_i$ if $\tilde{P}_i = \text{argmax}_{\tilde{P} \in \{\tilde{P}_1, \cdots \tilde{P}_n\}} - \sum_{j=1}^{m} f_j \mathcal{S}_j c_{sample}(\tilde{P}, q_j)$, i.e., if the reward of the naïve RL agent is the maximum among $\tilde{P}_1, \cdots \tilde{P}_n$ for this frequency vector. Afterwards, we then train one DRL agent for each of these subspaces. The resulting DRL agents can be considered experts for their frequency subspace. For each of the frequency subspaces the training is similar to training the DRL agent for the naïve approach. The only difference is that the DRL agents are only trained for frequencies of their dedicated subspace. One problem is how to sample more frequency vectors from the same subspace. To obtain frequency vectors for different subspaces, we sample frequencies uniformly and assign each frequency vector to the DRL agent for the respective reference partitioning $\tilde{P}_i$.

An important aspect is that the training of these subspace expert models does typically not require any actual execution of queries on the database cluster since we can reuse query runtimes in the Query Runtime Cache of the naïve approach. When training the subspace expert models, however, we might encounter partitionings that were not seen when training the naïve model. For these cases, we have no entries in the Query Runtime Cache and the queries need to be actually executed. However, these cases are rare since the naïve agent visits all optimal or near-optimal partitionings with high probabilities already.

Incremental Training: A final interesting aspect of our online approach is that we can easily support new queries by incremental training. The main idea is that if new queries are added to a workload, we do not have to train a new model from scratch. Instead, we add new inputs representing the query frequencies to the input state of the naïve model and retrain it only with frequency vectors that include the new queries. Again, the Query Runtime Cache can be reused and we only require actual runtimes for the new queries. Afterwards the naïve model can be used again to obtain the new reference partitionings. Only if a new reference partitioning is found, we have to train a new expert agent for that subspace. Otherwise, it is sufficient to refine the existing subspace experts with the cached query runtimes.

6 Model Inference

Having trained the learned partitioning advisor, we now describe how it can be used to suggest a partitioning. This can either be the case if an initial partitioning of the database should be suggested or the workload changes. We first assume that only one DRL agent is trained before we explain how the inference works if a committee of experts is used.

Inference with one DRL agent: We assume that a frequency vector is given that represents the current workload mix. The intuition is to fully exploit the knowledge of the trained agent by always selecting the partitioning action with maximum expected future rewards, i.e. the highest Q-value.

When applying the inference procedure, we always start with the same initial state s_0 also used during training. From the initial state s_0, we iteratively choose the action that maximizes the Q-function, i.e., $a_t = \text{argmax}_a\, Q_\theta(s_{t+1}, a)$. For this, we enumerate all possible actions in that state and evaluate the neural network for each action. Since we designed the action space to be small, this is very efficient. Every time we choose an action, this changes the state s and thus the partitioning. Note that we do not have to deploy every state in this sequence. Instead, we use the same simulation that is also used in the offline phase. Consequently, we execute $t_{\max}$ actions and thus obtain a sequence of actions $(s_0, a_0, r_0, s_1, \ldots, s_{t_{\max}}, a_{t_{\max}}, r_{t_{\max}})$. Afterwards, we do not simply suggest the partitioning represented by the last state $s_{t_{\max}}$, since the DRL agent tends to oscillate around the best partitioning P^* (i.e, the partitioning with the highest reward is not necessarily represented by the last state). Instead, we identify the state s_t in the sequence above with a maximum reward and return the corresponding partitioning P^*.

Inference with a committee of DRL agents: If we want to obtain a new partitioning when a committee of experts was trained, we first determine which subspace $\tilde{P}_i$ of the frequency space the vector belongs to: $\tilde{P}_i = \text{argmax}_{\tilde{P} \in \{\tilde{P}_1, \cdots \tilde{P}_n\}} - \sum_{j=1}^{m} f_j \mathcal{S}_j c_{sample}(\tilde{P}, q_j)$. The DRL agent is selected by choosing the DRL agent for the reference partitioning with the lowest estimated runtime (which is the same procedure we use when training the expert models). Afterwards, we use the inference procedure discussed before with the corresponding expert model for $\tilde{P}_i$.

7 Experimental Evaluation

In the following, we evaluate the benefits of using learned partitioning advisors for databases with schemas of varying complexity. We study the following aspects of our approach:

(1) **Performance after Offline Training**. In the first experiment (Section 7.2), we validate that DRL agents that are trained purely offline find partitionings outperforming typical heuristics and are competitive with those found by state-of-the-art partitioning advisors.
(2) **Improvement due to Online Training.** Furthermore, if additionally trained online, the DRL agent clearly

outperforms state-of-the-art systems and finds non-obvious partitionings with superior runtime as we demonstrate in our second experiment (Section 7.3). We moreover study the isolated runtime savings of our suggested optimizations of the online phase.

