Unifying RL based learning framework for workload aware database partitioning

May 2023

1 Introduction

Partitioning is an important technique for organizing data in data storage systems. Modern analytics systems split data into blocks, allowing queries to skip data blocks that are not required and significantly reduce I/O costs. Query processing engines (e.g. SparkSQL, Hive, and Trino/Presto) and distributed database systems (e.g. Citus, CockroachDB) often partition data across multiple machines to parallelize complex query operations. However, producing an optimal partitioning is a non-trivial task, and a sub-optimal partitioning may degrade performance. For example, if two large tables are not co-partitioned in a distributed database, a join between them will result in expensive data shuffling across the network.

Numerous methods have been proposed in the past to find the best partitioning for a given database and workload [1–11]. Recent works have avoided using heuristics and cost estimation from the query optimizer by taking advantage of new developments in reinforcement learning (RL) to produce better partitionings. For example, Yang et al. [12] propose using RL to learn a qd-tree, a data structure that facilitates routing tuples to data blocks amenable to data skipping for a given workload. However, the partitionings produced by this work are not optimized for join operations. On the other hand, Hilprecht et al. [13, 14] train an RL agent that decides whether to partition a table on certain columns or replicate the entire table, while adhering to co-partitioning constraints on some pairs of tables. Nevertheless, they only assume hash-partitioning, which nullifies the data skipping benefit. Jointly optimizing the partitioning of multiple tables for data locality while retaining their data skipping capability remains unexplored.

For our project, we aim to develop a comprehensive learning framework that can efficiently generate partitionings to cater to both efficient data filtering and efficient distributed joins. Our framework consists of two main steps. In the first step, we use reinforcement learning to train an agent that constructs a modified version of qd-tree that supports hash-based partitioning for each table. The result of this step includes a policy network and a value network of every table, which are subsequently used in the next step. It is important to note that the tables remain unpartitioned at this stage. In the second step, we train another RL agent using a different MDP, which determines a partitioning scheme that optimizes both data locality and data skipping. The value network from the first step measures the trade-off between hash-based partitioning and range-based partitioning in this step. Finally, we apply hash-based partitioning on the tables following the policy obtained from the second step, followed by a range-based partitioning, conditioned on the existing hash partitioning, using the policy obtained from the first step.

We implemented the first step, which involves an agent that is trained to construct a qd-tree for a given workload and data. In this report, we details the steps taken to implement such an agent.

Furthermore, we report our experimental results on the trade-off between hash-based partitioning and range-based partitioning.

2 Problem Definition

Our learning framework tackles the following problem: within a specific system configuration (e.g. storage capacity, disk speed, number of machines, etc.), determine the optimal combination of hash-based and range-based partitioning schemes that maximizes both data skipping (for efficient filtering) and data locality (for efficient joining) for a given database schema and query workload.

We consider a system where a table can first be partitioned using a hash-based scheme on certain columns and then further partitioned using a range-based scheme on a different set of columns. We note that range-based partitioning can be used to generate partitions produced by a qd-tree by range partitioning on a column that denotes the qd-tree partition a row belongs to. Therefore, we use the term "range-based partitioning" to refer to both range-based and qd-tree-based partitioning.

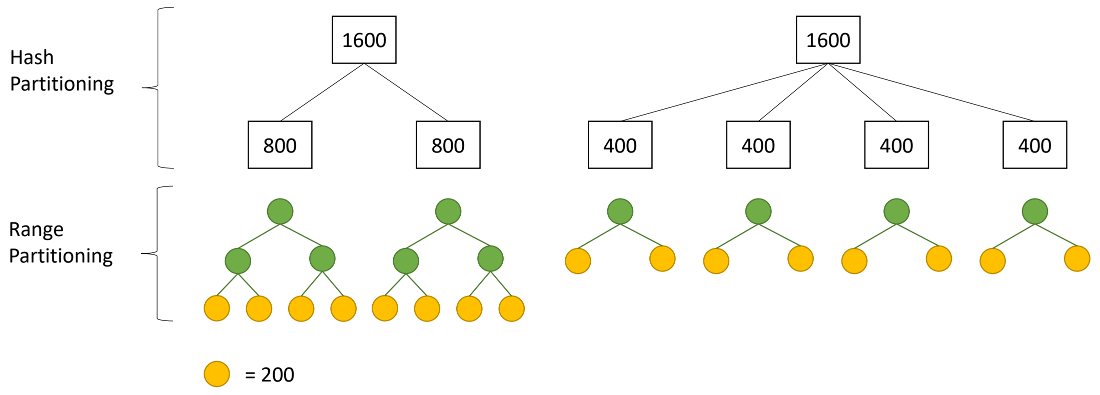


Figure 1: An example of the trade-off between hash partitioning and range partitioning

