SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning

IMMANUEL TRUMMER, JUNXIONG WANG, ZIYUN WEI, DEEPAK MARAM, SAMUEL MOSELEY, SAEHAN JO, JOSEPH ANTONAKAKIS, and ANKUSH RAYABHARI, Cornell University

SkinnerDB uses reinforcement learning for reliable join ordering, exploiting an adaptive processing engine with specialized join algorithms and data structures. It maintains no data statistics and uses no cost or cardinality models. Also, it uses no training workloads nor does it try to link the current query to seemingly similar queries in the past. Instead, it uses reinforcement learning to learn optimal join orders from scratch during the execution of the current query. To that purpose, it divides the execution of a query into many small time slices. Different join orders are tried in different time slices. SkinnerDB merges result tuples generated according to different join orders until a complete query result is obtained. By measuring execution progress per time slice, it identifies promising join orders as execution proceeds.

Along with SkinnerDB, we introduce a new quality criterion for query execution strategies. We upper-bound expected execution cost regret, i.e., the expected amount of execution cost wasted due to sub-optimal join order choices. SkinnerDB features multiple execution strategies that are optimized for that criterion. Some of them can be executed on top of existing database systems. For maximal performance, we introduce a customized execution engine, facilitating fast join order switching via specialized multi-way join algorithms and tuple representations.

We experimentally compare SkinnerDB's performance against various baselines, including MonetDB, Postgres, and adaptive processing methods. We consider various benchmarks, including the join order benchmark, TPC-H, and JCC-H, as well as benchmark variants with user-defined functions. Overall, the overheads of reliable join ordering are negligible compared to the performance impact of the occasional, catastrophic join order choice.

CCS Concepts: • **Information systems** → **Query optimization**; **Query planning**; Join algorithms;

Additional Key Words and Phrases: Query optimization, reinforcement learning, adaptive processing

ACM Reference format:
Immanuel Trummer, Junxiong Wang, Ziyun Wei, Deepak Maram, Samuel Moseley, Saehan Jo, Joseph Antonakakis, and Ankush Rayabhari. 2021. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *ACM Trans. Database Syst.* 46, 3, Article 9 (September 2021), 45 pages.
https://doi.org/10.1145/3464389

Authors' address: I. Trummer, J. Wang, Z. Wei, D. Maram, S. Moseley, S. Jo, J. Antonakakis, and A. Rayabhari, Cornell University, Gates Hall, 107 Hoyd Rd, Ithaca, NY 14853, USA; emails: {itrummer, jw2544, zw555, sm2686, sjm352, sj683, jma353, ar2354}@cornell.edu.

0362-5915/2021/09-ART9
https://doi.org/10.1145/3464389

1 INTRODUCTION

"The consequences of an act affect the probability of its occurring again." — B.F. Skinner.

Estimating execution cost of plan candidates is perhaps the primary challenge in query optimization [36]. Query optimizers predict cost based on coarse-grained data statistics and under simplifying assumptions (e.g., independent predicates). If estimates are wrong, then query optimizers may pick plans whose execution cost is sub-optimal by orders of magnitude. We present SkinnerDB,[1] a novel database system designed from the ground up for reliable query optimization.

SkinnerDB maintains no data statistics and uses no simplifying cost and cardinality models. Instead, SkinnerDB learns (near-)optimal left-deep query plans *from scratch* and on the fly, i.e., *during* the execution of a given query. This distinguishes SkinnerDB from several other recent projects that apply learning in the context of query optimization [32, 38]: Instead of learning from past query executions to optimize the next query ("inter-query learning"), we learn from the current query execution to optimize the remaining execution of the current query ("intra-query learning"). Hence, SkinnerDB does not suffer from any kind of generalization error across queries (even seemingly small changes to a query can change the optimal join order significantly).

SkinnerDB partitions the execution of a query into many, very small time slices (e.g., tens of thousands of slices per second). Execution proceeds according to different join orders in different time slices. We focus on in-memory data processing where random data accesses, caused by join order switching, have moderate overheads. Result tuples produced in different time slices are merged until a complete result is obtained. After each time slice, execution progress is measured, which informs us on the quality of the current join order. At the beginning of each time slice, we choose the join order that currently seems most interesting. In that choice, we balance the need for exploitation (i.e., trying join orders that worked well in the past) and exploration (i.e., trying join orders about which little is known). We use the UCT algorithm [31] to optimally balance between those two goals.

Along with SkinnerDB, we introduce a new quality criterion for query evaluation methods. Traditional query optimization [47] guarantees optimal plans according to a cost model that is based on simplifying assumptions about query and data. This optimality criterion becomes less useful if there is a disconnect between estimated and actual cost. Instead, our goal is to at least *bound* the *expected* distance between actual execution cost and actual cost of an optimal plan (with a focus on join ordering decisions). Under some simplifying assumptions, this becomes possible in the context of an adaptive processing strategy, allowing us to update query plans based on runtime observations. Table 1 illustrates the differences between those two quality criteria.

Our criterion is motivated by formal regret bounds provided by many reinforcement learning methods. They bound the expected distance between reward obtained by an evaluated algorithm and optimal reward [31]. Here, the optimal reward is obtained by a fixed policy with optimal expected reward. In our scenario, this corresponds to a fixed join order choice that works best for all data in average. Hence, we calculate regret in the following by comparing our adaptive join order selection against an optimal static join order (i.e., the more traditional approach to query optimization and execution). Traditional query optimization guarantees optimal plans, provided that complete information (e.g., on predicate selectivity and predicate correlations) is *a priori* available. We assume that no *a priori* information is available at the beginning of query execution. Our scenario, therefore, matches the one considered in reinforcement learning. This motivates us to apply

[1]The source code of SkinnerDB is available online at https://github.com/cornelldbgroup/skinnerdb (the main version presented in Section 6) and https://github.com/itrummer/SkinnerPG (the variants presented in Section 5).

Table 1. Comparing Different Quality Criteria in Query Optimization and Their Context

Criterion	**Metric**	**Guarantee**	**Bound Type**	**Processing**
Traditional	Cost Model	Optimal	Hard Bound	Non-adaptive
Regret Bounds	Cost	Near-optimal	In Expectation	Adaptive

a similar quality criterion. The adaptive processing strategies used in SkinnerDB are optimized for that criterion. We analyze regret bounds for SkinnerDB in Section 7.

SkinnerDB comes in multiple variants. Skinner-G sits on top of a generic SQL processing engine. Using optimizer hints (or equivalent mechanisms), we force the underlying engine to execute specific join orders on data batches. We use timeouts to limit the impact of bad join orders (which can be significant, as intermediate results can be large even for small base table batches). Of course, the optimal timeout per batch is initially unknown. Hence, we iterate over different timeouts, carefully balancing execution time dedicated to different timeouts while learning optimal join orders. Skinner-H is similar to Skinner-G in that it uses an existing database management system as execution engine. However, instead of learning new plans from scratch, it partitions execution time between learned plans and plans proposed by the original optimizer.

Both Skinner-G and Skinner-H rely on a generic execution engine. However, existing systems are not optimized for switching between different join orders during execution with a very high frequency. Skinner-C exploits a customized execution engine that is tailored to the requirements of regret-bounded query evaluation. It features a multi-way join strategy that keeps intermediate results minimal, thereby allowing quick suspend and resume for a given join order. Further, it allows to share execution progress between different join orders and to measure progress per time slice at a very fine granularity (which is important to quickly obtain quality estimates for join orders).

SkinnerDB pays for reliable join ordering with overheads for learning and join order switching. In our experiments with various baselines and benchmarks, we study under which circumstances the benefits outweigh the drawbacks. When considering accumulated execution time on benchmarks where optimization is difficult, it turns out that SkinnerDB can beat even highly performance-optimized systems for analytical processing with a traditional optimizer. While per-tuple processing overheads are significantly lower for the latter, SkinnerDB minimizes the total number of tuples processed via better join orders.

We summarize our original scientific contributions:

- We introduce a new quality criterion for query evaluation strategies that compares expected and optimal execution cost.
- We propose several adaptive execution strategies based on reinforcement learning.
- We formally prove correctness and regret bounds for those execution strategies.
- We experimentally compare those strategies, implemented in SkinnerDB, against various baselines.

The remainder of this article is organized as follows: We discuss related work in Section 2. In particular, we summarize differences to a prior conference version of this article. We describe the primary components of SkinnerDB in Section 3. In Sections 4 to 6, we describe our query evaluation strategies based on reinforcement learning. In Section 7, we analyze those strategies formally, we prove correctness and performance properties. In Section 8, we compare our approaches experimentally against a diverse set of baselines. We describe the SkinnerDB implementation in more detail in the online appendix.

2 RELATED WORK

This article is an extended version of our prior conference publication [52]. The prior publication was extended in the following ways: First, it features various new experimental results that did not appear in the prior version. Most importantly, over the past six months, we have created a new and more mature version of SkinnerDB entirely from scratch. Our goal was to streamline code and to incorporate standard techniques such as runtime code generation and data compression. We believe that doing so was necessary to distinguish intrinsic overheads, due to the particularities of intra-query learning, from avoidable ones (thereby gaining a better understanding of the potential of the proposed approach). The resulting SkinnerDB version is approximately two times faster than the original, showing that intra-query learning is more powerful than our initial results implied. This article contains the first experimental evaluation of that new version. Besides that, we use new benchmarks (e.g., JCC-H with and without skew, IMDB benchmark variants on large data and with user-defined functions), compare against new baselines (e.g., TelegraphCQ and newer versions of MonetDB and PostgreSQL), and present results for new types of experiments (e.g., a detailed breakdown of execution time into different phases executed by SkinnerDB, test of robustness towards semantically equivalent query rewrites). Also, we compare different reinforcement learning algorithms for intra-query learning against the original UCT algorithm. Besides new experiments, we have also extended the proposed approach. In particular, we describe new reward functions (see Sections 6.5 and 6.6) and "fast backtracking," an extension to our join algorithm (see Section 6.9). Both extensions are crucial to achieve optimal performance (see Section 8.3).

Our approach connects to prior work collecting information on predicate selectivity by evaluating them on data samples [9, 10, 26, 27, 29, 35, 39, 57]. We compare in our experiments against a recently proposed representative [57]. Most prior approaches rely on a traditional optimizer to select interesting intermediate results to sample. They suffer if the original optimizer generates bad plans. The same applies to approaches for interleaved query execution and optimization [1, 5, 7] that repair initial plans at runtime if cardinality estimates turn out to be wrong. Robust query optimization [3, 4, 6, 14] assumes that predicate selectivity is known within narrow intervals, which is often not the case [21]. Prior work [19, 20] on query optimization without selectivity estimation is based on simplifying assumptions (e.g., independent predicates) that are often violated.

There has been significant interest in applying machine learning techniques for query optimization and database tuning. In the context of query optimization, machine learning has been applied at multiple levels. Some prior work has focused on leveraging machine learning to estimate processing cost of query plans or workflows [2, 18, 23, 34, 44, 50]. Often, incorrect cardinality estimates are the root reason for sub-optimal plan choices. Hence, a popular line of work focuses on learning to predict predicate selectivity or cardinality of intermediate results via machine learning [1, 2, 28, 30, 34, 42, 43, 49, 56]. The latter work can be applied within traditional query optimization methods. Other recent work uses machine learning directly to make planning choices [32, 37, 38, 58]. All of the aforementioned approaches learn from past queries for the optimization of future queries. To be effective, new queries must be similar to prior queries and this similarity must be recognizable. Instead, we learn *during* the execution of a query.

Other recent work [33, 59] connects to SkinnerDB by the use of reinforcement learning for database tuning. Here, the focus is however on setting system configuration parameters, rather than on join ordering decisions.

Adaptive processing strategies have been explored in prior work [5, 15, 16, 45, 46, 53, 55]. Our work uses reinforcement learning and is therefore most related to prior work using reinforcement learning in the context of Eddies [53]. We compare against this approach in our experiments. Eddies do not provide formal guarantees on the relationship between expected execution time and

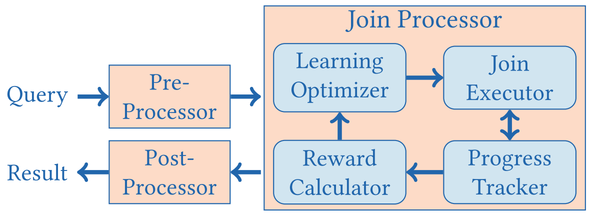


Fig. 1. Primary components of SkinnerDB.

the optimum. They never discard intermediate results, even if joining them with the remaining tables creates disproportional overheads. Eddies support bushy query plans in contrast to our approach. Bushy plans can in principle decrease execution cost compared to the best left-deep plan. However, optimal left-deep plans typically achieve reasonable performance [25]. Also, as we show in our experiments, reliably identifying near-optimal left-deep plans can be better than selecting bushy query plans via non-robust optimization.

Our work relates to prior work on filter ordering with regret bounds [12]. Join ordering introduces however new challenges, compared to filter ordering. In particular, applying more filters can only decrease the size of intermediate results. The relative overhead of a bad filter order, compared to the optimum, therefore grows linearly in the number of filters. The overhead of bad join orders, compared to the optimum, can grow exponentially in the query size. This motivates mechanisms that bound join overheads for single data batches, as well as mechanisms to save progress for partially processed data batches.

For Skinner-C, we introduce a multi-way join algorithms for optimal performance in our scenario. In that, our work connects to prior work on worst-case optimal join processing (which also relies on specialized multi-way join algorithms). Worst-case optimal join algorithms [41, 54] bound cost as a function of worst-case query result size. We bound expected execution cost as a function of cost for processing an optimal join order. Further, prior work on worst-case optimal joins focuses on conjunctive queries, while we support a broader class of queries, including queries with user-defined function predicates. Our approach applies to SQL with standard semantics, while systems for worst-case optimal evaluation typically assume set semantics [54].

3 SYSTEM OVERVIEW

Figure 1 shows the primary components of SkinnerDB. This high-level outline applies to all of the SkinnerDB variants.

The pre-processor is invoked first for each query. Here, we filter base tables via unary predicates. Also, depending on the SkinnerDB variant, we partition the remaining tuples into batches or hash them (to support joins with equality predicates).

Join execution proceeds in small time slices. The join processor consists of several subcomponents. The learning optimizer selects a join order to try next at the beginning of each time slice. It uses statistics on the quality of join orders that were collected during the current query execution. Selected join orders are forwarded to the join executor. This component executes the join order until a small timeout is reached. We add result tuples into a result set, checking for duplicate results generated by different join orders. The join executor can be either a generic SQL processor or, for maximal performance, a specialized execution engine. The same join order may get selected repeatedly. The progress tracker keeps track of which input data has been processed already. For Skinner-C, it even tracks execution state for each join order tried so far and merges progress across join orders. At the start of each time slice, we consult the progress tracker to restore the latest state stored for the current join order. At the end of it, we backup progress achieved during the current time slice. The reward calculator calculates a reward value based on progress achieved during the current time slice. This reward is a measure for how quickly execution proceeds using the chosen join order. It is used as input by the optimizer to determine the most interesting join order to try in the next time slice.

Finally, we invoke the post-processor, using the join result tuples as input. Post-processing involves grouping, aggregation, and sorting. In the next section, we describe the algorithms executed within SkinnerDB.

4 JOIN ORDER LEARNING

In Section 4.1, we describe a popular reinforcement learning algorithm. In Section 4.2, we show how this algorithm can be applied for intra-query join order learning.

4.1 Background on UCT

Our method for learning optimal join orders is based on the UCT algorithm [31]. This is an algorithm from the area of reinforcement learning. It assumes the following scenario: We repeatedly make choices that result in rewards. Each choice is associated with reward probabilities that we can learn over time. Our goal is to maximize the sum of obtained rewards. To achieve that goal, it can be beneficial to make choices that resulted in large rewards in the past ("exploitation") or choices about which we have little information ("exploration") to inform future choices. The UCT algorithm balances between exploration and exploitation in a principled manner that results in probabilistic guarantees. More precisely, assuming that rewards are drawn from the interval $[0, 1]$, the UCT algorithm guarantees that the expected regret (i.e., the difference between the sum of obtained rewards to the sum of rewards for optimal choices) is in $O(\log(n))$ where n designates the number of choices made [31].

We specifically select the UCT algorithm for several reasons. First, UCT has been applied successfully to problems with very large search spaces (e.g., planning Go moves [24]). This is important, since the search space for join ordering grows quickly in the query size. Second, UCT provides formal guarantees on cumulative regret (i.e., accumulated regret over all choices made). Other algorithms from the area of reinforcement learning [22] focus for instance on minimizing simple regret (i.e., quality of the final choice). The latter would be more appropriate when separating planning from execution. Our goal is to interleave planning and execution, making the first metric more appropriate. Third, the formal guarantees of UCT do not depend on any instance-specific parameter settings [17], distinguishing it from other reinforcement learning algorithms.