(3) **Adaptivity to Data and Workload.** Another benefit of our approach is the flexibility w.r.t. changes in the workload (Section 7.4). Hence, in the third experiment we first show that the committee of experts can suggest partitionings that improve over the naïve model for changing workloads. Furthermore, we examine the additional training time required if new queries are added to a workload and the effect of database updates.

(4) **Other Learned Approaches.** We empirically validate that using DRL for the partitioning problem is superior to learning a neural cost model (Section 7.5) which is minimized for a given workload to find suitable partitionings.

(5) **Adaptivity to Hardware Characteristics.** Finally, in the last experiment (Section 7.6) we show that our agent can also adapt to changes in the deployment (i.e., if hardware characteristics change) which is not trivial with existing approaches.

7.1 Workloads, Setup and Baselines

For the experiments, we used different databases and workloads that we explain in the following. Moreover, we also discuss the learning setup that we used for training the partitioning advisors as well as the baselines.

Data and Workloads: We evaluated the partitioning advisor on three different database schemas and workloads varying in complexity: (1) As the simplest case, we used the Star Schema Benchmark (SSB) and its workload [25]. SSB is based on TPC-H and re-organizes the database in a pure star schema with 5 tables (1 fact and 4 dimension tables) and 13 queries. (2) The second database and workload we used was TPC-DS [3]. TPC-DS comes with a much more complex schema of 24 tables (7 fact and 17 dimensions tables) and 99 queries (including complex nested queries). For Postgres-XL, which is one of the systems used in the evaluation, only 60 of the 99 queries could be executed due to restrictions in which queries it supports. (3) In cloud data warehouses such as Amazon Redshift, customers are not required to use a star schema but can design an arbitrary schema for their database. To test how well our learned advisor can cope with more complex schemata which are not based on a star-schema, we additionally used the TPC-CH benchmark [11], which is the combination of the schema of the TPC-C benchmark with analytical queries of the TPC-H schema (adopted for the TPC-C schema). Originally, the TPC-CH benchmark combines analytical queries and transactions in a mixed workload. For the purpose of this paper, we only used the analytical queries to represent the workload in our evaluation. Furthermore, in the standard version of TPC-CH all tables can be co-partitioned by the `warehouse-id`. While our DRL agents also propose this solution when using the original TPC-CH schema, we do not think that such a trivial solution is realistic for many real-world schemata. Hence, we further added complexity and decided to restrict possible partitionings such that tables cannot be partitioned by `warehouse-id` only. For all benchmarks (SSB, TPC-DS, and TPC-CH), we used the scaling factor SF=100.

Parameter	Value
Learning Rate	$5 \cdot 10^{-4}$
τ (Target network update)	10^{-3}
Optimizer	Adam
Experience Replay Buffer Size	10000
Batch Size for Experience Replay	32
Epsilon Decay	0.997
$t_{\max}$ (Max Stepsize)	100
Episodes	600/1200
Network Layout	128-64
γ (Reward Discount)	0.99

Table 1: Hyperparameters used for DRL training.

Setup: The partitionings for different analytical schemas were evaluated on two database systems. To show that our learned approach is in general applicable to both disk-based and memory-based distributed databases, we used Postgres-XL 10R1.1 (a popular open-source distributed disk-based database) [2] and System-X (a commercial distributed in-memory database). For running the databases in a distributed setup, we used CloudLab [1], a scientific infrastructure for cloud computing research. For our experiments, we provisioned clusters of different sizes ranging from 4 to 6 nodes. Each node was configured to use 128GB of DDR4 main memory, two Intel Xeon Silver 4114 10-core CPUs and a 10Gbps interconnect. The partitioning advisor is built using neural networks implemented in Keras. In particular, the neural network to approximate the Q-functions used 2 hidden layers with 128 and 64 neurons, respectively. We used the standard ReLU activation function in every layer and a linear function for the output (to represent the Q-value) which is a common combination for DRL. An overview of all hyperparameters which we found to work best for training can be seen in Table 1. The only hyperparameter we changed for the different databases was the amount of episodes we used to train the model. Since SSB has a significantly lower amount of tables and queries we only trained the DRL agents for 600 episodes instead of 1200 episodes for TPC-DS and TPC-CH.

Baselines: Previous approaches typically use the optimizer cost estimates or heuristics to optimize the partitioning design [4, 24, 31]. We additionally compared the partitionings found by our approaches to heuristics that are typically used by a database administrator [35]. For both simple and more complex star schemata (SSB and TPC-DS) this means that usually fact tables are co-partitioned with either the most frequently joined dimension table (Heuristic (a)) or the largest dimension table (Heuristic (b)). For the more complex schema TPC-CH, we either naïvely replicated small tables and partitioned larger tables by primary key (Heuristic (a)) or greedily co-partitioned the largest pairs of tables while still replicating smaller tables (Heuristic (b)).