A fundamental constraint in partitioning a table is the minimum size of each partition due to I/O or compression efficiency reasons. This limit leads to a trade-off between hash partitioning and range partitioning. The more partitions that a table is first hash partitioned into, the qd-tree of each intermediate partitions has less room to grow before the final partitions become too small, thus impacting the effectiveness of data skipping; on the other hand, using less partitions for the hash partitioning phase can improve data skipping but reduce the parallelism when executing a join.

As an example, Figure 1 shows two partitioning strategies for the same table containing 1600 tuples. The left strategy first hash-partitions the table into two partitions and the subsequent qd-trees can grow to 2 levels before each final partition reaches the limit of 200 tuples. In contrast, the right strategy has 4 partitions for the hash partitioning, but can only construct 1-level qd-trees. Consequently, our framework need to find the optimal balance between partition size, hash partitioning, and range partitioning.

3 Proposed Solution

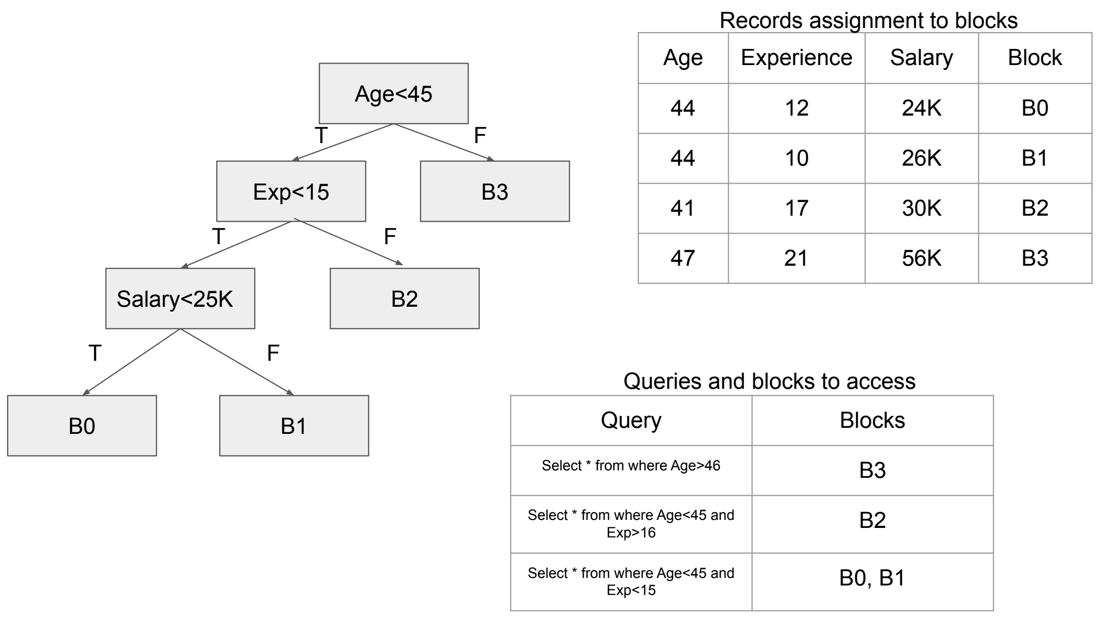
Our proposed framework consists of two steps. The first step aims to develop a cost model to range partition each table conditioned on varying levels of hash partitioning. The second step involves training an agent that utilizes the cost model to determine the optimal partitioning strategy for the entire database. We describe these two steps in this section.

3.0.1 First Step

We provide an overview of the qd-tree and its learning algorithm. Following this, we describe our approach for this step.