We assume that the space of choices can be represented as a search tree. In each round, the UCT algorithm makes a series of decisions that can be represented as a path from the tree root to a leaf. Those decisions result in a reward from the interval $[0, 1]$, calculated by an arbitrary, randomized function specific to the leaf node (or as a sum of rewards associated with each path step). Typically, the UCT algorithm is applied in scenarios where materializing the entire tree (in memory) is prohibitively expensive. Instead, the UCT algorithm expands a partial search tree gradually towards promising parts of the search space. The UCT variant used in our system expands the materialized search tree by at most one node per round (adding the first node on the current path that is outside the currently materialized tree).

Materializing search tree nodes allows to associate statistics with each node. The UCT algorithm maintains two counters per node: the number of times the node was visited and the average reward that was obtained for paths crossing through that node. If counters are available for all relevant nodes, then the UCT algorithm selects at each step the child node c maximizing the formula $r_c + w \cdot \sqrt{\log(v_p)/v_c}$, where r_c is the average reward for c, v_c and v_p are the number of visits for

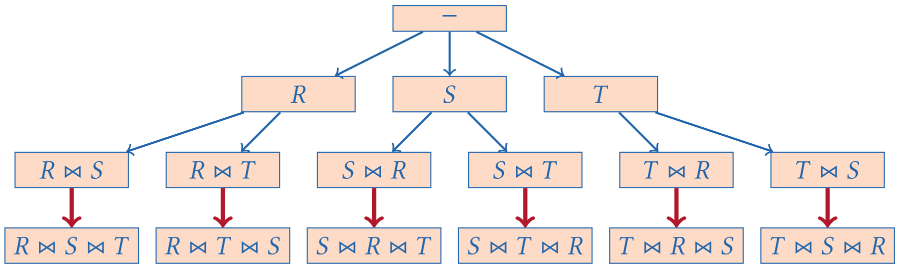


Fig. 2. We model join order choices for a query as an episodic Markov Decision Process. Actions (arrows) correspond to table choices and states (rectangles) to join order prefixes. A selected join order is executed on one data batch (Skinner-G and Skinner-H) or for one time slice (Skinner-C). Reward is calculated based on execution progress and associated with the final state (transition marked with thick red line).

child and parent node, and w a weight factor. In this formula, the first term represents exploitation while the second term represents exploration. Their sum represents the upper bound of a confidence bound on the reward achievable by passing through the corresponding node (hence the name of the algorithm: UCT for Upper Confidence bounds applied to Trees). Setting $w = \sqrt{2}$ is sufficient to obtain bounds on expected regret. It can however be beneficial to try different values to optimize performance for specific domains [17].

4.2 Learning Optimal Join Orders

Our search space is the space of join orders. Optionally, we can exclude join orders that introduce Cartesian product joins without need [25] (however, we do not use this heuristic for our experiments). To apply the UCT algorithm for join ordering, we need to represent the search space as a tree. We assume that each tree node represents one decision with regards to the next table in the join order. Tree edges represent the choice of one specific table. The tree root represents the choice of the first table in the join order. All query tables can be chosen, since no table has been selected previously. Hence, the root node will have m child nodes where m is the number of tables to join. Nodes in the next layer of the tree (directly below the root) represent the choice of a second table. We cannot select the same table twice in the same join order. Hence, each of the latter nodes will have at most $m - 1$ child nodes associated with remaining choices. The number of choices depends on the structure of the join graph. If at least one of the remaining tables is connected to the first table via join predicates, then only such tables will be considered. If none of the remaining tables is connected, then all remaining tables become eligible (since a Cartesian product join cannot be avoided given the initial choice). In total, the search tree will have m levels. Each leaf node is associated with a completely specified join order.

We generally divide the execution of a query into small time slices in which different join orders are tried. For each time slice, the UCT algorithm selects a path through the aforementioned tree, thereby selecting the join order to try next. As discussed previously, only part of the tree will be "materialized" (i.e., we keep nodes with node-specific counters in main memory). When selecting a path (i.e., a join order), UCT exploits counters in materialized nodes wherever available to select the next path step. Otherwise, the next step is selected randomly. After a join order has been selected, this join order is executed during the current time slice. Results from different time slices are merged (while removing overlapping results). We stop once a complete query result is obtained.

Our goal is to translate the aforementioned formal guarantees of UCT, bounding the distance between expected and optimal reward (i.e., the regret), into guarantees on query evaluation speed. To achieve that goal, we must link the reward function to query evaluation progress. The approaches for combined join order learning and execution, presented in the following subsections, define the reward function in different ways. They all have, however, the property that higher rewards correlate with better join orders. After executing the selected join order for a bounded amount of time, we measure evaluation progress and calculate a corresponding reward value. The UCT algorithm updates counters (average reward and number of visits) in all materialized tree nodes on the previously selected path.

Conceptually, we model join order optimization as an episodic **Markov Decision Process (MDP)**. We describe an MDP by a tuple $\langle\mathcal{S},\mathcal{A},\mathcal{T},\mathcal{R}\rangle$, where $\mathcal{S}$ is a set of states and $\mathcal{A}$ a set of actions. $\mathcal{T}:\mathcal{S}\times\mathcal{A}\rightarrow\mathcal{S}$ is a deterministic transition function and $\mathcal{R}$ a stochastic reward function. Here, $\mathcal{R}(s)$ is the probability distribution over rewards obtained when arriving in state s (via an arbitrary action). Multiple, slightly different formalisms are used to describe MDPs throughout the literature. Here, we restrict the one used for instance by Silver [48] to deterministic transitions and stochastic rewards that depend only on the target state after a transition.

For a query joining m relations R_1 to R_m, states correspond to permutations of table subsets, i.e., $\mathcal{S}=\{Perm(R)|R\subseteq\{R_1,\ldots,R_m\}\}$ where $Perm(R)$ denotes the set of permutations of elements from R. In this definition, we assume that all Cartesian product joins are considered. Actions correspond to query tables, i.e., $\mathcal{A}=\{R_1,\ldots,R_m\}$. The transition function appends the table, represented by action a, to a partial join order j without a: $\mathcal{T}(j,a)=j\circ a$. Each MDP episode starts with an empty join order and ends with a fully specified join order (after a number of transitions that is equal to the number of joined tables). The reward function assigns a constant reward of zero to all states except for final states (i.e., $\mathcal{R}(j)=0$ if $|j|<m$ where m is the number of tables and $||$ the length of a join order). For a transition leading to an end state (i.e., a fully specified join order), we calculate reward based on how well the join order fares when used for execution during the current episode. The precise reward function for Skinner-G and Skinner-H is discussed in Section 5.3. For Skinner-C, we propose two alternatives in Sections 6.5 and 6.6. The reward of a join order is modeled as stochastic in our model, as selecting (and executing) the same join order in different episodes can yield different rewards (e.g., due to inhomogeneous data). The optimal (fixed) join order is the one with highest expected reward. Our goal is to converge to that join order. The UCT algorithm associates MDP states with tree nodes and converges to leaf nodes with highest expected reward [31]. Figure 2 illustrates the MDP associated with a query joining three tables.

The following algorithms use the UCT algorithm as a sub-function. More precisely, we use two UCT-related commands in the following pseudo-code: UctChoice(T) and RewardUpdate(T,j,r). The first one returns the join order chosen by the UCT algorithm when applied to search tree T (some of the following processing strategies maintain multiple UCT search trees for the same query). The second function updates tree T by registering reward r for join order j. Sometimes, we will pass a reward function instead of a constant for r (with the semantics that the reward resulting from an evaluation of that function is registered).

5 INTRA-QUERY LEARNING WITH GENERIC EXECUTION ENGINES

We show how we can learn optimal join orders when treating the execution engine as a black box with an SQL interface. This approach can be used on top of existing DBMS without changing a single line of their code. While we describe the following algorithms only for SPJ queries, it is straight-forward to add sorting, grouping, or aggregate calculations in a post-processing step (we do so in our actual implementation). Nested queries can be treated via decomposition [40].

5.1 Basic Approach

Our goal is to apply reinforcement learning for join order optimization within the execution of a single query. Hence, we must divide query execution into small episodes, allowing us to try different join orders in different episodes. Also, we want to obtain an unbiased estimate of join order quality after each episode (thereby informing join order choices for the remaining episodes). Next, we outline a basic query evaluation strategy that satisfies those desiderate under simplifying assumptions (see our online appendix for implementation details).

To simplify the following explanations, we assume that cost follows approximately the C_{out} cost metric [11]. This metric estimates execution cost of a join order by summing up intermediate join result sizes. Despite its simplicity, this metric has been shown, via theoretical [11] analysis, to correlate well with more sophisticated cost models. Also, empirical analysis shows that inaccuracies introduced by such simple cost models are typically dwarfed by the impact of inaccurate cardinality estimates [25].

Now, for a fixed join order, assume that we reduce the size of the *left-most table*. In expectation, the sizes of all intermediate results reduce proportionally. Hence, expected cost, according to the C_{out} cost metric, reduces proportionally as well. This insight motivates the following strategy: In each episode, we select a join order via reinforcement learning. When executing this order, we replace the left-most table by a batch of tuples from that table. Besides the left-most table, we use the original tables for all other positions in the join order. Under the C_{out} cost metric, we can scale up execution cost for one batch (of the left-most table) to obtain an unbiased estimate for execution cost on the full table. For instance, if a table batch contains one-tenth of the full table's tuples, then we multiply per-batch execution cost by factor 10 to obtain the estimate. We can exploit such estimates for a reward signal, steering the learning algorithm towards good join orders.

Executing a join order on parts of the input data produces parts of the result. More precisely, we produce all result tuples referencing tuples in the current batch from the left-most table. We collect partial results from different episodes and combine them to form a complete join result. If all batches of a table have been processed (during episodes where that table appeared as left-most table), then we have collected a complete join result. In that case, query evaluation stops and the union of partial results is returned to the user. Note that we can discard batches for which result tuples have been collected, thereby avoiding redundant work.

This basic approach comes with a problem. The cost of good and bad join orders often differs by multiple orders of magnitude [25]. Hence, the cost of executing a bad join order for one single episode may easily dominate total execution costs. Therefore, to protect ourselves from bad join orders, we must limit execution time per episode via timeouts.

5.2 Choosing Per-batch Timeouts

This leads however to a new problem: What timeout should we choose per batch in each iteration? Ideally, we would select as timeout the time required by an optimal join order. Of course, we neither know an optimal join order nor its optimal processing time for a new query. Using a timeout that is lower than the optimum prevents us from processing an entire batch before the timeout. On the other side, choosing a timeout that is too high leads to unnecessary overheads when processing sub-optimal join orders.

The choice of a good timeout is therefore crucial while we cannot know the best timeout *a priori*. The solution lies in an iterative scheme that tries different timeouts in different iterations. We carefully balance allocated execution time over different timeouts, avoiding to use higher timeouts unless lower ones have been tried sufficiently often. More precisely, we will present a timeout scheme that ensures that the total execution time allocated per timeout does not differ by more than factor two across different timeouts. Figure 3 gives an intuition for the corresponding timeout scheme (numbers indicate the iteration in which the corresponding timeout is chosen). We use timeouts that are powers of two (we also call the exponent the *Level* of the timeout). We always choose the highest timeout for the next iteration such that the accumulated execution time for

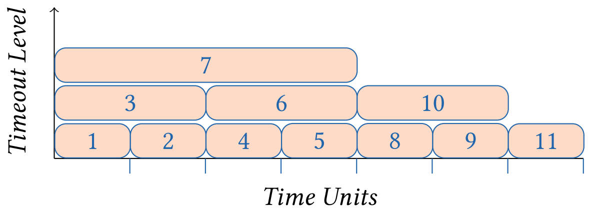


Fig. 3. Illustration of time budget allocation scheme: We do not know the optimal time per batch and iterate over different timeouts, allocating higher budgets less frequently.

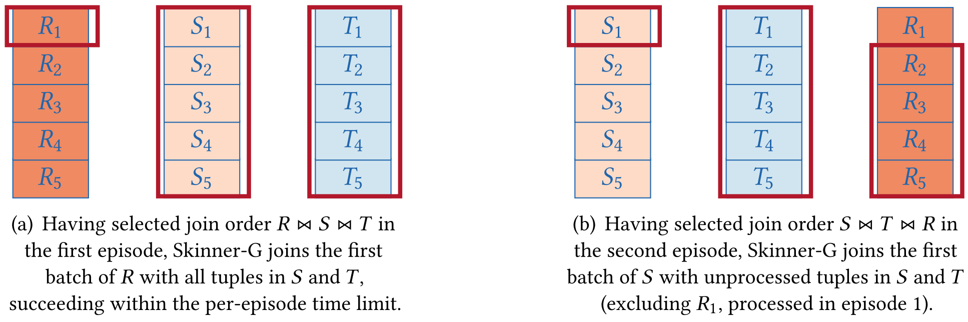

R_1 R_2 R_3 R_4 R_5
S_1 S_2 S_3 S_4 S_5
T_1 T_2 T_3 T_4 T_5
(a) Having selected join order $R \bowtie S \bowtie T$ in the first episode, Skinner-G joins the first batch of R with all tuples in S and T, succeeding within the per-episode time limit.
S_1 S_2 S_3 S_4 S_5
T_1 T_2 T_3 T_4 T_5
R_1 R_2 R_3 R_4 R_5
(b) Having selected join order $S \bowtie T \bowtie R$ in the second episode, Skinner-G joins the first batch of S with unprocessed tuples in S and T (excluding R_1, processed in episode 1).

Fig. 4. Illustrating first (a) and second (b) episode of Skinner-G for an example query with three tables (R, S, and T). The red rectangles, surrounding the tables, highlight tuples joined in the corresponding episode.

that timeout does not exceed time allocated to any lower timeout. Having fixed a timeout for each iteration, we assign a reward of one for a fixed join order if the input was processed entirely. We assign a reward of zero otherwise. Alternatively, we could use a reward metric that considers execution time for successfully processed batches (assigning higher rewards for lower time). We opt for the simpler metric, as time is already indirectly integrated via the timeout.

5.3 Pure Reinforcement Learning Algorithm

Algorithm 1 presents pseudo-code matching the verbal description. First, tuples are filtered using unary predicates and the remaining tuples are partitioned into b batches per table (we omit pseudo-code for pre-processing). We use function DBMS to invoke the underlying DBMS for processing one batch with a timeout. The function accumulates partial results in a result relation if processing finishes before the timeout and returns **true** in that case. Vector o_i stores for each table an offset, indicating how many of its batches were completely processed (it is implicitly initialized to one for each table). As we process batches in a fixed order, o only stores one scalar offset for each table. Variable n_l stores for each timeout level l how much execution time was dedicated to it so far (it is implicitly initialized to zero and updated in each invocation of function NextTimeout).

Example 5.1. Figure 4 illustrates processing of an example query with three tables for the first two episodes. In the first episode, the UCT algorithm selects join order $R \bowtie S \bowtie T$. Hence, Algorithm 1 tries to join the first batch of table R with all tuples in the other two tables. We assume that it succeeds within the time limit (collecting a reward of one and making a selection of this join order in future episodes more likely). The join result tuples are added to the result set. In the second episode, Skinner-G selects join order $S \bowtie T \bowtie R$. Hence, Algorithm 1 joins the first batch of S with the remaining, *unprocessed* tuples from the other tables (which excludes the first batch of R, R_1, which was already processed before).

ALGORITHM 1: Regret-bounded query evaluation using a generic execution engine

```
// Returns timeout for processing next batch,
// based on times $n$ given to each timeout before.
function NextTimeout($n$)
    // Choose timeout level
    $L \leftarrow \max\{L | \forall l < L : n_l \geq n_L + 2^L\}$
    // Update total time given to level
    $n_L \leftarrow n_L + 2^L$
    // Return timeout for chosen level
    return $2^L$
end function
// Process SPJ query $q$ using existing DBMS and
// by dividing each table into $b$ batches.
procedure SkinnerG($q = R_1 \bowtie \cdots \bowtie R_m, b$)
    // Apply unary predicates and partitioning
    $\{R_1^1, \ldots, R_m^b\}$ ←PreprocessingG($q, b$)
    // Until we processed all batches of one table
    while $\nexists i : o_i > b$ do
        // Select timeout using pyramid scheme
        $t$ ←NextTimeout(n)
        // Select join order via UCT algorithm
        $j$ ←UctChoice($T_t$)
        // Process one batch until timeout
        $suc$ ←DBMS($R_{j1}^{o_{j1}} \bowtie R_{j2}^{o_{j2}..b} \cdots \bowtie R_{jm}^{o_{jm}..b}, t$)
        // Was entire batch processed successfully?
        if $suc$ then
            // Mark current batch as processed
            $o_{j1} \leftarrow o_{j1} + 1$
            // Store maximal reward in search tree
            RewardUpdate($T_t, j, 1$)
        else
            // Store minimal reward in search tree
            RewardUpdate($T_t, j, 0$)
        end if
    end while
end procedure
```

5.4 Algorithmic Variants

Algorithm 1 discards processed tuple batches in any table. Alternatively, we can use unprocessed batches in the left-most table in join order, while using original tables for all other join order positions. This may lead to redundant work (for re-generating join result tuples) but reduces overheads for discarding processed tuples. Additional bookkeeping is required to avoid passing on redundant result tuples to the next phase; e.g., we can separate result tuples based on the left-most table of the join order used for generating them. Once all batches have been processed for one specific table, we pass on the associated result tuple collection (and discard others).