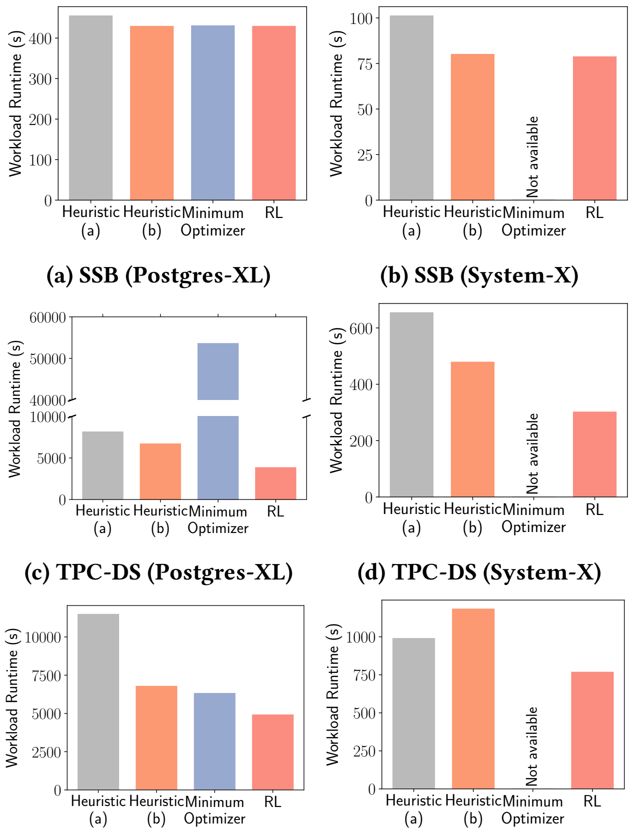
Automated partition designers [4, 24, 31] usually make use of the optimizer cost estimates, i.e. they enumerate different physical designs, let the optimizer estimate the costs for all queries in the workload and choose the partitioning candidate with minimal costs. While several optimizations exist that make the integration of the optimization and the database cost estimation tighter (e.g., Nehme et al. [24] make use of the MEMO data structure in Microsoft SQL server), they still suggest the partitioning with minimal query optimizer cost estimates. As a second baseline, we thus implemented a similar optimization algorithm enumerating candidate solutions and minimizing the optimizer cost estimates for Postgres-XL. However, for System-X this was not possible because the optimizer cost estimates are not accessible. Cloud providers offering multiple commercial DBMS systems face similar problems and thus this approach is not available for System-X.

7.2 Exp. 1: Offline Training

For each database mentioned before in the setup, we trained a dedicated DRL agent with offline training, i.e. using our simple network-centric cost model. We report the averaged total runtime of all queries for five runs for the partitionings suggested by our advisor and the baselines in Figure 3.

Results for SSB: For the SSB benchmark, the two heuristics co-partition the fact table with either the most frequently joined dimension table (`Date`) or the largest dimension table (`Customer`). The optimizer predicts minimal costs when partitioning the lineorder table by primary key and replicating all dimension tables. Our learned advisor also suggests to co-partition the fact table with the largest dimension table for Postgres-XL (same as Heuristic (b)). For System-X our learned advisor additionally suggests to partition the `Part` dimension table by its primary key leading to a minimal runtime improvement.

Results for TPC-DS: For TPC-DS, which is a more complex schema composed of several fact tables with shared dimensions, the DRL agents finds superior solutions that are non-obvious. Here, the improvements are more significant reducing the runtime over Heuristic (a) by approximately 50%. For both Postgres-XL and System-X, the DRL agents propose to co-partition the fact tables with a medium-sized dimension table, i.e. `Item`. This has the advantage that local joins are possible if two fact tables are joined, e.g. the

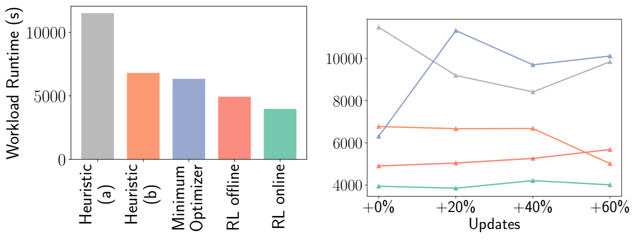


(a) SSB (Postgres-XL) (b) SSB (System-X)

(c) TPC-DS (Postgres-XL) (d) TPC-DS (System-X)

(e) TPC-CH (Postgres-XL) (f) TPC-CH (System-X)

Figure 3: Offline RL vs. Baselines.