Qd-tree. A qd-tree is a binary decision tree where each inner node is labeled with a filter predicate (i.e. a cut); the leaf nodes represent the different data blocks that the tuples are eventually routed to through the qd-tree. The two children of an inner node correspond to the true and false answers when a tuple is evaluated with the predicate of that inner node. We route every tuple of a table through the qd-tree until it reaches the data block that it will be stored. For example, in the qd-tree in Figure 2, tuples are routed to different blocks based on the filter predicates:

- Tuples satisfying $Age < 45$, $Experience < 15$, and $Salary < 25K$ are routed to B0.
- Tuples satisfying $Age < 45$, $Experience < 15$, and not satisfying $Salary < 25K$ are routed to B1.
- Tuples satisfying $Age < 45$, and not satisfying $Experience < 15$ are routed to B2.
- Tuples satisfying $Age < 45$ are routed to B3.



Records assignment to blocks

Age	Experience	Salary	Block
44	12	24K	B0
44	10	26K	B1
41	17	30K	B2
47	21	56K	B3

Queries and blocks to access

Query	Blocks
Select * from where Age>46	B3
Select * from where Age<45 and Exp>16	B2
Select * from where Age<45 and Exp<15	B0, B1

Figure 2: An example qd-tree

During query execution, the qd-tree guides the query to the blocks that are relevant to the query. For example, consider the following query: `SELECT * FROM table WHERE` $Age < 45$ `AND` $Experience < 15$. Only blocks B0 and B1 in Figure 2 intersect with the predicate in this query, hence we can skip block B2 and B3.

To construct the RL agent called Woodblock, the author defines the tree construction process as a Markov Decision Process (MDP). The **state** of the agent is defined as an uncut node on the tree. For example, initially, the root node does not have any predicate assigned to it, making it the first state to be visited by the agent. The set of allowed cuts defines the **actions** that can be taken by the agent. Taking an action on a state results in the assignment of the cut on that state and the creation of two new nodes. These new nodes are added to a queue for later visits. An **episode** begins with only a root node and ends when a stopping condition is met. At the end of each episode, the reward is calculated, which is defined as the number of tuples skipped across the workload. Woodblock uses PPO, which involves training both a policy network and a value network. During training, the action is sampled using these networks, and gradient updates occur only when the reward is computed at the end of an episode.

Our approach. The aim of this step is to train the policy and value networks that enables the agent to derive an optimal qd-tree under varying levels of hash partitioning. We note that we do not immediately apply the learned policy to construct the qd-tree after this step but pass the resulting neural networks to the next step.

To achieve this, we modify the qd-tree by enabling it to start with its root being split once through hash partitioning. The number of children of the root corresponds to the number of partitions the table is hash partitioned into. Each of these children's sub-tree is the qd-tree for the hash partition of that child, referred to as a sub-qd-tree. To accommodate this change in the MDP, we augment the state representation with two values (d, h) where d represents the number of hash partitions and h denotes the current index of the bucket after taking the modulo of the hash value by d. Figure 3 illustrates a modified qd-tree with $d = 3$. There are 3 sub-qd-trees constructed for the 3 modulo buckets numbered from 0 to 2.