Algorithm 1 maintains different UCT trees T_t for each timeout t (implicitly initialized as a single root node representing no joined tables). This allows us to learn optimal join orders for each

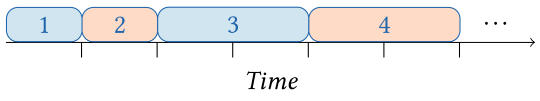


Fig. 5. The hybrid approach alternates with increasing timeouts between executing plans proposed by the traditional optimizer (red) and learned plans (blue).

specific timeout separately. Alternatively, we can use one single UCT tree for all timeouts. Even if we record rewards for different timeouts in the same tree, we can still converge to an optimal join order. Smaller timeouts are tried more frequently. Hence, join orders using less processing time per batch receive higher average rewards.

As a variant of Algorithm 1, we can alternate between batched and non-batched executions. Executing join orders on small data batches yields information on their quality. We can also execute those join orders on larger batches or on the original data. Executing join orders on larger data batches is often more efficient, as it amortizes startup overheads. Of course, we cannot know at which point the UCT algorithm has converged to a near-optimal join order. Hence, we alternate between batched execution (good for quickly collecting information on join order quality) and non-batched execution (good for exploiting the most promising join orders). For non-batched execution, we use the join order that was used most frequently for processing batches. If time allocated for non-batched execution remains proportional to time spent in batched execution, then execution overheads can only increase by a constant factor. The tradeoffs between batched and non-batched execution are the following: By increasing time dedicated to non-batched execution, we reduce overheads due to plan switching. On the other side, we reduce the number of iterations for reinforcement learning steps (one batch is tried in each iteration). This means that we may spend time executing highly sub-optimal plans via non-batched execution. Which tradeoff is optimal depends on properties of the underlying execution engine as well as query properties. For queries that are more difficult to optimize, allowing more iterations for learning can be beneficial. On the other side, if the overheads of starting new query plans are higher, then reducing the number of iterations for batched execution may be preferable. For our experiments, we empirically choose the tradeoff that works best for our system and benchmark.

5.5 Hybrid Algorithm

If queries are easy to optimize via traditional query optimization, then the Skinner-G approach adds unnecessary overheads for learning and join order switching. We present a hybrid algorithm that combines reinforcement learning with a traditional query optimizer. Instead of using an existing DBMS only as an execution engine, we additionally try benefiting from its query optimizer whenever possible. We do not provide pseudo-code for the hybrid algorithm, as it is quick to explain. We iteratively execute the query using the plan chosen by the traditional query optimizer, using a timeout of 2^i where i is the number of invocations (for the same input query) and time is measured according to some atomic units (e.g., several tens of milliseconds). In between two traditional optimizer invocations, we execute the learning-based algorithm described in the last subsection. We execute it for the same amount of time as the traditional optimizer. We save the state of the UCT search trees between different invocations of the learning approach. Optionally, if a table batch was processed by the latter, then we can remove the corresponding tuples before invoking the traditional optimizer. Figure 5 illustrates the hybrid approach. As shown in Section 7, the hybrid approach bounds expected regret (compared to the optimal plan) and guarantees a constant factor overhead compared to the original optimizer.

6 INTRA-QUERY LEARNING WITH CUSTOMIZED EXECUTION ENGINE

We present our main variant: Skinner-C. Skinner-C uses intra-query learning and features an execution engine that is specialized, in terms of operators and data structures, to this scenario.

6.1 Customizing Execution Engines for Intra-query Learning

Most execution engines are designed for a traditional approach to query evaluation. They assume that a single join order is executed for a given query (after being generated by the optimizer). Learning optimal join orders while executing a query leads to unique requirements on the execution engine. First, we execute many different join orders for the same query, each one only for a short amount of time. Second, we may even execute the same join order multiple times with many interruptions (during which we try different join orders). This specific scenario leads to (at least) three desirable performance properties for the execution engine. First, the execution engine should minimize overheads when switching join orders. Second, the engine should preserve progress achieved for a given join order even if execution is interrupted. Finally, the engine should allow to share achieved progress, to the maximal extent possible, between different join orders as well. The generic approach realizes the latter point only to a limited extend (by discarding batches processed completely by any join order from consideration by other join orders).

The key towards achieving the first two desiderata (i.e., minimal overhead when switching join orders or interrupting execution) is a mechanism that backs up execution state as completely as possible. Also, restoring prior state when switching join order must be very efficient. By "state," we mean the sum of all intermediate results and changes to auxiliary data structures that were achieved during a partial query evaluation for one specific join order. We must keep execution state as small as possible to back it up and to restore it efficiently.

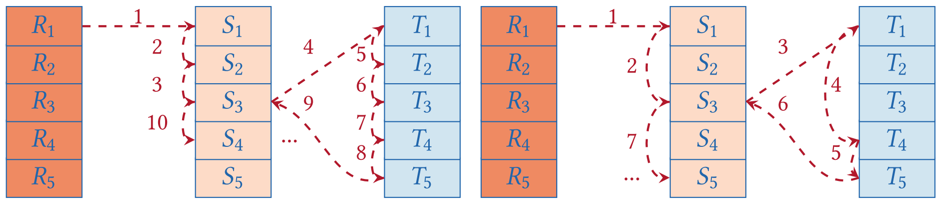
6.2 Representing Execution State

Two key ideas enable us to keep execution state small. First, we represent tuples in intermediate results concisely as vectors of tuple indices (each index pointing to one tuple in a base table). Second, we use a multi-way join strategy limiting the number of intermediate result tuples to at most one at any point in time (we may however store multiple final result tuples). Next, we discuss both ideas in detail.

Traditional execution engines for SPJ queries produce intermediate results that consist of actual tuples (potentially containing many columns with elevated byte sizes). To reduce the size of the execution state, we materialize tuples only on demand. Each tuple, be it a result tuple or a tuple in an intermediate result, is the result of a join between single tuples in a subset of base tables. Hence, whenever possible, we describe tuples simply by an array of tuple indices (whose length is bounded by the number of tables in the input query). We materialize partial tuples (i.e., only the required columns) temporarily to check whether they satisfy applicable predicates or immediately before returning results to the user. To do that efficiently, we assume a column store architecture (allowing quick access to selected columns) and a main-memory resident dataset (reducing the penalty of random data access).

6.3 Multi-way Join Algorithm

Most traditional execution engines for SPJ queries process join orders by a sequence of binary join operations. This can generate large intermediate results that would become part of the execution state. We avoid that by a multi-way join strategy whose intermediate result size is restricted to at most one tuple. We describe this strategy first for queries with generic predicates. Later, we discuss an extension for queries with equality join predicates based on hashing.



(a) Depth-first multi-way join strategy: we increase the join order index once the first tuple satisfying all applicable predicates is found, we decrease it once all tuples in the current table were considered.

(b) Depth-first multi-way join strategy exploiting indices on join columns: using indices, the algorithm skips tuples that do not satisfy binary equality join predicates.

Fig. 6. Comparison of simple (left) and refined (right) variant of depth-first multi-way join algorithm.

Intuitively, our multi-way join strategy can be understood as a depth-first search for result tuples. Considering input tables in one specific join order, we fix one tuple in a predecessor table before considering tuples in the successor table. We start with the first tuple in the first table (in join order). Next, we select the first tuple in the second table and verify whether all applicable predicates are satisfied. If that is the case, then we proceed to considering tuples in the third table. If not, then we consider the next tuple in the second table. Once all tuples in the second table have been considered for a fixed tuple in the first table, we "backtrack" and advance the tuple indices for the first table by one. Execution ends once all tuples in the first table have been considered.

Example 6.1. Figure 6(a) illustrates the process for a three-table join. Having fixed a tuple in the left-most table (at the left, we start with the first tuple), the join order index is increased. Next, we find the first tuple in the second table satisfying the join condition with the current tuple in the first table. Having found such a tuple, we increase the join order index again. Now, we iterate over tuples in the third table, adding each tuple combination satisfying all applicable conditions to the result. After all tuples in the last table have been considered, we decrease the join order index and consider the next tuple in the second table.

Algorithm 2 implements that approach. Function ContinueJoin realizes the execution strategy described before. For a fixed amount of processing time (we use the number of outer while loop iterations as a proxy in our implementation) or until all input data is processed, it either increases "depth" (i.e., join order index i) to complete a partial tuple, satisfying all applicable predicates, further, or it advances tuples indices using Function NextTuple. The latter function increases the tuple indices for the current join order index or backtracks if the table cardinality is exceeded. Note that the same result tuple might be added multiple times in invocations of the execution engine for different join orders. However, we add tuple index vectors into a result *set*, avoiding duplicate entries (of course, two different tuple index vectors can represent two result tuples with the same values in each column).

6.4 Learning with Custom Join Operator

We discuss the main function (SkinnerC) learning optimal join orders using a customized execution engine (see Algorithm 3). The most apparent difference to the version from Section 5 is the lack of a dynamic timeout scheme. Instead, we use the same timeout for each invocation of the execution engine.

This becomes possible, since progress made when executing a specific join order is never lost. By minimizing the size of the execution state, we have enabled an efficient backup and restore mechanism (encapsulated by functions BackupState and RestoreState whose pseudo-code we omit) that operates only on a small vector of indices. The number of stored vectors is furthermore

ALGORITHM 2: Multi-way join algorithm supporting fast join order switching

```
// Advance tuple index in state $s$ for table at position $i$
// in join order $j$ for query $q$, considering tuple offsets $o$.
function NextTuple($q = R_1 \bowtie \cdots \bowtie R_m, j, o, s, i$)
    // Advance tuple index for join order position
    $s_{j_i} \leftarrow s_{j_i} + 1$
    // While index exceeds relation cardinality
    while $s_{j_i} > |R_{j_i}|$ and $i > 0$ do
        $s_{j_i} \leftarrow o_{j_i}$
        $i \leftarrow i - 1$
        $s_{j_i} \leftarrow s_{j_i} + 1$
    end while
    return $\langle s, i \rangle$
end function
// Execute join order $j$ for query $q$ starting from
// tuple indices $s$ with tuple offsets $o$. Add results
// to $R$ until time budget $b$ is depleted.
function ContinueJoin($q = R_1 \bowtie \cdots \bowtie R_m, j, o, b, s, R$)
    $i \leftarrow 1$ // Initialize join order index
    while processing time $< b$ and $i > 0$ do
        $t \leftarrow$ Materialize($R_{j_1}[s_{j_1}] \times \cdots \times R_{j_i}[s_{j_i}]$)
        if $t$ satisfies all newly applicable predicates then
            if $i = m$ then // Is result tuple completed?
                $R \leftarrow R \cup \{s\}$ // Add indices to result set
                $\langle s, i \rangle \leftarrow$ NextTuple($q, j, o, s, i$)
            else// Tuple is incomplete
                $i \leftarrow i + 1$
            end if
        else// Tuple violates predicates
            $\langle s, i \rangle \leftarrow$ NextTuple($q, j, o, s, i$)
        end if
    end while
    // Join order position 0 indicates termination
    return ($i < 1$)
end function
```

proportional to the size of the UCT tree. The fact that we do not lose partial results due to inappropriate timeouts anymore has huge impact from the theoretical perspective (see Section 7) as well as for performance in practice (see Section 8). Learning overheads are lower than before, since we only maintain a single UCT search tree accumulating knowledge from all executions.

6.5 Output Reward

In Section 5, we used a binary reward function based on whether the current batch was processed. We do not process data batch-wise anymore and must therefore change the reward function.

Our general goal is to make reward proportional to query evaluation progress. The UCT algorithm guarantees to achieve near-optimal reward. For a properly chosen reward function, this translates into near-optimal evaluation progress per time unit. We present multiple reward

ALGORITHM 3: Regret-bounded query evaluation using a customized execution engine

```
// Regret-bounded evaluation of SPJ query $q$,
// length of time slices is restricted by $b$.
function SkinnerC($q = R_1 \bowtie \cdots \bowtie R_m, b$)
    // Apply unary predicates and hashing
    $q \leftarrow$PreprocessingC($q$)
    $R \leftarrow \emptyset$ // Initialize result indices
    $finished \leftarrow$ false // Initialize termination flag
    while $\neg finished$ do
        // Choose join order via UCT algorithm
        $j \leftarrow$ UctChoice($T$)
        // Restore execution state for this join order
        $s \leftarrow$ RestoreState($j, o, S$); $s_{prior} \leftarrow s$
        // Execute join order during time budget
        $finished \leftarrow$ ContinueJoin($q, j, o, b, s, R$)
        // Update UCT tree via progress-based rewards
        RewardUpdate($T, j$, Reward($s - s_{prior}, j$))
        // Backup execution state for join order
        $\langle o, S\rangle \leftarrow$ BackupState($j, s, o, S$)
    end while
    return [Materialize($R_1[s_1] \times R_2[s_2] \ldots$)$|s \in R$]
end function
```

functions in the following. Besides using any single one of them, we can also combine them via a weighted sum. We compare different alternatives in our experimental evaluation (Section 8.3).

Query evaluation terminates once the entire query result is produced. Equivalently, we can say that query evaluation terminates once all input is processed. We present two reward functions that are based on those two complementary perspectives.

The first reward function is based on the frequency at which result tuples are generated. We count the number of join result tuples generated in each invocation of Algorithm 2 (Function ContinueJoin, result tuples are added in Line 23). The regret bounds proven for the UCT algorithm [31] apply to reward functions with values from the interval $[0, 1]$. We scale rewards to this interval as follows: As discussed before, we use the number of iterations of the while loop in Function ContinueJoin as a proxy for execution time. The per-invocation budget for the join algorithm b therefore restricts the number of loop iterations. At most one join result tuple is generated in each iteration. Hence, the maximal number of join result tuples per join invocation is b. We obtain a reward between zero and one by dividing the number of result tuples generated in a single invocation by b.

Figure 7 illustrates the principle of the output reward function and why it is helpful. We compare two join orders (left and right plot) processing the same query. We illustrate the number of result tuples generated per episode on the y axis (which is proportional to output reward) and the episode on the x axis. As can be seen from the figure, the number of tuples generated per episode varies for both join orders (e.g., due to inhomogeneous data). However, the second join order (right plot) finishes processing after only 5 episodes, while the first join order needs 10. Both join orders process the same query and must ultimately generate the same number of result tuples. As the second join order finishes faster, the average number of result tuples per episode must be higher. From the perspective of the UCT optimizer, the per-episode reward distribution associated with the

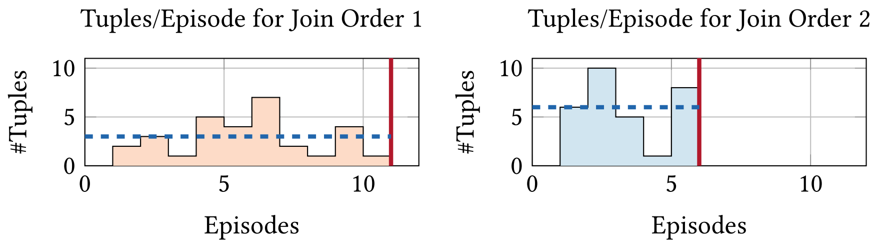


Fig. 7. The output reward metric calculates reward proportional to the number of join result tuples generated per episode. Comparing two join orders processing the same query, the total number of tuples (area under the curve) must be equal. However, as the query result is non-empty and since the second join order executes faster (solid red line marks termination), its average reward per episode (dashed line in blue) is higher.

second join order therefore has a higher mean. This makes it more likely that the UCT optimizer converges to that join order, thereby maximizing performance.

6.6 Input Reward

Using the output reward function and assuming a sufficiently large join result, SkinnerDB will converge to join orders that produce more result tuples per time unit (in average). As all join orders produce the same number of tuples in total, we thereby converge to faster join orders. However, this approach assumes a sufficiently large join result. It becomes problematic for extremely small result sizes. In the most extreme case, the query result is empty. In that case, all join orders have a constant reward of zero. Then, the aforementioned reward function will not allow us to distinguish more from less-efficient join orders.

Unlike the size of the join result, the sizes of all input tables are known before join execution starts. We can exploit this for a reward function that is based on the amount of input processed (rather than the amount of output generated).