(a) TPC-CH (b) TPC-CH with Updates

Figure 4: Online RL vs. Baselines.

fact tables for `StoreSales` and `StoreReturns`. Moreover, for System-X the `Customer` table is co-partitioned with the `CustomerAddress` table allowing local joins. In contrast, the partitioning with the minimal optimizer costs for Postgres-XL leads to a suboptimal partitioning. This is due to the high query complexity resulting in erroneous cost estimates.

Results for TPC-CH: As discussed before, TPC-CH uses a significantly more complex schema than SSB and TPC-DS since it is not similar to a star schema. While Heuristic (b) has better runtimes than Heuristic (a) on Postgres-XL, Heuristic (a) outperforms Heuristic (b) on System-X. This counter-intuitive result is due to the fact that partitioning a table by

Optimizations	Training Time	Speedup
None	4621h	-
+ Runtime Cache	1160.4h	4.0
+ Lazy Repartitioning	60h	19.3
+ Timeouts	33.4h	1.8
+ Offline Phase	13.3h	2.5

Table 2: Training Time Reduction of Optimizations.

district-id (as Heuristic (b) does) results in skewed partition sizes in System-X. Compared to the two heuristics, the DRL-agent proposes improved partitionings. For Postgres-XL it proposes to co-partition the Customer, Order, NewOrder and additionally the Orderline table by district-id but to replicate the Stock table. This avoids that Orderline has to be shuffled over the network for a join. For System-X, the DRL agent additionally partitioned the Stock table but also used a compound key combining warehouse-id and district-id to mitigate the skew (which was reflected in the simple network-centric cost model).

7.3 Exp. 2: Online Training

In this experiment we evaluate whether DRL agents trained online are superior over purely offline-trained agents. We focus on the most complex schema, i.e. TPC-CH, and Postgres-XL to analyze the additional online phase that leverages actual runtimes instead of cost estimates. For the online training we refine the DRL agent that was already bootstrapped with our simple network-centric cost model offline.

The runtime of the benchmark queries using the suggested partitionings on the full TPC-CH database are shown in Figure 4a. The partitioning suggested by the online-trained agent is 20% superior to the partitioning of the offline-trained agent. The online-trained DRL agent suggests a new partitioning where the NewOrder, Order and Orderline table are co-partitioned by Order-Id and the Customer table is replicated in addition. Interestingly, this partitioning has higher costs according to our simple network-centric cost model. However, the online phase is not affected by the inaccuracy of our simple network-centric cost model and was thus able to improve over the offline-trained agent.

If executed naïvely, the online training phase is time-consuming. We thus want to examine the effect of different optimizations. For this experiment, we were only running the training with all optimizations (except timeouts) activated. By keeping track of the queries that would be executed twice without *Runtime Caching*, as well as how often a table would be repartitioned without *Lazy Repartitioning* and how much time could be saved with a particular *Timeout*, we could determine the savings of the optimizations. As we can see in Table 2 every optimization significantly reduces the runtime and the largest improvement can be obtained with *Lazy Repartitioning*. The last optimization compares the training time of an agent that was bootstrapped in an offline phase with a randomly initialized agent.

The online-phase with all optimizations and for a model that was bootstrapped offline took 13.3 hours. We believe that a training time of several hours is acceptable since the model has to be trained only once for different workload mixes and can afterwards be used as a partitioning advisor if the workload changes (as we show in the next experiment). Moreover, especially in cloud setups, we can easily clone the instances. Hence, setting up a similar cluster to retrain the agent for several hours to obtain a refined model should be feasible considering that customers usually have one cluster provisioned all the time to do analytics. The cloning is especially efficient for our setup since we do not have to clone the entire data but only a sample of each table.

7.4 Exp. 3: Adaptivity to Data & Workload

The following experiments validate the adaptivity of a DRL agent to changing data and workloads. We first demonstrate that our approach can still find optimal partitionings without additional training even if the data and the mix of queries changes. Moreover, in a last experiment we examine the additional training time required if completely new queries are added to the workload.

Exp. 3a: Changing Data: First, we evaluated how robust the trained RL agent is if the data changes. In this experiment, we use the TPC-CH schema as before and train the RL advisor on the full database (100%). Afterwards, we update the DBMS and bulk load up to 60 % of new data into the TPC-CH schema. We use the bulk update procedure of TPC-H and transform the data to the TPC-CH schema since our main focus is on warehousing and the TPC-CH benchmark does not support bulk updates. Figure 4b shows the results of using our online-trained RL advisor (without any retraining) compared to all other baselines. The large deterioration of the "minimal optimizer" baseline in the measurement is due to different query plans chosen by the PGXL optimizer after updates are applied. As we can see, the partitioning found by the RL advisor constantly performs best even for relatively large update rates of up to 60%. However, if the database significantly changes, we need to retrain our advisor (which is not needed in this experiment though). A helpful indicator to decide when retraining is needed might be a change of the query plan. Moreover, there exists a huge body of work in ML to detect drifts in training data (which is related to this problem). Developing techniques to robustly detect when to retrain is an interesting avenue for future work though.