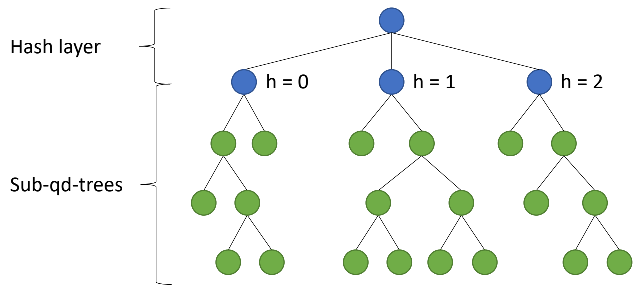


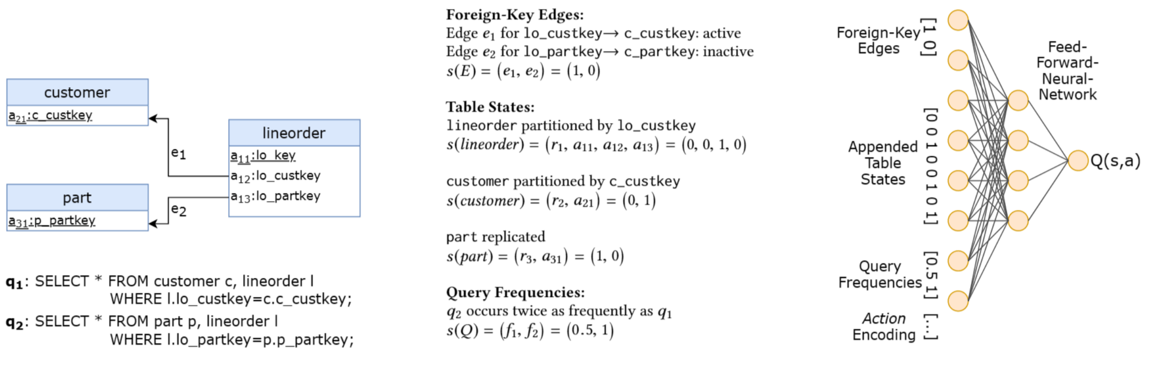
Figure 3: An example of a modified qd-tree with $d = 3$

During training, we vary d from 0 to a configurable maximum value. We first train the networks to construct a qd-tree when there is no hash partition ($d = 0$). Subsequently, for $d > 0$ we run the qd-tree training algorithm on each sub-qd-tree one after the other. All of the sub-qd-trees, including the qd-tree of $d = 0$, share the same policy and value networks. We hypothesize that this weight sharing strategy can accelerate learning since all sub-qd-trees are constructed on a random subset of the data from the same table.

The value network obtained from this step serves as the cost model for the next step. It provides the estimated amount of skipped tuples for a given (d, h) value. The policy network is used at the end of the second step to perform the actual partitioning.

3.0.2 Second Step

We base our solution for the second step on the framework proposed by Hilprecht et al.[14]. A table T_i can either be replicated or hash partitioned by one of its attributes $a_{i1}, a_{i2}, ..., a_{in}$. To represent the state of a table, they use a one-hot binary vector $s(T_i) = (r_i, a_{i1}, a_{i2}, ..., a_{in})$, where r_i indicates whether the table is replicated, and the rest of the bits denote whether the corresponding attribute is used for partitioning. For example, if the `customer` table from Figure 4 is replicated, its state vector is $(1, 0)$. They use edges between join attributes to indicate co-partitioning, and each edge can be activated or de-activated. In Figure 4, the active edge e_1 implies that the `customer` and `lineorder` tables are co-partitioned ($s(E) = (e_1, e_2) = (1, 0)$). They represent a workload using a frequency vector $s(Q) = (f_1, ..., f_m)$, where f_i is the relative frequency of the query Q_i. The overall state, obtained by concatenating these state vectors, is then input to a Q-network, which is trained using the DQN algorithm. The reward function is computed based on a cost model in offline mode and based on real execution in online mode.



```
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;
```

Foreign-Key Edges:
Edge e_1 for `lo_custkey`→ `c_custkey`: active
Edge e_2 for `lo_partkey`→ `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
$s(Q) = (f_1, f_2) = (0.5, 1)$

(a) Database and Workload **(b) State Representation** **(c) Q-Network with Encoded State**

Figure 4: State representation from Hilprecht et al. [13]

To support range partitioning, we propose adding an extra bit to each table vector to indicate whether the table is range partitioned, in addition to any hash partitioning that may already be in place. Furthermore, we integrate the cost model obtained from the first step into the cost model used for offline learning. This process should be straightforward, given that the cost model of the qd-tree is interpretable, as it outputs the number of tuples that can be skipped.

4 Implementation

We implemented the reinforcement learning agent as described in the qd-tree paper using Rllib [15] and PyTorch. The agent's architecture comprises separate policy and value networks, each consisting of two fully-connected layers with 512 units and ReLU activation.

We did not support categorical values and advanced cuts in our implementation. Each node in the qd-tree is represented by a list of ranges produced from the cuts leading to that node. For example, in Figure 2, the root node is represented by the ranges

$$(-\infty, \infty), (-\infty, \infty), (-\infty, \infty)$$

corresponding to the attributes *Age*, *Exp*, and *Salary* respectively.