We can adopt the following perspective on the join phase: *Conceptually*, we decide for each tuple in the Cartesian product between all joined tables whether it is part of the join result. Of course, our join algorithm does not consider each Cartesian product tuple explicitly (which would lead to prohibitive overheads). Instead, it implicitly excludes tuple groups that are guaranteed not to satisfy some join predicates. For instance, given a currently selected tuple in the first table (in join order), assume we cannot find a matching tuple in the second table. Then, all Cartesian product tuples using the current tuple in the first table are implicitly excluded. No tuple in that group can be part of the join result.

Intuitively, the input reward function measures the number of Cartesian product tuples (implicitly) considered per time unit. Independently from the join order, all Cartesian product tuples need to be considered to finish query processing. Hence, the join order considering most Cartesian product tuples per time unit (in average, over the entire query evaluation) is also the fastest one.

Our multi-way join algorithm advances from one tuple to the next in the ith table in join order after considering all input tuple combinations that use the currently selected tuples in the first i tables (i.e., it adds all associated result tuples to the result set). We prove that property formally in Section 7.1. Due to that property, we can efficiently calculate the percentage of the Cartesian product space considered during an episode by comparing the indices of selected tuples before and after the episode.

Denote by δ_i the delta between initial and final tuple index for the ith table in join order, comparing the state before and after an episode. Furthermore, denote by c_i the cardinality of the ith

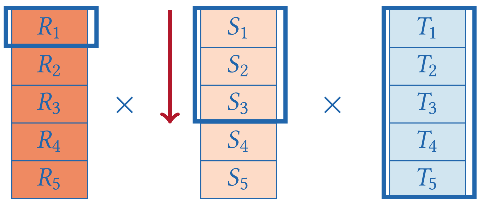


Fig. 8. The input reward function is based on the percentage of the space of Cartesian product tuples that has been considered per episode. When advancing by three tuples in the second table (for a fixed tuple in the first), we process a fraction of (1/5) · (3/5) of the Cartesian product space. All join orders must ultimately search 100% of the Cartesian product space, giving faster join orders a higher average reward per episode.

table in join order. We calculate input reward for that episode using the formula

$$\sum_{1 \leq i \leq m} \left(\delta_i / \left(\prod_{1 \leq k \leq i} c_i \right) \right),$$

where m denotes the number of tables. Note that δ_i may be negative if we reset and select the first tuple in a table (during backtracking). Nevertheless, the reward must always be non-negative, since we reset tuples only if advancing tuples for prior tables in join order (which have higher weight in the reward formula). The following example illustrates reward calculations:

Example 6.2. We reconsider the join invocation described in the previous example (Example 6.1 and Figure 6(a)). We assume an invocation budget of 10 steps. Here, as shown in Figure 6(a), we are joining three tables with five tuples each. The Cartesian product therefore has $5^3 = 125$ tuples. During the invocation represented in the figure, the tuple in the first table does not change. Hence, no reward is counted for the first table in join order alone. In contrast, we advance by three tuples in the second table (the fourth tuple is not yet completely processed after 10 steps). We implicitly considered 60% of tuples in the second table for one fixed tuple in the first table. Hence, the reward is $(1/5) \cdot (3/5) = 0.024$. For the third table, there is no change in tuples considered (as for the next tuple in the second table, we must start by considering the first tuple in the third table again). Hence, the total reward in this invocation is 0.024. Figure 8 illustrates the example, marking with blue boxes the part of the Cartesian product space that was covered.

For both reward functions, the per-invocation reward is not constant. Already due to inhomogeneous data, the per-invocation reward will vary even for a fixed join order. We can model the per-invocation reward for each join order as a probability distribution over reward values. The aforementioned reward metrics guarantee that the reward distributions with highest mean belong to the fastest join orders. As the UCT algorithm converges to actions with highest expected reward [31], we converge to optimal join orders. A more formal explanation is given in Section 7.

6.7 Sharing Progress between Join Orders

We have not yet discussed how our approach satisfies the third desiderata (sharing as much progress as possible among different join orders) mentioned at the beginning. We use, in fact, several techniques to share progress between different join orders (those techniques are encapsulated in Function RestoreState). Both of them exploit the fact that the multi-way join algorithm advances to the next tuple in a table, only after generating all results with the currently selected tuples up to that table. We prove that property in Section 7.1. Next, we discuss the two mechanisms by which progress is shared.

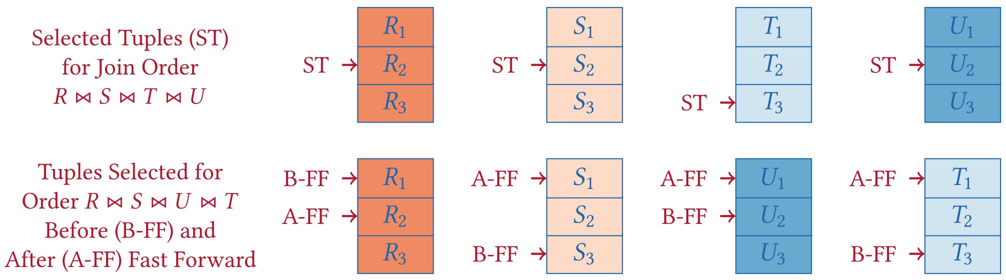

Selected Tuples (ST) for Join Order $R \bowtie S \bowtie T \bowtie U$
ST
R_1
R_2
R_3
S_1
S_2
S_3
T_1
T_2
T_3
U_1
U_2
U_3
Tuples Selected for Order $R \bowtie S \bowtie U \bowtie T$ Before (B-FF) and After (A-FF) Fast Forward
B-FF
A-FF

Fig. 9. We fast-forward the execution of join order $R \bowtie S \bowtie U \bowtie T$ (lower part) by integrating evaluation progress made for join order $R \bowtie S \bowtie T \bowtie U$ (upper part).

First, we use again offset counters to exclude for each table tuples that have been joined with all other tuples already (vector o in the pseudo-code, which is implicitly initialized to one). In contrast to the version from Section 5, offsets are not defined at the granularity of data batches but at the granularity of single tuples. This allows for a more fine-grained sharing of progress between different join orders than before.

Second, we share progress between all join orders with the same prefix. Whenever we restore state for a given join order, we compare execution progress between the current join order and all other orders with the same prefix (iterating over all possible prefix lengths). Comparing execution states s and s' for two join orders j and j' with the same prefix of length k (i.e., the first k tables are identical), the first order is "ahead" of the second if there is a join order position $p \leq k$ such that $s_{j_i} \geq s'_{j_i}$ for $i < p$ and $s_{j_p} > s'_{j_p} + 1$. In that case, we can "fast-forward" execution of the second join order, skipping result tuples that were already generated via the first join order. We do so by executing j' from a merged state s'' where $s''_{j'_i} = s_{j'_i}$ for $i < p$, $s''_{j'_p} = s_{j'_p} - 1$, and $s''_{j'_i} = o_{j'_i}$ for $i > p$ (since we can only share progress for the common prefix). Progress for different join orders is stored in the data structure represented as S in Algorithm 3, and Function RestoreState takes care of fast-forwarding (selecting the most advanced execution state among all alternatives).

Example 6.3. Figure 9 illustrates progress sharing for a query joining four tables. We consider two join orders, $R \bowtie S \bowtie T \bowtie U$ (upper part) and $R \bowtie S \bowtie U \bowtie T$ (lower part). We can share progress between them as they have a common prefix. ST marks the last selected tuple combination for the upper order. To reach this combination, all result tuples joining R_1 with others (if any) must have been generated. B-FF marks the last selected tuple combination for the lower order. If resuming execution with this combination, then we might generate redundant result tuples using R_1 (currently selected). Instead, we "fast-forward" when resuming execution for the lower order, thereby integrating progress achieved via the upper order. The selected tuple combination, after fast-forwarding, is marked up as A-FF. It skips any remaining join results that use tuple R_1.

6.8 Joins with Binary Equality Predicates

So far, we described the algorithm for queries with generic predicates. Our actual implementation uses an extended version supporting equality join predicates via hashing. If equality join predicates are present, then we create hash indices for all columns subject to equality predicates during pre-processing. Of course, creating hash indices to support all possible join orders creates overheads. However, those overheads are typically small, as only tuples satisfying all unary predicates are hashed. For tables without unary predicates, we can exploit previously created indices (on key and foreign key columns) if available. We analyze pre-processing overheads, with and without previously created indices, in Section 8.3.

ALGORITHM 4: Improved function for selecting the next tuple to examine, exploiting indices on equi-join columns

```
1: // Use index to propose next tuple in relation $r_2$ after tuple $t_2$ satisfying equi-join predicate
2: // between column $c_1$ in relation $r_1$ and column $c_2$ in $r_2$ with fixed tuple $t_1$ in $r_1$.
3: function NextMatch($\langle r_1.c_1, r_2.c_2\rangle, t_1, t_2$)
4:     // Retrieve join column value of current tuple in first relation
5:     $val \leftarrow$AccessColumn($r_1.c_1, t_1$)
6:     // Retrieve index on join column of current table
7:     $idx \leftarrow$GetIndex($r_2.c_2$)
8:     // Query index for next matching tuple after $t_2$
9:     return NextIndexed($idx, val, t_2$)
10: end function
11: // Propose index of next interesting tuple in $i$th table of join order $j$,
12: // starting from currently selected tuples specified in $s$ for query $q$.
13: function Advance($q = R_1 \bowtie \cdots \bowtie R_m, j, s, i$)
14:     // Advance at least by one tuple
15:     $n \leftarrow s_{j_i} + 1$
16:     // Iterate over binary equi-join predicates connecting current to prior tables
17:     for $\langle r_1.c_1, r_2.c_2\rangle \in q.binEquiPreds : r_2 = j_i \wedge \exists i' < i : j_{i'} = r_1$ do
18:         $n \leftarrow \max(n,$NextMatch$(\langle r_1.c_1, r_2.c_2\rangle, s_{r_1}, s_{r_2}))$
19:     end for
20:     return $n$
21: end function
22: // Advance tuple index in state $s$ for table at position $i$
23: // in join order $j$ for query $q$, considering tuple offsets $o$.
24: function NextTuple($q = R_1 \bowtie \cdots \bowtie R_m, j, o, s, i$)
25:     // Advance tuple index for join order position
26:     $s_{j_i} \leftarrow$Advance($q, j, s, i$)
27:     // While index exceeds relation cardinality
28:     while $s_{j_i} > |R_{j_i}|$ and $i > 0$ do
29:         $s_{j_i} \leftarrow o_{j_i}$
30:         $i \leftarrow i - 1$
31:         $s_{j_i} \leftarrow$Advance($q, j, s, i$)
32:     end while
33:     return $\langle s, i\rangle$
34: end function
```

Algorithm 4 extends the function for selecting the next tuple combination to analyze (it replaces the NextTuple function in Algrithm 2). In contrast to the generic version, it treats binary equi-join predicates separately. Instead of advancing the tuple index by one each time, it exploits indices on join columns to skip tuples that cannot satisfy some equi-join predicates. Function Advance exploits all applicable indices to propose a tuple index for the current table that *may* satisfy all applicable predicates.

Function Advance first identifies all binary equi-join predicates (provided by $q.binEquiPreds$) that are relevant. A predicate is relevant if two conditions are satisfied. First, the predicate must refer to the current table for which we seek to propose the next interesting tuple. Second, the predicate must connect the current table to one of the prior tables in join order. For such predicates, we find the next tuple in the current table, matching the currently selected tuple in the prior table. Function NextMatch finds the next matching tuple according to one specific equi-join predicate. It first retrieves the value in the join column for the tuple in the prior table. Then, it uses the index on the join column of the current table to retrieve the next matching tuple. Function NextIndexed returns the cardinality of the current table plus one if no matching tuple exists. Function Advance simply takes the maximum over the tuple indices proposed for each predicate.

The resulting tuple is not guaranteed to satisfy all equi-join predicates. Finding such a tuple (or concluding that none exists) may require intersecting multiple indices. Our goal is to try out join orders only for short periods of time. We want to avoid operations with potentially high time overheads. Hence, we only skip tuples that we can exclude with one single index access per predicate. The main loop of the join algorithm (Algorithm 2) will verify whether the resulting tuples satisfy all predicates indeed.

Figure 6(b) illustrates the principle for a query with binary equality join predicates, joining three tables (join order from left to right in the figure). Instead of considering each consecutive tuple in a table (compare to Figure 6(a)), we use indices on the join columns. Those indices are generated during the pre-processing phase of query evaluation on tuples satisfying all unary predicates of a query. Given currently selected tuples in prior tables in join order, we exploit the index (or indices) to retrieve positions of matching tuples in the current table. As illustrated in the figure, this means that less tuple combinations are considered and less predicates are evaluated.

6.9 Fast Backtracking

After considering all tuples in the current table, Algorithms 2 and 4 decrement the join index by one, thereby focusing on the previous table in join order. Implicitly, we assume that changing the selected tuple in the previous table may lead to new join result tuples. However, there are scenarios in which we can exclude that possibility.

Consider a join predicate, connecting the current table to a prior table in join order. Assume that we cannot find any matching tuple in the current table (excluding tuples before the table offset). In that case, no result tuples can be produced unless we select a different tuple in the prior table.

Algorithm 5 exploits this reasoning via a technique we call “Fast Backtracking.” A call to Function FastBack precedes the call to Function NextTuple in Line 29 of Algorithm 2. This means the function is called only if the tuple selected in the current table does not satisfy some predicates.

Algorithm 5 first checks whether the tuple index in the current table is at or before the offset. If so, then this indicates that no prior tuples in the current table needs to be considered (i.e., even if those prior tuples satisfy all applicable predicates, they have already been included in the join result). Next, the algorithm iterates over all binary equi-join predicates on the current table. For predicates connecting the current to prior tables, Algorithm 5 uses Function NextMatch (defined in Algorithm 4) to obtain the next matching tuple in the current table. A tuple index larger than the current table cardinality indicates no matching tuples.

If no matching tuples exists according to at least one predicate, then no matching tuples exist in the current table, given the selected tuples for previous tables in join order. The algorithm maps each such predicate to the index of the connected, previous table in join order. Variable *back*, representing the target index for fast backtracking, is set to the index of the first such table (in join order). For the selected tuple in that first table, no matching tuple exists in the current table. Hence, calling the NextTuple function cannot lead to result tuples unless it changes the tuple in the first table. Instead, we can immediately reset the tuples (i.e., select the first relevant tuple in each table) for all tables between the first and current table in join order. At the same time, we reset the join index to the first table. The new join index and tuple states are returned.

ALGORITHM 5: Use fast backtracking to focus directly on the first table in join order where a change in tuple selections can lead to new result tuples

```
// Backtrack from position $i$ in join order $j$ for query $q$ to first interesting
// table in join order, resetting tuple states $s$ and considering offsets $o$.
function FastBack($q = R_1 \bowtie \cdots \bowtie R_m, j, o, s, i$)
    // No prior tuples in current table satisfy all predicates?
    if $s_{j_i} \leq o_{j_i}$ then
        // Initialize target for backtracking
        $back \leftarrow i$
        // Iterate over equi-join predicates connected to current table
        for $\langle r_1.c_1, r_2.c_2\rangle \in q.binEquiPreds : r_2 = j_i$ do
            // Get index of connected table
            $i' \in \{1, \dots, m\}$ s.t. $j_{i'} = r_1$
            // Connected table appears before current one in join order?
            if $i' < i$ then
                // No matching tuple according to this predicate?
                if NextMatch($\langle r_1.c_1, r_2.c_2\rangle, s_{r_1}, s_{r_2}$)= $|R_{j_i}| + 1$ then
                    $back \leftarrow \min(back, i')$
                end if
            end if
        end for
        // Reset tuples between current join index and backtracking target
        for $x \leftarrow back + 1, \dots, i$ do
            $s_{j_x} \leftarrow o_{j_x}$
        end for
        $i \leftarrow back$ // Reset join index
    end if
    return $\langle s, i\rangle$
end function
```

We deliberately decided against several possible extensions of the algorithm. First, we restrict ourselves to cases in which we recognize quickly that no matching tuples exist in the current table. For instance, we do not consider cases in which matching tuples exist according to single predicates but not according to a combination of predicates. We could identify such cases by intersecting multiple indices during a single invocation of FastBack. Alternatively, we could keep track of the number of matching tuples in the current table and maintain state that persists across invocations. Both possibilities lead to additional time or space overheads. We opted for a relatively simple version. Second, we restrict fast backtracking to binary equi-join predicates. Currently, we use indices only to evaluate such predicates. Hence, the chances to recognize a lack of matching tuples quickly is best for those predicates.