Exp. 3b: Changing Workload Mix: In this experiment we show that our learned advisor finds optimal partitionings for different query mixes. To this end, we trained an DRL agent with the naïve approach for different workload frequencies for the TPC-CH schema. Moreover, we additionally trained a committee of experts for the subspace experts approach as

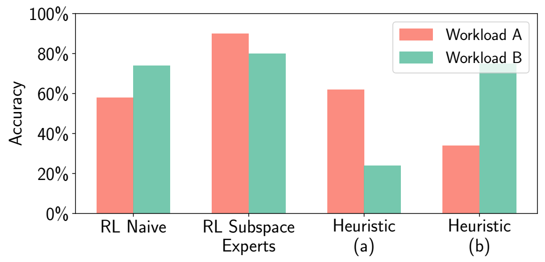


Figure 5: Best Partitioning found by Different Approaches for Varying Workloads (higher is better).

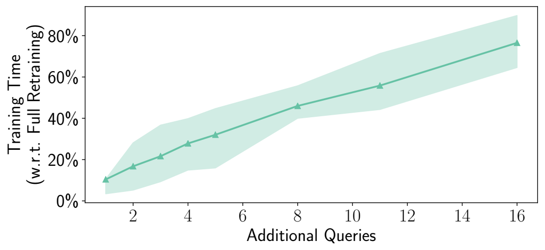
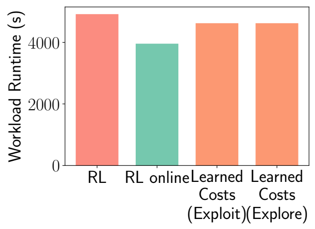


Figure 6: Training Time of Additional Training (relative to Full Retraining) with 25% and 75% Quantiles.

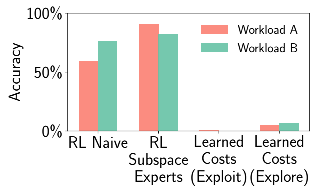
described in Section 5. In any case, the advisor only has to be trained once and generalizes to different workloads as we will show in this experiments. For the naïve model and the committee of experts, we both used an online training phase on Postgres-XL. Note that we can apply all optimizations for the online training as well. Moreover, we can reuse the Query Runtime Cache of the previous experiments if we train multiple experts.

After training both approaches, we report the percentage of correct partitionings for two different workloads clusters in Figure 5. Each cluster is a set of different frequency vectors (i.e., workload mixes): for cluster A the frequencies were sampled uniformly and for cluster B queries joining the `Stock` and the `Item` tables are more likely to occur. If the partitioning found by either approach is best for the respective cluster, we say that the approach has found the optimal partitioning for this workload mix. We compare the naïve approach and the subspace experts approach with two heuristics. Heuristic (a) always chooses the optimal partitioning found after online training in the previous Section. Heuristic (b) always chooses a partitioning where the `Stock` and `Item` tables are co-partitioned. The results are given in Figure 5. As we can see, the accuracy can significantly be improved when using subspace experts outperforming all other approaches. We conclude that is beneficial to divide the problem of finding an optimal partitioning for a given workload into subproblems which are then solved by the dedicated expert model. This is due to the well known technique of using ensembles of ML models to improve the performance.

Exp. 3c: New Queries: In our formulation of the problem we decided not to encode the complex nested queries typically



(a) TPC-CH Schema



(b) Workload Adaptivity

Figure 7: RL vs. Neural Baselines.

occurring in OLAP-workloads to avoid an overly complex neural network architecture requiring too much training data. Instead, we represent the workload as frequencies of a representative set of queries. Once trained, our learned partitioning advisor can suggest partitionings for any of those workloads. However, if completely new queries occur that have a significant impact on the workload runtime and do not have a similar query in the set of representative queries we need incremental training, i.e. we train the agent for workloads where these new queries occur which is significantly faster than training a new agent from scratch. In this experiment, we evaluate the additional training overhead if such new queries are introduced. In particular, it proves that additional training is much cheaper than training the agent from scratch if new queries occur.

We again trained a committee of experts for TPC-CH on top of Postgres-XL as the underlying database. However, in contrast to the previous experiment, we first randomly removed a fraction of the queries of the TPC-CH benchmark. We then retrained the advisor for the additional queries and calculated, with the help of already measured runtimes, how long such an additional training takes on average if part of the workload is not known initially.