Consequently, the cut $Age < 45$ at the root produces two child nodes:

$$(-\infty, 45), (-\infty, \infty), (-\infty, \infty)$$

$$[45, \infty), (-\infty, \infty), (-\infty, \infty)$$

.

We featurize each range into four values: `left-open`, `left-value`, `right-value`, `right-open`, indicating whether the left side is open or not, the left value, the right value, and whether the right side is open or not, respectively. All literal values from the cuts are extracted and dictionary encoded, hence `left-value` and `right-value` are integer indices from the dictionary. Prior to concatenating these four values into a vector, `left-value` and `right-value` are binarized into arrays of 16 binary numbers. Thus, each range is ultimately represented as a binary vector of length 34.

We encapsulate the qd-tree within a custom Gymnasium environment [16]. A state of the environment is the current node that the agent is currently located in, starting from the root node. Each node is featurized by concatenating its range vectors. Actions correspond to cuts that can be applied to the current node. If it is legal to apply a cut on the current node (i.e. each child node resulting from the cut contains more data than the mininum block size), the resulting child nodes are added to a queue, otherwise, nothing will happen. In both cases, a new node is then popped off the queue to be used as the next state. The reward for each action is not immediately returned but is instead computed after the environment reaches the terminal state, following the approach described in the qd-tree paper.

5 Results

In this section, we present the result of the experiment that explores the trade-off between hash partitioning and range partitioning.

We train our agent on the denormalized data from the TPC-H benchmark with scale factor 1. The denormalized table contains about 6 million tuples and we set the minimum block size to be 10,000 tuples. We select query templates in the benchmark that have range predicates (q1, q3, q4, q5, q6, q7, q8, q10, q12, q14, q15, q19, and q20). We generate 7 queries for each template, resulting in a total of 91 queries. The predicates in each query are extracted using regular expression and are broken down into 140 unique cuts. To speed up training and due to resource constraints, we use a sample that is 0.01 times the original data size. Evaluation on the skippability of the resulting qd-tree is performed on the original data.

To investigate the impact of hash partitions, we conduct an experiment with varying partition numbers. For each configuration, we train an agent for an hour to produce a qd-tree specifically on the first partition of the partitioned data. In Figure 5, we present the percentage of tuples skipped by the workload. Note that we only report the tuples skipped by the range predicates extracted from the queries. Given that the WHERE clauses of most queries consist of a conjuction of many more predicates, evaluating the entire query would likely yield more skipped tuples.

With the exception of the initial two data points that represent hash partitioning into 1 and 2 partitions, the qd-tree exhibits a reduction in skipped tuples as the number of hash partitions increases. This decline can be attributed to the decrease in tuple count within each partition as the

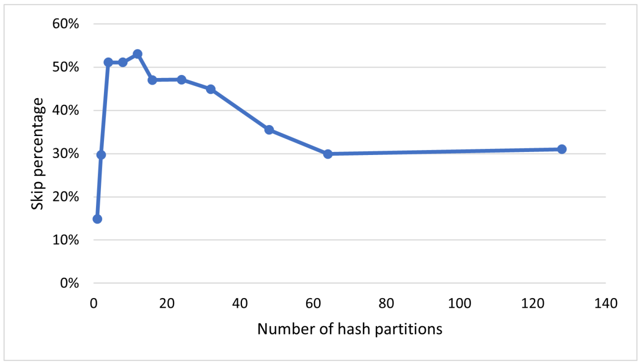
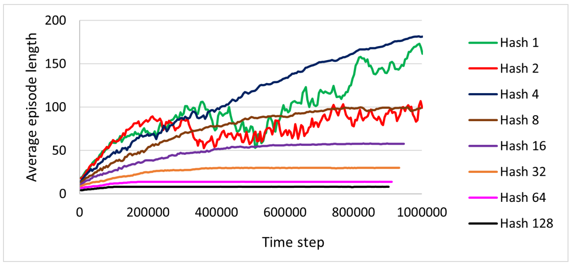