Figure 10 illustrates fast backtracking. We join four tables (from left to right). Join predicates (in green) connect the first with the third, and the second with the fourth table. For the currently selected tuples in the first and second table, we cannot find any tuple in the third table satisfying all applicable join predicates (otherwise, the join algorithm would advance to the fourth table after having found a satisfying tuple in the third). The previous join version (see Figure 6(b)) would now backtrack to the second table in join order and select the next tuple there. However, changing the selected tuple in the second table cannot enable us to find matching tuples in the third table (since

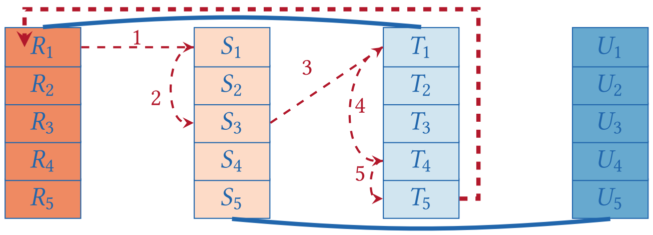


Fig. 10. Fast backtracking when joining four tables from left to right. Solid lines indicate connecting join predicates. Instead of returning to the second table after step 5, we jump directly to the first (as selecting a different tuple in the second table alone cannot yield join result tuples).

no join predicate connects the second to the third table). To find matching tuples in the third table (a pre-condition to generating join result tuples), we must change the selected tuple in the first. Fast backtracking exploits that fact and directly backtracks to the first (instead of the second) table in join order. Thereby, we skip steps for considering alternative tuples for the second table and trying to find matching tuples in the third.

7 FORMAL ANALYSIS

We prove correctness (see Section 7.1), introduce assumptions and notations for our regret analysis (see Section 7.2), and calculate regret bounds (see Section 7.3).

7.1 Correctness

Next, we prove correctness (i.e., that each algorithm produces a correct query result). We start by analyzing Skinner-G.

Lemma 7.1. *Skinner-G produces each join result tuple at least once.*

Proof. We conduct a proof by contradiction. Assume a row r in the correct join result is not generated by Algorithm 1. Algorithm 1 terminates once it finds a first table for which the offset exceeds the number of batches per table (condition $o_i > b$ in Line 17). Let i^* be the table satisfying the termination condition. Result row r joins tuples from base tables. Denote by b^* the batch containing the tuple from table i^* that r was formed from. Offsets are generally increased in steps of one. If the offset for table i^* exceeds the number of batches, then we must have had $o_{i^*} = b^*$ in a prior iteration and o_{i^*} was incremented afterwards. Offsets are incremented only if the current batch was successfully processed (Line 27). In that case, all result rows that can be formed from tuples in the current batch have been added to the join result. This applies to r, which is therefore generated by Algorithm 1, leading to a contradiction. □

Lemma 7.2. *Skinner-G produces no incorrect duplicates in the join result.*

Proof. We conduct a proof by contradiction. Assume a row r was generated too often by Algorithm 1. We only join one batch with the remaining tables in each iteration. This operation cannot lead to incorrect duplicates (assuming that the underlying execution engines works correctly). Hence, if there are incorrect duplicates, then we must have generated the same tuple twice across different iterations. Consider the first of those two iterations. Denote by i^* the left-most table of the join order used in that iteration and by b^* the batch in the left-most table. We assume that this iteration changed the join result (by inserting the duplicate), which implies that the batch was successfully processed. But in that case, the offset o_{i^*} was increased from b^* to $b^* + 1$. For all of the following iterations, we consider only batches starting from $b^* + 1$ for table i^* (see Line 23). Hence, we cannot regenerate the same tuple, leading to a contradiction. □

Corollary 7.3. *Skinner-G produces the correct query result.*

Proof. This is directly implied by the combination of Lemmas 7.1 and 7.2. □

Skinner-H is based on Skinner-G. As shown next, correctness of Skinner-H follows from the correctness of Skinner-G.

Theorem 7.4. *Skinner-H produces the correct query result.*

Proof. Skinner-H terminates for one out of two reasons. Either the plan proposed by the traditional optimizer finishes. Assuming that the traditional optimizer and execution engine are working correctly, the generated result must be correct. Alternatively, Skinner-H may terminate if intra-query learning finishes. During intra-query learning phases, Skinner-H works like Skinner-G. We have shown that Skinner-G produces a correct join result (see Corollary 7.3). Hence, Skinner-H produces a correct result in the second scenario as well. □

Next, we prove correctness of Skinner-C. We start by analyzing the specialized, multi-way join algorithm that is used by Skinner-C. The following proofs apply to Algorithm 2, as well as to the refined version (Algorithm 4).

Theorem 7.5. *Not considering progress sharing, the multi-way join algorithm used by Skinner-C advances to the next tuple in the xth table in join order not before generating all result tuples with the currently selected tuples in tables one to x in join order.*

Proof. Assume that x is equal to the number of tables (induction start), i.e., we consider the last table in join order. For each tuple selected in the last table, the algorithm verifies whether all join predicates are satisfied. If so, then it adds the tuple into the result set. The theorem holds in all cases. Now, assuming the theorem holds starting from some x (with $x > 1$), we prove that it holds for $x-1$ as well (induction step). The tuple in table $x-1$ is changed in three scenarios. First, it may be changed during backtracking (Line 7) if the tuple index exceeds the table cardinality. Then, the tuple index does not represent an actual tuple and the theorem is trivially satisfied. Second, it may be changed after we determine that the currently selected tuples in tables one to i violate a join predicate. Then, no result tuples can be formed with the currently selected tuples and the theorem holds again. Third, it may get changed after i was increased and decreased again. Algorithm 2 starts with the first and considers all tuples in table x after i increases to x. Exploiting the inductional assumption, we know that Algorithm 2 advances to the next tuple in the xth table only after adding all result tuples that use the currently selected tuples in tables one to x. Hence, after decreasing i to $x-1$ again, we have added all result tuples with the currently selected tuples in tables one to $x-1$. Compared to Algorithm 2, Algorithm 4 only differs by skipping tuples that do not satisfy binary equality join predicates (by exploiting join indices). Hence, the proof generalizes to that algorithm as well. Finally, note that we restore the last selected tuples when resuming execution for a specific join order (unless progress sharing is considered). Therefore, join interruptions without progress sharing (see next) do not influence correctness. □

The multiway join algorithm may skip tuples due to progress sharing. Next, we prove that the progress-sharing mechanism does not prevent Skinner-C from producing a complete result.

Theorem 7.6. *Tuples skipped by Skinner-C due to progress sharing are not required for a complete join result.*

Proof. Skinner-C shares evaluation progress across join orders via two related, but slightly different, mechanisms. First, we share progress via offsets. For each table, the offset is the number of tuples for which all associated join result tuples were already generated. We increase the offset for the left-most table in join order whenever the multiway join algorithm advances to the next tuple in that table. According to Theorem 7.5, all result tuples that use the currently selected tuple in the first table have been generated before advancing. Hence, we can skip the offset tuples during processing, independently of the join order.

Second, we share progress across join orders with the same prefix. Assume we update the currently selected tuples of a join order j_1 by exploiting progress achieved by join order j_2, assume both orders have the same tables until position x. When resuming join order j_1, we select the same tuples as j_2 before position x, the prior tuple at position x, and the first tuple in each table after that. As both orders are equal up to position x, they consider the same tuple combinations for the associated tables in the same order. According to Theorem 7.5, before advancing to the current tuple in table x, all join result tuples that can be formed from prior tuples in that table were generated. Hence, the current join result may only lack tuples that can be formed using the currently selected tuple in the xth table. We make sure that all of them are considered by resetting tuples in all of the following tables to the first tuple (potentially skipping tuples in the offset, which is correct as shown before). □

Theorem 7.7. *Skinner-C produces the correct query result.*

Proof. Executing different join orders in isolation, the multiway join algorithm used by Skinner-C generates each join result tuple at least once (according to Theorem 7.5). Furthermore, merging progress across join orders does not lead to missed result tuples according to Theorem 7.6. Finally, note that Skinner-C inserts all result tuples into a result set (variable R in Algorithm 2). More precisely, this set stores each result tuple as a vector, indicating for each base table the index of the selected tuple in it. So, if two join orders form the same result tuple (i.e., using the same combination of base table tuples), then only one of them remains in the result set. This is due to the set semantics of variable R (R is implemented as a hash set of tuple index vectors in the system). Hence, each join result tuple is produced at least once and duplicates are eliminated. □

7.2 Regret Model and Assumptions

Expected regret measures the quality of a reinforcement learning algorithm. Regret is typically defined as the distance between reward of the studied algorithm and reward of an optimal policy. We model join order selection for a query as an episodic MDP with stochastic rewards (see Section 4.2). Hence, an optimal policy is a fixed action sequence with maximal expected reward [31]. In our scenario, a fixed optimal policy corresponds to a fixed join order. We generally link reward to evaluation progress per episode (e.g., successfully processed data batches for Skinner-G or produced join result tuples for Skinner-C). The join order with maximal expected reward per episode is therefore the one that works best when averaging over all data; e.g., assuming a non-empty join result, the join order producing most result tuples per time slice in average finishes first (comparing alternative join orders for the same query). When referring to an "optimal execution" in the following, we mean an optimizer that selects the same optimal join order for SkinnerDB in each episode. We calculate regret by comparing expected time of the optimal execution to the one achieved by the SkinnerDB optimizer. Intuitively, given SkinnerDB's execution time for a query, we ask which part of it was wasted, in expectation, due to not selecting a fixed, optimal join order.

We denote execution time by n and optimal time by n^*. Skinner-G and Skinner-H choose timeout levels (represented by the y axis in Figure 3) that we denote by l. We use the subscript notation (e.g., n_l) to denote accumulated execution time spent with a specific timeout level. We study regret for fixed query properties (e.g., the number of joined tables, m, or the optimal reward per time slice, r^*) for growing amounts of input data (i.e., table size) and execution time. In particular, we assume that execution time, in relation to query size, is large enough to make the impact of transitory regret negligible [13]. We focus on regret of the join phase, as pre-processing overheads are linear in data and query size (while post-processing overheads are polynomial in query and join result size). We assume that time slices are chosen large enough to make overheads related to learning and join order switching negligible. Specifically for Skinner-G and Skinner-H, we assume that the optimal timeout per time slice applies to all batches. To simplify the analysis, we study slightly simplified versions of the algorithms from Section 4. In particular, we assume that offsets are only applied to exclude tuples for the left-most table in the current join order. All progress is shared between join orders having the same left-most table. For Skinner-C, we assume that the simpler reward function (progress in left-most table only) is used. We base our analysis on the properties of the UCT variant proposed by Kocsis and Szepesvari [31].

Our analysis focuses on join orders executed in SkinnerDB. The optimal join order may generally differ across database systems, depending on available operators or indices. This means that the optimal join order for SkinnerDB is not necessarily optimal for other systems. Also, our approach focuses on join ordering alone, Skinner-G and Skinner-H rely on the traditional optimizer to select operators and indices (while Skinner-C uses one single join operator). However, e.g., if execution time can be approximated by a cost metric that is not influenced by operator choices, data access paths, and batch-wise execution (e.g., the C_{out} cost metric, which sums intermediate result cardinality [11]), then the optimal order of SkinnerDB is optimal in general.

7.3 Regret Bounds

Before analyzing Skinner-G, we first prove several properties of the pyramid timeout scheme introduced in Section 5.

Lemma 7.8. *The number of timeout levels used by Skinner-G is upper-bounded by* $\log(n)$.

Proof. We add a new timeout level L, whenever the equation $n_l \geq n_L + 2^L$ is satisfied for all $0 \leq l < L$ for the first time. As n_l is generally a sum over powers of two (2^l), and as $n_L = 0$ before L is used for the first time, the latter condition can be tightened to $2^L = n_l$ for all $0 \leq l < L$. Hence, we add a new timeout whenever the total execution time so far can be represented as $L \cdot 2^L$ for $L \in \mathbb{N}$. Assuming that n is large, specifically $n > 1$, the number of levels grows faster if adding levels whenever execution time can be represented as 2^L for $L \in \mathbb{N}$. In that case, the number of levels can be bounded by $\log(n)$ (using the binary logarithm). □

Lemma 7.9. *The total amount of execution time allocated to different (already used) timeout levels cannot differ by more than factor two.*

Proof. Assume the allocated time differs by more than factor two between two timeout levels, i.e., $\exists l_1, l_2 : n_{l_1} > 2 \cdot n_{l_2}$ (and $n_{l_1}, n_{l_2} \neq 0$). Consider the situation in which this happens for the first time. Since $\forall i : n_i \geq n_{i+1}$, we must have $n_0 > 2 \cdot n_L$ where L is the largest timeout level used so far. This was not the case previously, so we either selected timeout level 0 or a new timeout level L in the last step. If we selected a new timeout level L, then it was $n_l \geq n_L + 2^L$ for all $0 \leq l < L$, which can be tightened to $\forall 0 \leq l < L : n_l = 2^L$ (exploiting that $n_L = 0$ previously and that timeouts are powers of two). Hence, selecting a new timeout cannot increase the maximal ratio of time per level. Assume now that timeout level 0 was selected. Denote by $\delta_i = n_i - n_{i+1}$ for $i < L$ the difference in allocated execution time between consecutive levels before the last selection. It is $\delta_i \leq 2^i$, since n_i is increased in steps of size 2^i and strictly smaller than 2^{i+1} (otherwise, level $i + 1$ or a higher one would have been selected). It was $n_0 - n_L = \sum_{0 \leq i < L} \delta_i \leq \sum_{0 \leq i < L} 2^i < 2^L$. On the other side, it was $n_L \geq 2^L$ (as $n_L \neq 0$ and, since n_L is increased in steps of 2^L). After n_0 is increased by one, it is still $n_0 \leq 2 \cdot n_L$. The initial assumption leads always to a contradiction. □

We are now ready to provide worst-case bounds on the expected regret when evaluating queries via Skinner-G.

Theorem 7.10. *Expected execution time regret of Skinner-G is upper-bounded by* $(1-1/(\log(n)\cdot m\cdot 4))\cdot n+O(\log(n))$.

Proof. Total execution time n is the sum over execution time components n_l that we spent using timeout level l, i.e., we have $n=\sum_{0\leq l\leq L}n_l$ where $L+1$ is the number of timeout levels used. It is $L+1\leq\log(n)$ due to Lemma 7.8 and $\forall l_1,l_2\in L: n_{l_1}\geq n_{l_2}/2$ due to Lemma 7.9. Hence, for any specific timeout level l, we have $n_l\geq n/(2\cdot\log(n))$. Denote by l^* the smallest timeout, tried by the pyramid timeout scheme, which allows to process an entire batch using the optimal join order. It is $n_{l^*}\geq n/(2\cdot\log(n))$. We also have $n_{l^*}=n_{l^*,1}+n_{l^*,0}$ where $n_{l^*,1}$ designates time spent executing join orders with timeout level l^* that resulted in reward 1, $n_{l^*,0}$ designates time for executions with reward 0. UCT guarantees that expected regret grows as the logarithm in the number of rounds (which, for a fixed timeout level, is proportional to execution time). Hence, $n_{l^*,0}\in O(\log(n_{l^*}))$ and $n_{l^*,1}\geq n_{l^*}-O(\log(n_{l^*}))$. Denote by b the number of batches per table. The optimal algorithm executes b batches with timeout l^* and the optimal join order. Skinner can execute at most $m\cdot b-m+1\in O(m\cdot b)$ batches for timeout l^* before no batches are left for at least one table, terminating execution. Since l^* is the smallest timeout greater than the optimal time per batch, the time per batch consumed by Skinner-G exceeds the optimal time per batch at most by factor 2. Hence, denoting by n^* time for an optimal execution, it is $n^*\geq n_{l^*,1}/(2\cdot m)$, therefore $n^*\geq(n_{l^*}-O(\log(n)))/(2\cdot m)\geq n_{l^*}/(2\cdot m)-O(\log(n))$ (since m is fixed), which implies $n^*\geq n/(4\cdot m\cdot\log(n))-O(\log(n))$. Hence, the regret $n-n^*$ is upper-bounded by $(1-1/(4\cdot m\cdot\log(n)))\cdot n+O(\log(n))$. □

Next, we analyze regret of Skinner-H.

Theorem 7.11. *Expected execution time regret of Skinner-H is upper-bounded by* $(1-1/(\log(n)\cdot m\cdot 12))\cdot n+O(\log(n))$.

Proof. Denote by n_O and n_L time dedicated to executing the fixed optimal plan or learned plans, respectively. Assuming pessimistically that optimizer plan executions consume all dedicated time without terminating, it is $n_O=\sum_{0\leq l\leq L}2^l$ for a suitable $L\in\mathbb{N}$ at any point. Also, we have $n_L\geq\sum_{0\leq l<L}2^l$ as time is divided between the two approaches. It is $n_L/n\geq(2^L-1)/(2^{L+1}+2^L-2)$, which converges to 1/3 as n grows. We obtain the postulated bound from Theorem 7.10 by dividing the "useful" (non-regret) part of execution time by factor three. □

The following theorem is relevant if traditional query optimization works well (and learning creates overheads):

Theorem 7.12. *The maximal execution time regret of Skinner-H compared to traditional query execution is* $n\cdot 4/5$.