Figure 6 shows the time for incremental training relative to the time required to train an DRL agent from scratch, depending on how many additional TPC-CH queries were added after the initial training. As we can see, the overhead of incremental training is much lower than training a partitioning advisor from scratch. This is because, similar to exploiting a bootstrapped DRL agent using the offline phase, we can start with a lower ε-value in the incremental training of the new naïve model resulting in fewer explorations. In addition, incremental training can also make use of the Query Runtime Cache, which keeps actual query execution to a minimum as many queries are already known.

7.5 Exp. 4: Other Learned Approaches

An alternative to using RL is to learn an ML model to predict the costs of a partitioning and use a classical optimization procedure to select the best partitioning for a given workload. Recently, learned cost models have been used for query optimization [20] or cardinality estimation [13]. For instance,

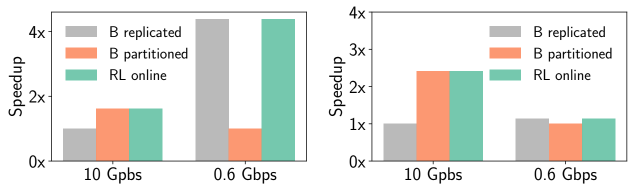
[20] iteratively choose optimal query plans according to their neural cost model. In the following, we explain how we implemented the neural cost model for partitioning.

Similar to our offline phase, we first use an offline bootstrapping step where the neural cost model is trained using runtime estimates based on our simple network-centric cost model. We use 100k workload/partitioning pairs for the offline phase since this is equivalent to the number of workload/partitioning pairs that our RL agent sees in its offline phase. Afterwards, analogous to our online training, we then run a workload mix on a real DBMS to improve the neural cost model using actual runtimes.

For the online training, we use multiple iterations, where we repartition the database to minimize the current cost model in every iteration. We then retrain the neural cost model with the runtimes collected on the partitionings observed during the iteration and then start the next iteration (i.e., sample a new workload and again minimize the cost model). To be fair, we allow the same overall training time for both approaches in the online phase (RL and the neural cost models) and also enable all optimizations we also use for our RL agent (e.g., runtime caching etc.). To simulate a more exploration-driven variant in contrast to the exploitation-driven variant above which selects the best partitioning in each iteration, we also implemented a variant that starts with a random partitioning in every iteration.

To show the efficiency we compare the runtime of the partitioning schemes suggested by the online-trained neural cost models, our online-trained RL agent and the RL agent that was only trained offline. In this experiment, we use the same workload (i.e., TPC-CH) as in Exp. 2. As a result, we can see in Figure 7a that the online-trained neural cost models (i.e., both the exploitation- and the exploration-driven variant) improve the offline-trained RL agent by only 6% while our online-trained RL shows an improvement of 20% compared to the offline-trained RL agent. Moreover, we also tested how well the neural cost model generalizes to new (unseen) workloads by using the same setup as in Exp. 3 where we sample new workloads uniformly. As we can see in Figure 7b the online-trained neural cost model only finds the optimal partitioning in 5% of the cases (vs. 91% for online-RL) for workload A and 7% of the cases (vs. 82% for online-RL) for workload B.

We investigated why the neural cost model approach does not perform as well as our RL agent in both experiments above. The reason is that our RL agent observed three times as many different partitionings as the learned cost model in the same training time. This effectively means that our RL agent explores partitionings with a lower average runtime and shows that the exploration/exploitation strategy of our RL agent actually leads to a more efficient navigation through the solution space.



(a) Standard HW **(b) Slower Compute**

Figure 8: Adaptivity to Deployment.

7.6 Exp. 5: Adaptivity to Deployment

Another advantage of using an DRL agent as partitioning advisor is that it can adapt the partitioning for different deployments which is an important scenario for cloud providers that allow customers to migrate their cluster to a new set of virtual machines with different characteristics. For showing the adaptivity of our learned advisor, we created a simple microbenchmark to empirically validate this. It consists of three relations `A`, `B` and `C` where `A` is a fact table and `B` and `C` are dimension tables. The relation sizes are inspired by the relation sizes of the `Lineorder`, `Order` and `Partsupp` table of the TPC-H benchmark. The workload consists of just two queries joining the fact table `A` with one of the dimension tables `B` or `C` with selectivities between 2% and 5%.

In the optimal partitioning, table `A` and `C` have to be co-partitioned because `C` is significantly larger than `B`. Depending on the network bandwidth, however, it might be optimal to either partition or replicate table `B`. For example, for a high-bandwidth network it might be beneficial to partition `B`, say, on its primary key. When joined with table `A` the scan of table `B` can be distributed among all cluster nodes (if the table is partitioned) and the remaining tuples have to be shuffled over the network. If table `B`, however, is replicated we do not have to send tuples over the network for the join but the scan is also not distributed across nodes. Hence, the question whether or not partitioning is beneficial depends on the speed of the network compared to the scan speed of the table. As network costs are more significant if one does not need costly disk accesses we decided to use System-X for the evaluation which is an in-memory database.