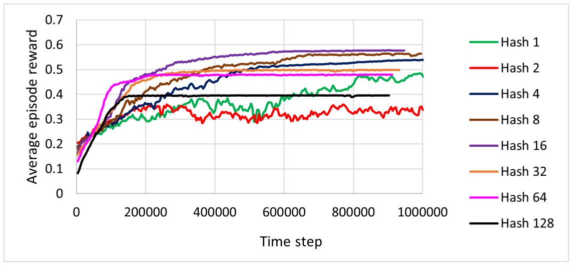
Figure 5: Percentage of skipped tuples vs. number of hash partitions

data is divided into more partitions. Consequently, the qd-tree has limited expansion potential for its leaves before reaching the minimum block size. However, the reduction rate is not substantial compared to the increase in the number of hash partitions. For instance, the skipping percentage shows minimal change when transitioning from 4 to 24 partitions. Significant degradation in the skipping percentage only becomes evident when scaling to more extreme numbers, such as 64 partitions. As a result, we conclude that the hash/range partition trade-off only becomes problematic in large-scale deployments of the data system.

In order to gain a better understanding of the previous findings, we visualize the training episode information in Figure 6. The episode length strongly correlates with the size of the qd-tree. Therefore, Figure 6a shows that increasing the number of hash partitions results in a decrease in average episode length. The reduction in tree size limits the possible qd-tree arrangements, consequently imposing a constraint on the episode reward for larger hash partition sizes, as depicted in Figure 6b.



(a) Average episode length



(b) Average episode reward

Figure 6: Metrics about the training episodes over time

Furthermore, we observe significant variance in both the average reward and episode length when the number of hash partitions is set to 1 or 2. This variance explains the inferior performance of the qd-tree in these scenarios as shown in Figure 5. Unless there is an error in our implementation, we hypothesize that this underperformance is attributed to the enlarged search space associated with these configurations. The increased variance complicates the learning process, making it more challenging to achieve high rewards consistently. This outcome suggests a potential optimization strategy for qd-tree training, which involves sampling the data to a smaller proportion without multiplying the minimum block size by the same factor. By doing so, we effectively limit the search space of possible qd-trees, leading to a reduction in variance during the training process. For instance, we find that the best-performing qd-tree in our experiment is generated in the configuration of 4 hash partitions, which is equivalent to sampling the data by a factor of 0.25.

6 Conclusion

In summary, we present a potential solution for the problem of integrating automated generation of hash-based and range-based partitioning. Our experiment successfully confirms the existence of a trade-off between hash and range partitioning. However, we observe that this trade-off only becomes problematic in large data system deployments. Furthermore, we identify a possible optimization technique for qd-tree learning by sampling a reduced dataset while keeping the minimum block size.

References

[1] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. Adaptdb: Adaptive partitioning for distributed joins. *Proc. VLDB Endow.*, 10(5):589–600, jan 2017.

[2] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 359–370, New York, NY, USA, 2004. Association for Computing Machinery.

[3] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.*, 10(4):445–456, nov 2016.

[4] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, nov 2014.

[5] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Broneske, Gunter Saake, Maya S. Sekeran, Fabián 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*, aiDM'18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358514.

[6] Rimma Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 1137–1148, New York, NY, USA, 2011. Association for Computing Machinery.

[7] Tilmann Rabl and Hans-Arno Jacobsen. Query centric partitioning and allocation for partially replicated database systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 315–330, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974.

[8] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1–2):48–57, sep 2010.

[9] Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge Quiane, and Aaron J. Elmore. A robust partitioning scheme for ad-hoc query workloads. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 229–241, New York, NY, USA, 2017. Association for Computing Machinery.

[10] K. Ashwin Kumar, Abdul Quamar, Amol Deshpande, and Samir Khuller. Sword: Workload-aware data placement and replica selection for cloud data management systems. *The VLDB Journal*, 23(6):845–870, dec 2014.

[11] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware partitioning in parallel database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 17–30, New York, NY, USA, 2015. Association for Computing Machinery.

[12] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-tree: Learning data layouts for big data analytics. SIGMOD '20, page 193–208, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. Towards learning a partitioning advisor with deep reinforcement learning. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '19, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. Learning a partitioning advisor for cloud databases. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 143–157, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Ray Team. Rllib: Industry-grade reinforcement learning. URL `https://docs.ray.io/en/latest/rllib/index.html`. Accessed: May 17, 2023.

[16] Gymnasium farama. URL `https://gymnasium.farama.org/`. Accessed: May 17, 2023.