Proof. Denote by n^* execution time of the plan produced by the traditional optimizer. Hence, Skinner-H terminates at the latest once the timeout for the traditional approach reaches at most $2\cdot n^*$ (since the timeout doubles after each iteration). The accumulated execution time of all prior invocations of the traditional optimizer is upper-bounded by $2\cdot n^*$ as well. At the same time, the time dedicated to learning is upper-bounded by $2\cdot n^*$. Hence, the total regret (i.e., added time compared to n^*) is upper-bounded by $n\cdot 4/5$. □

Finally, we analyze expected regret of Skinner-C. Among the two reward metrics presented in Sections 6.5 and 6.6, we consider the input-based metric. We cannot give regret bounds for the output-based metric in general, as join results may be empty. If so, then the output metric is not helpful.

Theorem 7.13. *Expected execution time regret of Skinner-C is upper-bounded by* $(1 - 1/m) \cdot n + O(\log(n))$.

Proof. Regret is the difference between optimal execution time, n^*, and actual time, n. It is $n - n^* = n \cdot (1 - n^*/n)$. Denote by R the total reward achieved by Skinner-C during query execution and by r the average reward per time slice. It is $n = R/r$. Denote by r^* the optimal expected reward per time slice. Reward is calculated as the relative tuple index delta in the left-most table (i.e., tuple index delta in left-most table divided by table cardinality). Using a fixed join order with optimal expected reward terminates once the accumulated reward reaches one. Hence, we obtain $n^* = 1/r^*$. We can rewrite regret as $n - n^* = n \cdot (1 - (1/r^*)/(R/r)) = n \cdot (1 - r/(R \cdot r^*))$. The difference between expected reward and optimal reward is bounded as $r^* - r \in O(\log(n)/n)$ [31]. Substituting r by $r^* - (r^* - r)$, we can upper-bound regret by $n \cdot (1 - 1/R) + O(\log(n))$. Denote by $R_t \leq R$ rewards accumulated over time slices in which join orders starting with table $t \in T$ were selected. Skinner-C terminates whenever $R_t = 1$ for any $t \in T$. Hence, we obtain $R \leq m$ and $n \cdot (1 - 1/m) + O(\log(n))$ as upper bound on expected regret. □

Instead of the (additive) difference between expected and execution time of an optimal fixed order, we can also consider the ratio.

Theorem 7.14. *The ratio of expected to optimal execution time for Skinner-C is upper-bounded and that bound converges to m as n grows.*

Proof. Let $a = n - n^*$ be additive regret, i.e., the difference between actual and optimal execution time. It is $n^* = n - a$ and, as $a \leq (1 - 1/m) \cdot n + O(\log(n))$ due to Theorem 7.13, it is $n^* \geq n - (1 - 1/m) \cdot n - O(\log(n)) = n/m - O(\log n) = n \cdot (1/m - O(\log(n))/n)$. Optimal execution time is therefore lower-bounded by a term that converges to n/m as n grows. Then, the ratio n/n^* is upper-bounded by m. □

8 EXPERIMENTAL EVALUATION

We describe our experimental setup in Section 8.1. Section 8.2 reports performance measurements for SkinnerDB and other systems on various benchmarks. Section 8.3 analyzes in more detail which SkinnerDB features contribute to its performance. Section 8.4 reports additional statistics with regards to the runtime of different execution phases. In Section 8.5, we evaluate several alternative reinforcement learning algorithms in terms of their performance and robustness to changing scenarios. Section 8.6 analyzes the performance tradeoffs when executing SkinnerDB on top of traditional execution engines. Finally, we summarize our findings in Section 8.7.

8.1 Experimental Setup

The following experiments focus on the performance of SkinnerDB. Unless noted otherwise, the term "SkinnerDB" designates our main variant, Skinner-C.

We use a budget of $b = 500$ steps per time slice and set the exploration weight w to 10^{-5}. By default, we use a reward function that combines input and output reward with weights 0.5, respectively. We parallelize the pre-processing phase and exploit hash indices on key and foreign key columns. Our join optimizer considers the full space of join orders, including Cartesian product joins. We restrict the amount of heap space to 16 GB ($-Xmx = 16g$) and select the concurrent mark sweep garbage collector ($+UseConcMarkSweepGCSkinner.jar$).

For Skinner-G and Skinner-H, we set the exploration factor to $w = \sqrt{2}$ and use the single tree variant (see Section 5.3). We use Postgres as underlying execution engine and partition into 10,000 batches per table (if tables have less than 10,000 rows, some batches are empty). We use a base timeout of 20 milliseconds per batch and ramp up timeouts by a factor of two. We switch from batched to non-batched execution after 500 UCT samples. For Skinner-H, we execute the plan proposed by the original optimizer with a timeout of initially five seconds.

Besides SkinnerDB, we use Postgres (version 10.10), MonetDB (MonetDB Database Server v11.35.9, Nov2019-SP1), and TelegraphCQ,[2] a Postgres-based implementation of Eddies, as baselines. We compared SkinnerDB against older versions of Postgres and MonetDB in our prior work [52].

Our goal was to select benchmarks that cover the full range from particularly easy to particularly hard to optimize. We focus on query optimization challenges that relate to cost estimation. Hence, we judge benchmarks based on how difficult it is to estimate plan cost before execution. For TPC-H [51], queries contain standard SQL predicates, and data is synthetically generated with uniform distribution. This makes TPC-H an easy benchmark from the optimizer's point of view. The IMDB join order benchmark [25] operates on real data with realistic amounts of skew. This benchmark has been designed to be more realistic in terms of optimizer challenges. We consider multiple variants. IMDB-S designates in the following the queries of the original benchmark, applied to the May 2013 version of the IMDB database (prior work evaluates Postgres on this version [25]). IMDB-L uses the same queries but applies them to an extended version of the same data: the last publicly available version from December 2017.[3] We consider IMDB-S and IMDB-L as moderately difficult from the optimizer's perspective.

Finally, we consider multiple benchmarks that make optimization hard. First, we consider a variant of IMDB that replaces unary predicates in queries by semantically equivalent, user-defined functions. We use the same data as for IMDB-L for this benchmark, hence the name IMDB-L-UDF. Here, query optimization becomes hard, since predicates must be treated as black boxes by the optimizer. This makes the benchmark representative for cases where selectivity estimation is hard not because of data skew but of predicate properties. Also, we consider the JCC-H benchmark [8]. This benchmark uses the same query templates as TPC-H but operates on artificially generated, highly skewed data. This benchmark is representative for cases where query optimization becomes hard, not due to predicate properties but due to data skew.

For TPC-H and JCC-H, we generate data with scaling factor one (i.e., the database has a size of around 1 GB). For IMDB-S, the data size is around 4 GB, and it is around 7 GB for IMDB-L and IMDB-L-UDF. For TPC-H and JCC-H, we report results for 19 out of the 22 queries. The remaining three queries are not yet supported by our newest SkinnerDB version due to lack of support for outer joins (TPC-H Query 13), views (Query 15), and string manipulation functions (substring function in Query 22). None of those limitations is fundamental, and we plan to add support in coming SkinnerDB versions. We use a timeout of five minutes per benchmark and baseline.

Unless noted otherwise, the runtimes reported for SkinnerDB assume that indices on (primary and foreign) key columns were created before runtime. SkinnerDB may create additional indices on filtered tables at runtime. This index creation time is counted towards its runtime. MonetDB automatically creates indices on key and foreign key columns.[4] Postgres automatically creates indices on primary key columns. We additionally created indices on foreign key columns for TPC-H and JCC-H (for the IMDB benchmarks, foreign key indices were shown to make optimization

[2] http://telegraph.cs.berkeley.edu/telegraphcq/v0.2/.
[3] ftp://ftp.fu-berlin.de/pub/misc/movies/database/frozendata/.
[4] https://www.monetdb.org/Documentation/SQLReference/PerformanceOptimization/PerformanceTips.

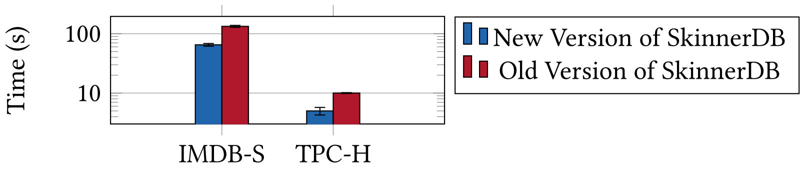


Fig. 11. Comparison between SIGMOD 2019 version of SkinnerDB and re-implementation: We observe significant speedups over multiple benchmarks.

harder for the Postgres optimizer [25]). Unless noted otherwise, as recommended [25], we disable Postgres' nested loop joins for the IMDB benchmarks.

The following experiments were executed on two servers with similar properties. The first server is a Dell PowerEdge R640 server with two Intel Xeon 2.3 GHz CPUs with 24 cores, 256 GB of RAM, and 1 TB of disk space. Server 2 is a Dell T640 machine with 384 GB of main memory, 10 TB of hard disk, and two Intel Xeon 2.3 GHz CPUs with 32 physical cores total. Both servers run Ubuntu 11.8 and the OpenJDK 64-Bit Server JVM. Unless noted otherwise, the following experiments were executed on Server 2.

8.2 Performance Comparisons

In this subsection, we evaluate the following hypotheses:

Hypothesis 1. *Our re-implementation of SkinnerDB outperforms the original [52].*

Hypothesis 2. *SkinnerDB can select better join orders than traditional optimizers (according to the C_{out} metric) if cardinality estimation is difficult.*

Hypothesis 3. *SkinnerDB can perform better than processing engines with traditional optimizers for benchmarks where query optimization is difficult.*

Hypothesis 4. *The performance of SkinnerDB is more robust to query rewritings than standard query optimization.*

First, we compare our new SkinnerDB version against the original implementation [52]. We compare both systems on the hardware (Server 1) that was used for the original experiments. Figure 11 compares runtimes (in seconds) on the IMDB and TPC-H benchmark. In this and the following plots showing averages, we report 90% confidence bounds as error bars, derived from three runs. For TPC-H, and only for this comparison, we restrict ourselves to the queries on which the old SkinnerDB version was evaluated on (Queries 2, 3, 5, 7, 8, 9, 10, 11, 18, 21). The new SkinnerDB version was rewritten from scratch. It is complementary to the prior version in multiple ways. For instance, the old version only has an interpreter for evaluating SQL expression, while the new version exclusively uses a byte code compiler. Also, the old version operates only on raw strings, while the new version encodes strings using a dictionary. The complementary nature of the two implementations makes it hard to trace back performance differences to specific features. However, in aggregate, the new SkinnerDB version is about twice as fast for both benchmarks. At the same time, it consumes significantly less main memory than the prior version (e.g., 16 GB versus 38 GB for the database of the join order benchmark with indices). Those results generally demonstrate that a significant part of the performance gap, separating SkinnerDB from other systems in our previous experiments [52], is due to a non-optimized implementation (and not intrinsic to the particularities of intra-query learning). This verifies Hypothesis 1. We use the new version for the following experiments.

Next, we compare the new SkinnerDB version against several baselines. Figure 12 reports runtimes on various benchmarks. Compared to our prior evaluation [52], we have upgraded not only

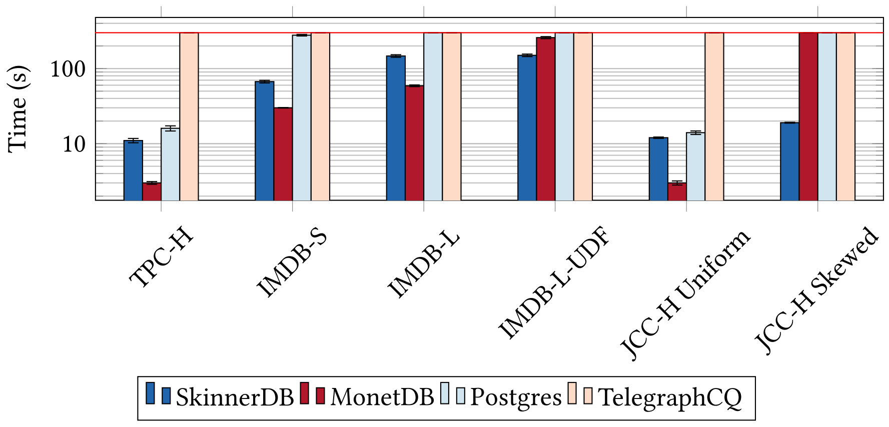


Fig. 12. Total runtime for different benchmarks and systems, the red line marks the timeout.

SkinnerDB but also the MonetDB and Postgres baselines. The difference is significant. Postgres introduced parallel operators in version 9.6 and is significantly faster at processing query plans on our multi-core machine. MonetDB chooses better join orders for two queries of the join order benchmark, which together accounted for a large part of total runtime in our previous experiments.

Overall, MonetDB performs best (among all compared systems) for benchmarks where query optimization is relatively easy. This applies in particular to the TPC-H benchmark (and the non-skewed JCC-H version). Here, the synthetically generated data is uniform and predicates (mostly equality and inequality predicates) are rather easy to analyze. This renders cardinality estimation (and therefore execution cost estimation) more reliable. On the other side, SkinnerDB excels for benchmarks where query optimization is difficult. This applies to the JCC-H benchmark as well as to the IMDB-L-UDF. JCC-H is very challenging for traditional query optimizers due to data skew. IMDB-L-UDF is difficult, as query predicates (user-defined function predicates) are difficult to analyze, even if fairly accurate data statistics are available. In both cases, SkinnerDB, which does not rely on data statistics or cardinality models, benefits (validating Hypothesis 3).

We analyzed the query plans generated by different approaches to explain the observed performance differences. The C_{out} cost metric [11] evaluates join orders in terms of the sizes of intermediate results they generate. For a given join order, we simply sum up the cardinality of all intermediate results. Query execution time is a function of both, the quality of selected plans and the quality of the execution engine processing them. The C_{out} cost metric, however, allows us to separate impact of optimization from the impact of the execution engine.

Figure 13 compares the performance of MonetDB and SkinnerDB, according to execution time and the C_{out} cost metric. We use the IMDB-S benchmark. For each query, we calculate relative performance by dividing the associated number of MonetDB by the one of SkinnerDB. We partition queries based on relative performance, rounding the logarithm (with base two) to the nearest integer. In Figure 13, we report the number of queries for each partition, considering relative runtime for the left plot and the C_{out} metric in the plot on the right.

SkinnerDB produces less intermediate result tuples for 76 out of the 113 queries (for a total of 100 million result tuples over the entire benchmark, compared to 270 million result tuples for MonetDB). Also, the queries where SkinnerDB has highest relative tuple count, compared to MonetDB, are cheap to execute. SkinnerDB produces 1,441 tuples, on average, for the three queries with maximal relative tuple count (it is 15 million tuples for MonetDB). However, MonetDB takes only 30 seconds, compared to SkinnerDB's 67 seconds, to execute the entire benchmark. Hence, it seems

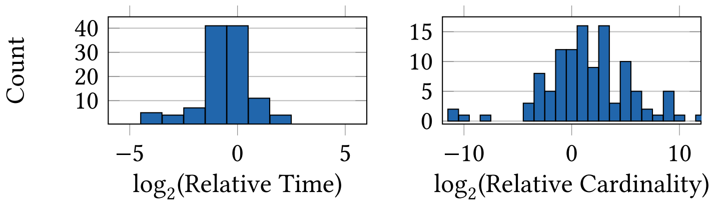


Fig. 13. We compare execution time and intermediate result cardinality of MonetDB and SkinnerDB. We report the number of queries in IMDB-S with specific ratios for the corresponding performance metrics (value of MonetDB divided by the one of SkinnerDB). MonetDB is faster for most queries (see left figure), while SkinnerDB tends to select better join orders according to the C_{out} metric (see right figure).

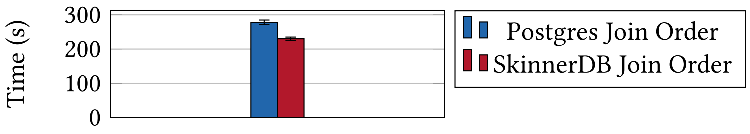


Fig. 14. Comparison of Postgres performance on IMDB-S with original join orders and with join orders proposed by SkinnerDB.

that SkinnerDB selects better join orders (Hypothesis 2), while MonetDB is faster at executing them. SkinnerDB uses intra-query learning to converge to optimal join orders and does not rely on potentially inaccurate cardinality estimates. This explains why SkinnerDB selects better join orders. On the other side, intra-query learning creates overheads due to learning and plan switching. Also, SkinnerDB is implemented in Java and does not parallelize the join phase. This explains why per-tuple processing overheads are higher for SkinnerDB. On IMDB-S, better join order selections cannot yet outweigh higher per-tuple processing overheads. This changes for the IMDB-L-UDF variant. Switching from standard predicates to UDF predicates does not influence the join order selections of SkinnerDB, while it worsens the join order quality for MonetDB (cost according to the C_{out} metric increases by factor 2.5 for MonetDB when switching predicates). Here, the added benefit of more reliable join order selections dominates overheads due to intra-query learning.