To show the effect, we used two different hardware deployments for System-X. One time, we used the usual 10 Gbps interconnect, one time we only used 0.6 Gbps interconnects. This is also the bandwidth offered for the basic deployment of Amazon Redshift. We trained one DRL agent on the full dataset (approx. 100 GB) for the two hardware deployments. In Figure 8 the effects of partitioning or replicating table `B` can be seen for both the slow and the fast network. In the Figure, we use the slowest approach of both as reference and show the speed-up of the others (i.e., higher is better). As we can see, for the slow network it is optimal to replicate table `B`, while for the fast network it is better to partition it.

We repeated the experiment on less powerful hardware (nodes with a 32-core AMD 7452 CPU and 128GB ECC Memory (8x 16 GB 3200MT/s RDIMMs)) in Figure 8b. In this case, the benefit of replicating table B is less significant for the slow network since the scan costs are more dominant. However, in all cases the DRL agent (after retraining the model on the hardware setup) suggests the optimal solution.

8 Related Work

Partitioning for OLTP and OLAP: Many approaches focus on transactional workloads [6, 7, 10, 28, 29]. In general, these approaches partition the data such that distributed transactions across nodes occur less frequently. For example, SCHISM [7] defines a graph consisting of tuples as nodes and transactions as edges and uses a min-cut to partition the tuples. Pavlo et al. [28] developed an alternative approach that is also capable of stored procedure routing and replicated secondary indexes. Fetai et al. focus especially on cloud environments [10]. For OLAP-workloads Eadon et al. [9] proposed REF-partitioning, i.e., to co-partition chains of tables linked via foreign key relationships. Since this technique can be exploited if a system supports hash-partitioning by any attribute most partitioning advisors and also our technique indirectly make use of REF-partitioning. Zamanian et al. [35] extend this approach such that even more locality can be obtained but at the cost of higher replication. For this, the database has to support predicate-based reference partitioning. In contrast, [19] iteratively improves the partitioning and relies on hyper-partitioning and hyperjoins. However, these features are currently not supported by Postgres-XL or System-X and could thus not be evaluated.

Automated Database Design: Automatic design advisors are an active area of research [4, 24, 30, 31, 31, 37]. However, many of these approaches [31, 37] focus only on single-node systems while only a few advisors for distributed databases are specialized on partitioning design [4, 24, 31]. These approaches, however, rely only on the cost model of the optimizer which is often inaccurate [16]. As in query optimization, this can result in wrong decisions [17] since the benefit of some query plans (partitionings) is over- or underestimated. Different from these approaches we developed a simple network-centric cost model and a dedicated online phase that is able to cope with inaccuracies of the cost model.

Even worse, some databases do not provide access to the cost estimates of the query optimizer at all such as System-X in our experiments. However, even databases offering cost estimates for query plans might not be suited for automated cost-based partitioning design since they do not provide a what-if mode for partitioning, i.e. the partitioning has to be actually deployed to obtain an estimate. This was the motivation for us to develop a simple network-centric cost model for partitionings to be used in the offline phase.

Another approach [30] optimizes both analytical and transactional workloads by partially allocating already partitioned tables in an optimal manner to minimize runtime or maximize throughput. Different from this approach, which is only focusing on the allocation, in this paper we provide a new solution to find a partitioning scheme which is an orthogonal problem to data allocation. Furthermore, the paper relies on an allocation heuristic which cannot take the actual execution cost into account. Marcus et al. [22] fragment tables and decide on replication and placement based on the how often they are queried. This strategy results in a custom partitioning scheme that is not supported by many databases and hence inapplicable for an automatic cloud partitioning advisor like ours. Moreover, it does not minimize network costs by leveraging local joins.

Recently, many approaches suggest to use machine learning to automate database administration and tuning [14, 27] and improve internal database components like join ordering [15] or cardinality estimation [13]. In particular, DRL [23, 32] was often used to tackle data management problems. For example Li et al. [18] focus on the scheduling problem for distributed stream data processing systems, Durand et al. [8] optimize the physical table layout or Zhang et al. [36] automate database configuration tuning. Different from those papers, we focus on data partitioning in distributed databases which was not yet considered.

9 Conclusion and Future Work

In this paper, we introduced a new approach for learning a cloud partitioning advisor based on DRL. The main idea is that a DRL agent learns its decisions based on experience by monitoring the rewards for different workloads and partitioning schemes. The agent is first bootstrapped using a simple network-centric cost model to make the training phase more efficient and afterwards refined with actual runtimes. In the evaluation, we showed that our approach is not only able to find partitionings that outperform existing approaches for automated partitioning design but that it can also adjust to different workloads and new queries. In the future, we plan to combine our approach with systems that predict future workloads to pro-actively re-partition the database as well as to decide whether the costs for repartitioning pay off in the long run.