We performed an additional experiment to corroborate our assumption that SkinnerDB generates high-quality join orders. Postgres allows bypassing the original optimizer to enforce specific join orders (for other database systems without this feature, forcing join orders would require explicitly materializing intermediate results with causes overheads). We compare the performance of Postgres on IMDB-S with join orders proposed by the Postgres optimizer versus join orders that SkinnerDB converges to. Figure 14 shows the results with 90% confidence intervals. Note that we deactivated nested loop joins in both cases, as recommended by the benchmark authors for Postgres [25]. It turns out that Postgres performs better when using the join orders from SkinnerDB.

IMDB-L and IMDB-L-UDF execute on the same data, and the queries are semantically equivalent. SkinnerDB's performance is robust while standard optimization can suffer due to those query rewritings (verifying Hypothesis 4). It does not necessarily take user-defined predicates or high data skew to make optimization difficult. We performed a micro-benchmark with an expensive query (query 16b) from the IMDB join order benchmark. We rewrote the query by applying small transformations, without changing its semantics. We considered the first predicate in that query (`cn.country_code ='[us]'`) and added one more redundant predicate. We tried adding an inequality predicate (`cn.country_code <>'[de]'`) as well as a like predicate ((`cn.country_code like '[us\%'`)). Besides that, we also tried replacing the equality condition by an equivalent IN expression ((`cn.country_code IN ('[us]')`)). As shown in Figure 15, those

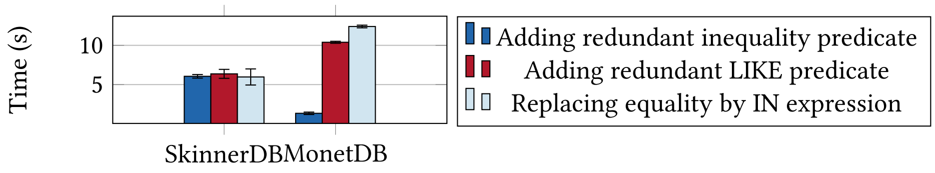


Fig. 15. Robustness of query plan choices towards small rewrites for IMDB query 16b.

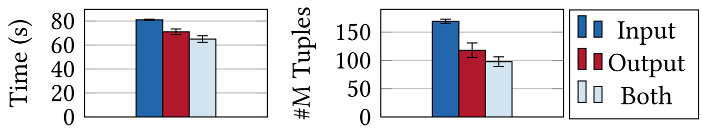


Fig. 16. Impact of reward function (IMDB-S).

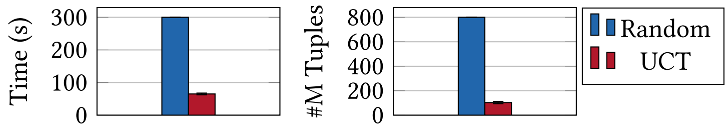


Fig. 17. Impact of search strategy on IMDB-S benchmark. The values for the random algorithm are lower bounds due to a timeout at five minutes.

rewrites influence the performance of MonetDB significantly due to different join order choices. On the other side, the performance of SkinnerDB remains stable (except for random variations). This is to be expected: As SkinnerDB optimizes based on observed performance alone, the query formulation does not influence its choices.

8.3 Evaluation of SkinnerDB Features

Next, we analyze the features that contribute to SkinnerDB's performance. In particular, we test the following hypothesis:

Hypothesis 5. *Reinforcement learning-based join ordering is more important for SkinnerDB's performance than other features such as indexing before runtime or parallel filtering.*

We focus on the IMDB-S join order benchmark. First, we consider variations of the learning-based optimizer. Figure 16 compares the previously proposed reward functions (we report arithmetic average values of three runs for runtime and the number of intermediate result tuples generated). Note that we consider the full space of join orders, including Cartesian product joins. Clearly, the reward function has significant impact on performance. Overall, rewarding generated result tuples works better than rewarding processed input. On the other side, we observed sub-optimal performance for several queries with small join output cardinality (in particular for query 10c, increasing from one to five seconds when switching from input to output reward). Using a linear combination of the two reward metrics yields best performance.

Next, we study the impact of join order learning. We compare the UCT-based optimizer against an approach that selects join orders randomly. Figure 17 compares both strategies in terms of execution time and number of generated tuples. Clearly, join order learning is crucial to obtain competitive performance. The randomized approach is not able to finish within five minutes and generates eight times as many tuples until the timeout. This shows that join order learning is fundamental for the performance of SkinnerDB.

We now turn to the pre-processing phase. Figure 18 reports the performance impact of indices. We report total runtime as well as pre-processing time. As discussed in more detail next, the

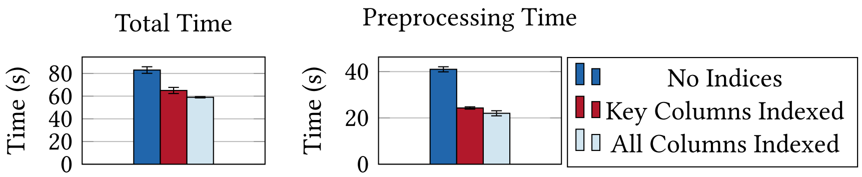


Fig. 18. Impact of indexing (IMDB-S).

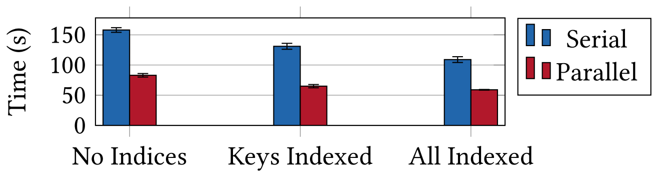


Fig. 19. Impact of parallelization on pre-processing performance for different index configurations (IMDB-S).

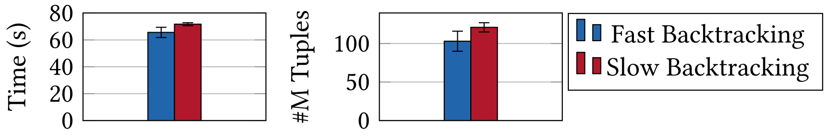


Fig. 20. Impact of fast backtracking (IMDB-S).

pre-processing phase of SkinnerDB is the only one to benefit from previously generated indices. We consider three scenarios: either no indices are created before query execution starts, or indices are available on key and foreign key columns, or indices are available on all columns.

Indices are exploited in two separate ways during pre-processing. First, indices can be used to evaluate unary predicates faster. Second, having indices already available reduces overheads, as we do not need to create them during pre-processing. The latter point requires further explanation. SkinnerDB must be able to efficiently execute arbitrary join orders during the join phase. This becomes only possible by creating in-memory hash indices on all join columns. Those indices need to be created during pre-processing (i.e., at runtime) if they are not already available.

Figure 18 shows that pre-processing time decreases significantly when creating indices on key and foreign key columns before runtime. If such indices are available for base tables, then we can avoid creating them for tables without unary predicates. SkinnerDB will still create indices for tables with unary predicates after filtering. However, typically, the filtered tables are significantly smaller than base tables. Hence, index creation is much faster. Pre-processing time decreases again if indices are available for all columns (this reduces overheads for filtering via unary predicates). However, the effect is less pronounced than for indices on join columns.

SkinnerDB exploits parallelism only during pre-processing in the current version (we are currently working on parallelizing the multi-way join algorithm, too). SkinnerDB parallelizes the evaluation of unary predicates as well as the creation of indices on join columns. The implementation uses Java parallel streams and is not yet highly optimized. Figure 19 reports performance gains via parallelism (31 is the degree of parallelism), depending on the set of previously created indices. Clearly, performance gains via parallelization are higher if fewer indices are initially available.

We introduced fast backtracking as an extension to our multi-way join algorithm. Fast backtracking enables us to skip tuples guaranteed not to lead to join results. It has the potential to reduce the total number of intermediate result tuples generated and processed. Figure 20 measures the impact of fast backtracking. The number of intermediate result tuples decreases by nearly 20%.

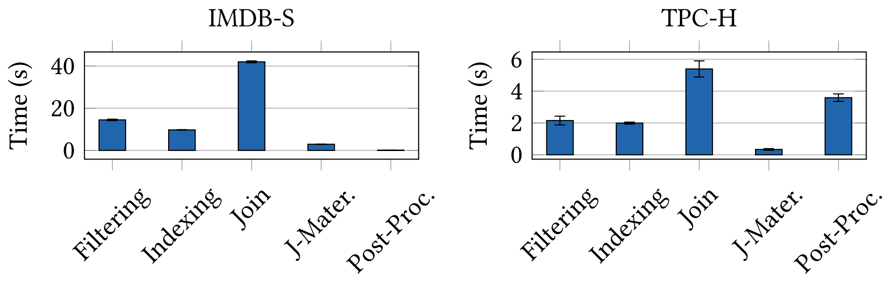


Fig. 21. Time breakdown for IMDB-S and TPC-H benchmark.

Accordingly, runtime decreases by several seconds. Note that fast backtracking only influences the join time. The relative reduction in execution time is therefore smaller than the relative reduction in the number of tuples.

Compared to features like parallel filtering, indexing, complex reward functions, and fast backtracking, we find that join order learning is most critical to SkinnerDB's performance (Hypothesis 5).

8.4 Further Analysis

We "zoom in" and analyze which query or benchmark properties influence SkinnerDB's performance. We also integrate new metrics, besides time and result sizes, to gain a deeper understanding of how SkinnerDB optimizes queries. In particular, we verify the following hypotheses:

Hypothesis 6. *Relative convergence overheads to good join orders decrease for long running queries.*

Hypothesis 7. *The relative overhead of convergence to good join orders increases for larger queries (measured by the number of joined tables).*

Figure 21 breaks down execution time into different phases, executed by SkinnerDB for each sub-query. SkinnerDB's pre-processing phase consists of two steps: filtering and indexing. Filtering refers to the evaluation of unary predicates as well as the creation and materialization of filtered tables (projected on columns that are required for the following steps). Indexing refers to the creation of all missing indices on join columns. The results in Figure 21 refer to the case where indices on key and foreign key columns of base tables were created at pre-processing time. During the join phase, we learn near-optimal join orders via the UCT algorithm and execute them via the specialized multi-way join algorithm. At the end of the join phase, result tuples are represented as vectors referencing tuple indices in base tables. We materialize them into corresponding tables before post-processing starts ("Join materialization"). Finally, we perform grouping and aggregation in the post-processing phase.

For the IMDB benchmark, join processing overheads are dominant. In part, this is due to the fact that pre-processing is parallelized while the join phase is not. During pre-processing, overheads for applying unary predicates dominate. The post-processing overheads are negligible. The picture changes for the TPC-H benchmark. Here, post-processing overheads account for nearly one-third of total processing time. Queries from the TPC-H benchmark feature more complex aggregates and require grouping (while the IMDB benchmark does not). In contrast to the IMDB benchmark, indexing time dominates filtering during the pre-processing phase. This is to be expected, as the IMDB benchmark features more expensive unary predicates such as LIKE expressions. The overheads for join result materialization are minor in both cases.

Table 2. Comparison of Search Space for Join Ordering

	IMDB-S		TPC-H	
Metric	**Average**	**Maximum**	**Average**	**Maximum**
# UCT Nodes	211	4,779	18	64
# Plans	143	1,029	6	17

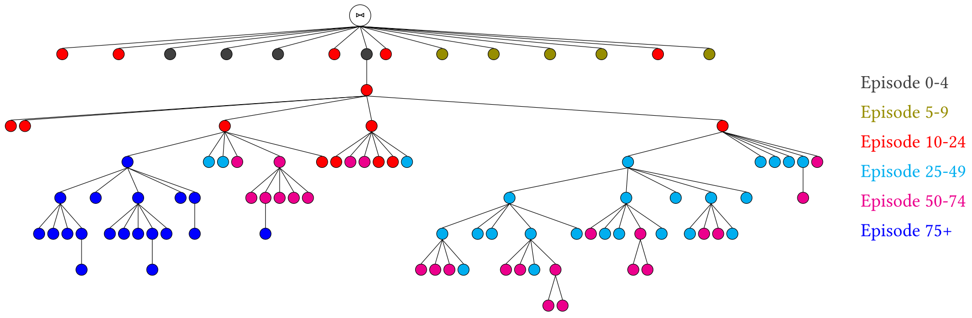


Fig. 22. Illustration of UCT growth over start of processing of IMDB-S query 33c: After random initial exploration, the tree is expanded towards join orders that seem most interesting. The node color indicates in which episode the corresponding node was added.

Table 2 focuses on the join phase and reports metrics related to the search space size. Clearly, the IMDB benchmark is more challenging from the join ordering perspective. This is to be expected, as the IMDB queries feature more tables. Figure 22 illustrates the growth of the UCT tree over the course of a query (here: query 33c of the IMDB-S benchmark). After random expansions in the initial episodes, tree growth soon focuses on the area of the search space that seem interesting. In each iteration, we expand the tree by at most one node along of the path of the join order selected by the UCT algorithm. The targeted growth explains why the number of explored plans remains far below the theoretical maximum (e.g., IMDB-S features multiple queries joining up to 17 tables). A video animation, showing UCT tree growth for another IMDB-S query, is available online.[5]

Next, we analyze the convergence behavior of SkinnerDB. First, we analyze how average reward and the number of explored plans evolves over a query execution. We partition queries into two groups: short running queries, which need up to 100 episodes for execution, and long running queries. To make different queries comparable, we scale per-episode rewards of each query and query group to the interval [0,1]. Also, we scale the number of join orders explored so far to the final number for each query. We divide the execution of each query into five equal parts (based on the number of episodes) and report averages over all queries and for each execution phase in Figure 23. The number of plans explored increases monotonically over the query execution. Similarly, the average reward per episode tends to increase over the execution of a query. This shows that SkinnerDB is successful at finding better join orders over time. At the same time, the number of explored plans grows slower as execution proceeds. This shows that join order choices converge over time. There is a clear distinction between the two query groups. For fast queries, new plans are discovered with a similar frequency over the entire execution. At the same time, reward is significantly below its peak for most of the execution. On the other side, for long running queries, reward is close to its peak over the entire query execution. Also, the frequency at which new plans are discovered keeps flattening as execution proceeds.

[5]https://youtu.be/u4jzEmF2q9U.

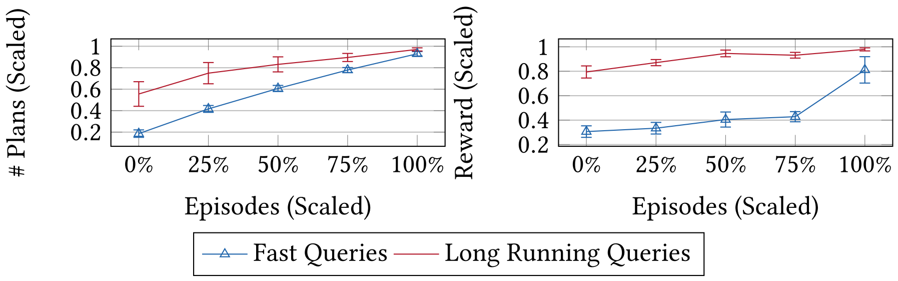


Fig. 23. Analyzing convergence of SkinnerDB to join orders for long running and short queries of the IMDB-S benchmark. For long running queries, reward remains close to its peak for most of the query execution time.

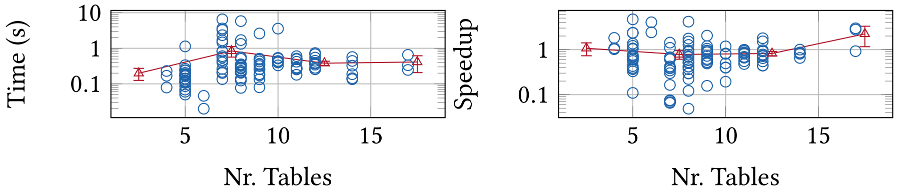


Fig. 24. Skinner-C time (left plot) and speedups, compared to MonetDB, (right plot) on IMDB-S benchmark.

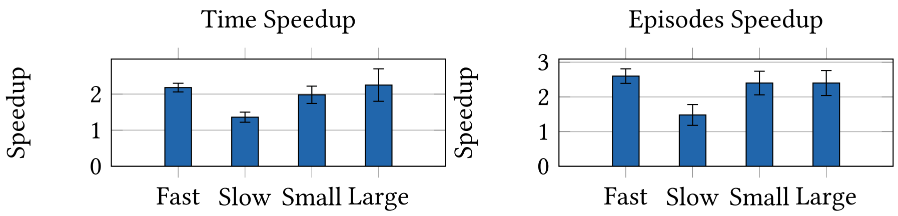


Fig. 25. Speedups by using final join order of prior run for each query for different query classes.