10 Acknowledgments

The authors thank the anonymous reviewers for their helpful feedback. This research is supported by SAP, Oracle, Huawei, the DFG Training Group "Adaptive Preparation of Information from Heterogeneous Sources" (AIPHES, GRK 1994/1), the National Research Center for Applied Cybersecurity ATHENE as well as the DFG Collaborative Research Center 1053 (MAKI).

References

[1] CloudLab. https://www.cloudlab.us/.

[2] Postgres-XL database. https://www.postgres-xl.org/.

[3] TPC-DS benchmark. http://www.tpc.org/tpcds/.

[4] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 930–932. ACM, 2005.

[5] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017.

[6] K. Chen, Y. Zhou, and Y. Cao. Online data partitioning in distributed database systems. In *EDBT*, 2015.

[7] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3:48–57, 2010.

[8] G. C. Durand, R. Piriyev, M. Pinnecke, D. Broneske, B. Gurumurthy, and G. Saake. Automated vertical partitioning with deep reinforcement learning. In *New Trends in Databases and Information Systems, ADBIS 2019 Short Papers, Workshops BBIGAP, QAUCA, SemBDM, SIMPDA, M2P, MADEISD, and Doctoral Consortium, Bled, Slovenia, September 8-11, 2019, Proceedings*, pages 126–134, 2019.

[9] G. Eadon, E. I. Chong, S. Shankar, A. Raghavan, J. Srinivasan, and S. Das. Supporting table partitioning by reference in oracle. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1111–1122. ACM, 2008.

[10] I. Fetai, D. Murezzan, and H. Schuldt. Workload-driven adaptive data partitioning and distribution âĂŤ the cumulus approach. *2015 IEEE International Conference on Big Data (Big Data)*, pages 1688–1697, 2015.

[11] F. Funke, A. Kemper, and T. Neumann. Benchmarking hybrid oltp&olap database systems. *Datenbanksysteme für Business, Technologie und Web (BTW)*, 2011.

[12] E. B. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 6348–6358, 2017.

[13] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. 2019.

[14] T. Kraska, M. Alizadeh, A. Beutel, E. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. CIDR, 2019.

[15] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[16] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[17] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal*, 27(5):643–668, Oct 2018.

[18] T. Li, Z. Xu, J. Tang, and Y. Wang. Model-free control for distributed stream data processing using deep reinforcement learning. *Proceedings of the VLDB Endowment*, 11(6):705–718, 2018.

[19] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden. Adaptdb: adaptive partitioning for distributed joins. *Proceedings of the VLDB Endowment*, 10(5):589–600, 2017.

[20] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *Proceedings of the VLDB Endowment*, 12(11):1705–1718, 2019.

[21] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–4, 2018.

[22] R. Marcus, O. Papaemmanouil, S. Semenova, and S. Garber. Nashdb: An end-to-end economic method for elastic database fragmentation, replication, and provisioning. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1253–1267. ACM, 2018.

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[24] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1137–1148. ACM, 2011.

[25] P. E. OâĂŹNeil, E. J. OâĂŹNeil, and X. Chen. The star schema benchmark (ssb). *Pat*, 200(0):50, 2007.

[26] A. Paliwal, F. Gimeno, V. Nair, Y. Li, M. Lubin, P. Kohli, and O. Vinyals. Reinforced genetic algorithm learning for optimizing computation graphs. In *International Conference on Learning Representations*, 2020.

[27] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, et al. Self-driving database management systems. In *CIDR*, volume 4, page 1, 2017.

[28] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *ACM SIGMOD*, pages 61–72. ACM, 2012.

[29] A. Quamar, K. A. Kumar, and A. Deshpande. Sword: scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 430–441. ACM, 2013.

[30] T. Rabl and H. Jacobsen. Query centric partitioning and allocation for partially replicated database systems. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 315–330, 2017.

[31] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 558–569, New York, NY, USA, 2002. ACM.

[32] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[33] J. Sun and G. Li. An end-to-end learning-based cost estimator. *arXiv preprint arXiv:1906.02560*, 2019.

[34] R. Sutton. *Reinforcement learning : an introduction.* The MIT Press, Cambridge, Massachusetts, 2018.

[35] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *ACM SIGMOD*, pages 17–30, 2015.

[36] J. Zhang, Y. Liu, K. Zhou, G. Li, Z. Xiao, B. Cheng, J. Xing, Y. Wang, T. Cheng, L. Liu, M. Ran, and Z. Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 415–432, New York, NY, USA, 2019. ACM.

[37] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. J. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: Integrated automatic physical database design. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 1087–1097, 2004.