If convergence time is a significant time component, then we may see correlation between performance of SkinnerDB and the number of joined tables (which determines the search space for join ordering). Figure 24 shows runtime and performance (single queries in blue, averages with 90% confidence intervals in red). Based on the results, there does not appear to be a strong performance degradation with growing numbers of query tables.

Figure 25 quantifies overheads of finding good join orders for different query classes. We run each query of the IMDB-S benchmark twice. The second run uses a fixed join order: the join order that the first run converged to. In doing so, we quantify the performance penalty of join order search. We measure the relative speedup of the second run compared to the first. To reduce noise, we only consider time of the join phase when calculating speedups (as the other phases are not influenced by the availability of a join order). We report speedups according to actual runtime (left plot) and according to the number of episodes (right plot). We report arithmetic average speedups (and 90% confidence intervals) obtained from three executions. We break results down by query classes, distinguishing slow (first run takes more than 500 ms) from fast (less than 500 ms) queries and small (joining at most 10 tables) from large (joining more than 10 tables) queries. We hypothesize (Hypothesis 6) that the value of given join orders should increase for fast queries (where convergence time may constitute a higher percentage of total execution time). Also, we assume the same and for large queries (Hypothesis 7), since finding good orders within a larger search space may take longer. The results are consistent with Hypothesis 6, while the results are unclear with regards to Hypothesis 7. In total, the join phase took on average 42 seconds without fixed join order and 38 seconds with a fixed join order.

8.5 Comparison of Learning Algorithms

We conducted experiments comparing the performance of UCT against newer reinforcement learning algorithms. In particular, we verify the following hypotheses:

Hypothesis 8. *Replacing UCT by more recent learning algorithms leads to further speedups.*

Hypothesis 9. *The performance of the UCT optimizer is sensitive to reward scaling.*

Hypothesis 10. *Using parameterless learning algorithms increases robustness to reward scaling.*

So far, we used the UCT algorithm to learn optimal join orders. Now, we compare against two newer reinforcement learning algorithms: BRUE [**?**] and BRUE*I* [17]. Both algorithms belong to the family of Monte-Carlo Tree Search methods as well. Hence, we can use the same search space and the same reward function as for the UCT algorithm. We choose to compare against those algorithms, as they improve over the formal convergence guarantees offered by the UCT algorithm. Also, they reduce the number of tuning parameters. Next, we describe those advantages in more detail (Domshlak and Feldman discuss the differences in full detail [17]). First, the UCT algorithm is aimed at minimizing cumulative regret, while BRUE and BRUE*I* (primarily) minimize simple regret. Simple regret focuses on the gap from expected to optimal reward in the last episode. Cumulative reward focuses on the gap between expected and optimal reward sum over all episodes instead. BRUE and BRUE*I* provide an exponential-rate reduction of simple regret over time (as opposed to a polynomial rate for UCT). Second, BRUE and BRUE*I* are parameter-free. While the formal guarantees of UCT apply for a default setting of the exploration weight, its empirical performance is often dependent on tuning that parameter to a specific scenario [17]. BRUE*I* is a variant of BRUE that aims at finding *good* solutions faster (while both algorithms offer the same convergence guarantees with regards to finding the *optimal* solution). Both, BRUE and BRUE*I*, gradually expand a search tree, similar to the UCT algorithm. They differ by the way in which the tree is expanded (BRUE*I* maintains a connected tree, while BRUE starts with a forest of trees that is gradually connected over time).

So far, we have not separated planning and execution phases. This is appropriate for the UCT algorithm, which is aimed at minimizing cumulative regret. It will carefully balance the overheads of choosing a potentially suboptimal join order in the current iteration versus the information gain (which may lead to speedups in future iterations). BRUE and BRUE*I* are aimed at minimizing simple regret. They will explore the search space more aggressively, disregarding overheads in the current iteration if more information can be gained. To use them effectively, we must separate execution and optimization again. We therefore alternate between sampling for optimization and executing the maximum reward join order for a fixed number of steps. We use the same (depth-first) join algorithm for both phases and share progress among all phases.

The exploration weight determines the balance between exploration and exploitation for the UCT algorithm. It balances a term that depends on reward values with one that does not. Hence, intuitively, the optimal setting for the exploration weight may depend on the reward distribution. That distribution depends in turn on properties of queries and data. For all experiments presented so far, we use the same setting for the exploration factor. This shows that there are settings that work well across a variety of queries and benchmarks. On the other side, there may be extreme cases in which the default setting stops working. BRUE and BRUE*I* do not use any parameters. Their performance should not depend on the reward distribution at all.

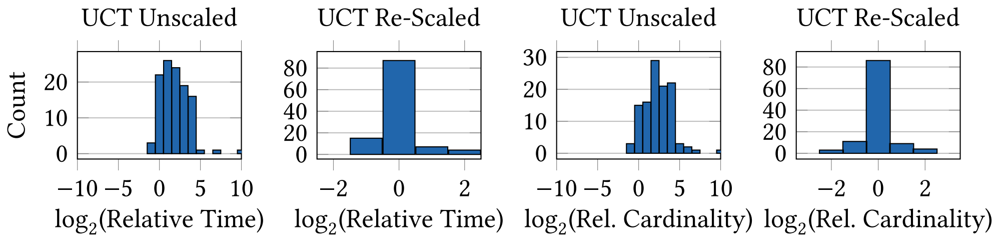


Fig. 26. We compare query execution time and intermediate result cardinality with and without reward scaling for IMDB-S (we report the number of queries for which the performance ratio falls into specific bins). Without rescaling the exploration factor, UCT suffers from higher execution times (see left) and produces more tuples (see right). After rescaling the exploration factor for lower rewards, runtimes and result sizes are comparable to the corresponding metrics before reward scaling.

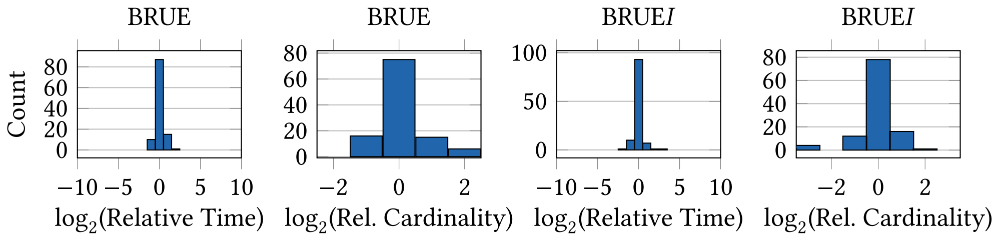


Fig. 27. We compare performance of BRUE (left) and BRUE*I* (right) before and after reward scaling (we report the number of queries where the ratio between performance before and after scaling falls into certain bins). In terms of execution time and in terms of intermediate result sizes, the performance remains stable.

In the following, we study two questions. First, do BRUE or BRUE*I* translate their improved convergence guarantees into better empirical performance? Second, are BRUE and BRUE*I* more robust towards drastic changes of the reward distribution? We answer them using two benchmarks. First, we use IMDB-S in its original configuration. Second, we use IMDB-S but scale down all rewards by factor 10^{-15} (i.e., by 15 orders of magnitude). This simulates an extreme change of data or query properties. For instance, it corresponds to an extreme change in the number of join result tuples produced (which our reward function is in part based upon). Besides BRUE and BRUE*I*, we use the UCT algorithm in two variants. First, we use UCT with the default exploration factor (10^{-5}). Second, we use UCT with an exploration factor of 10^{-20} (scaled down proportionally to the reward scaling).

We analyze robustness of different algorithms to reward scaling. Figure 26 shows results with regards to robustness of the UCT algorithm. We consider the IMDB-S benchmark and scale down all rewards by a factor of $R = 10^{-15}$. We consider the UCT algorithm in two different configurations. For the first configuration ("UCT Unscaled" in Figure 26), we use the original UCT exploration factor of $W = 10^{=5}$. For the second configuration ("UCT Re-Scaled" in Figure 26), we scale down the exploration factor to $W = 10^{-20}$ (proportional to reward scaling). For all queries, we divide execution time or intermediate result cardinality values (i.e., cost according to the C_{out} metric, shown in the right plot) before and after reward scaling. We partition queries based on the logarithm of that ratio and report the number of queries per partition in Figure 26. Clearly, extreme scaling of reward values affects the performance of the UCT algorithm (validating Hypothesis 9). On the other side, performance returns back to normal once the exploration factor is scaled down as well.

Figure 27 shows results for BRUE and BRUE*I*. Again, we compare the performance before and after reward scaling. We report relative per-query performance, i.e., the ratio of performance after

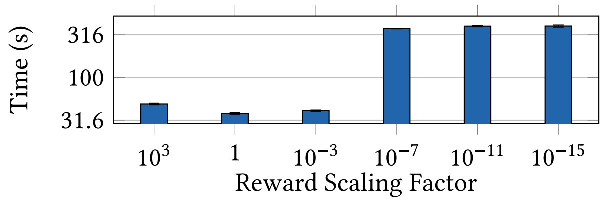


Fig. 28. Sensitivity of UCT algorithm to different reward scales.

and before reward scaling (performance by BRUE and BRUE*I*). Again, we partition queries based on the logarithm of the corresponding ratios and report the number of queries per bin. Here, there are no parameters that relate to the reward distribution. Hence, we only report results for one configuration. Clearly, scaling down rewards has significantly less performance impact than before (verifying Hypothesis 10).

When selecting join orders, the UCT algorithm maximizes the sum of exploration and exploitation terms. If reward is scaled down extremely, then exploration dominates. The choices made by the UCT algorithm start to resemble the ones made by random sampling. On the other side, scaling down the exploration term accordingly reverses the effect. For BRUE and BRUE*I*, scaling reward values has essentially no effect. The decisions made by both algorithms do not depend on absolute reward values. Instead, both algorithms rank actions based on reward averages. Scaling down reward values does not change that ranking.

So far, we have considered an extreme case by scaling down rewards by 15 orders of magnitude. Next, we consider scaling steps in between to identify the point at which performance is significantly affected. Figure 28 reports UCT join time (for the IMDB-S benchmark) when scaling reward by the factor specified in the upper row (we report the arithmetic average of three runs, together with 90% confidence intervals). The performance remains relatively stable (overheads of below 30%) when scaling rewards from 10^3 down to 10^{-3}. Hence, there is a range of reward values encompassing at least six orders of magnitude in which the default parameter setting works reasonably well. Performance decreases significantly starting from a reward scaling of factor 10^{-7}.

In summary, we find that extreme reward scaling may increase join time on IMDB-S from 41 seconds (left-most bar in Figure 28) to 384 seconds (right-most bar in Figure 28) for the UCT algorithm. BRUE and BRUE*I* are not sensitive to such changes. This confirms Hypothesis 10. On the other side, their absolute runtimes are inferior to a reasonably tuned UCT algorithm (204 seconds for BRUE and 224 seconds for BRUE*I*). This invalidates Hypothesis 8. Also, the same parameter settings for UCT work well across a relatively broad range of scenarios (see Figure 28). Overall, UCT remains our method of choice for typical cases.

8.6 Learning with Generic Execution Engines

So far, we focused on our main variant, Skinner-C, that uses a tailored execution engine. Next, we instantiate intra-query learning on top of a traditional execution engine (Skinner-G and Skinner-H). We use Postgres as execution engine and test, in particular, the following hypotheses:

Hypothesis 11. *Skinner-G can outperform untuned systems in special cases.*

Hypothesis 12. *Combining traditional optimization with reinforcement learning (Skinner-H) can improve performance, compared to pure learning.*

Hypothesis 13. *The specialized execution engine used by Skinner-C improves performance significantly, compared to execution on top of generic engines (Skinner-G and Skinner-H).*

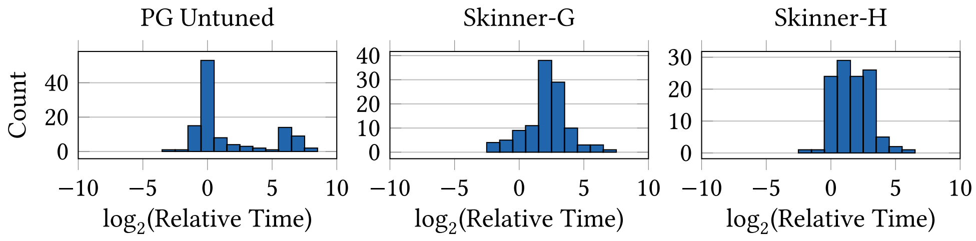


Fig. 29. We compare performance of different approaches on IMDB-S against performance of a manually tuned Postgres installation. Compared to an untuned installation, using Skinner-G or Skinner-H reduces the number of outlier queries where execution time is significantly above the fine-tuned version.

Table 3. Performance of SkinnerDB on Postgres with a Timeout of Five Minutes per Query

System	Time (s)	# Timeouts
Postgres - No Tuning	6,493	16
SkinnerG(PG)	2,049	1
SkinnerH(PG)	1,539	0
Postgres with Tuning	297	0

We use the IMDB (IMDB-S) benchmark for our comparisons. We evaluate Skinner-G and Skinner-H against the original Postgres system. The authors of the IMDB benchmark noticed that Postgres performs better when manually disabling nested loop joins [25]. The reason are unreliable cardinality estimates motivating plans with nested loop joins (that turn out to be highly suboptimal). Disabling this operator is straightforward. However, tracing back performance problems to this tuning option is difficult. It requires intimate knowledge of the query optimizer and processing engine as well as manual analysis of query plans. Many lay users would be incapable of identifying the right tuning knob for optimal performance. This is why we evaluate Postgres in two modes: with and without manual tuning (i.e., with and without nested loop joins).

In Figure 29, we use the performance of Postgres with tuning as baseline. For each query, we analyze relative execution time by comparison to that baseline. We use a per-query timeout of five minutes. For Postgres without tuning, Skinner-G, and Skinner-H, we partition queries based on relative performance (more precisely, we use the rounded logarithm with base two for partitioning). Table 3 summarizes runtimes and timeouts. The following trends become apparent: First, with the default settings, the Postgres optimizer chooses highly sub-optimal query plans for a few queries. This corroborates prior results. Second, Postgres outperforms both SkinnerDB variants with an optimal configuration. On the other side, both SkinnerDB variants are able to improve performance compared to the Postgres out-of-box configuration (validating Hypothesis 11). Finally, both SkinnerDB variants executing on top of Postgres are significantly slower than our main variant, using a specialized execution engine (verifying Hypothesis 13).

Execution engines such as Postgres are not optimized for fast join order switching. This shows clearly when comparing the performance of Skinner-G and Skinner-H against Postgres (tuned) as well as Skinner-C. On the other side, Skinner-G and Skinner-H are able to improve performance in pathological cases where the optimizer chooses highly sub-optimal plans. Note that the performance problems of Postgres are mostly due to the selection of sub-optimal join operators. Neither Skinner-G nor Skinner-H directly control the selection of join operators. However, sub-optimal operator choices translate into bad performance for corresponding join orders. While the

SkinnerDB variants cannot influence operator choices, they can select join orders for which the original optimizer selects good operators. Skinner-H performs slightly better than Skinner-G (validating Hypothesis 12). This is due to the fact that Skinner-H avoids overheads by batch-wise processing if the original optimizer produces plans that finish quickly enough (before learning is invoked).

Skinner-G performs several pre-processing steps in which it creates indices on join columns and clusters data according to the batch ID column (see online appendix for details). Also, our current implementation materializes join results and performs post-processing on the materialized data. We performed one last experiment in which we replaced Skinner-G's join order learning component by the Postgres optimizer without hand-tuning (i.e., we invoke the original Postgres optimizer on tables resulting from Skinner-G's pre-processing phase to generate the join result required for Skinner-G's post-processing stage). We measured a runtime of 19,816 seconds with 48 timeouts. Clearly, the overheads required for enabling intra-query learning on generic execution engines are significant. They do not pay off if combined with standard query optimization.

In summary, while Skinner-G and Skinner-H can help in extreme cases for runtime optimization, we believe that they may be most useful as offline analysis tools. They can be used to uncover alternative join orders for queries with performance problems.

8.7 Summary

We can roughly classify the benchmarks we used into easy (TPC-H, JCC-H without skew), moderately difficult (IMDB-S, IMDB-L), and very difficult (IMDB-L-UDF, JCC-H with skew) cases from the optimizer perspective. SkinnerDB pays for overheads due to learning and join order switching in cases where query optimization is easy. Here, systems such as MonetDB are preferable. However, SkinnerDB never performed more than four times worse than the best system across all benchmarks. This is remarkable, especially given that we have not yet fully parallelized our processing engine (i.e., the join phase is sequential). For benchmarks where optimization is hard, SkinnerDB proves to be the most reliable alternative.

9 CONCLUSION AND OUTLOOK

We introduce SkinnerDB, a database system that uses reinforcement learning to find near-optimal join orders. Under simplifying assumptions, we show that expected performance regret, compared to optimal join order choices for SkinnerDB, is bounded ("regret-bounded query evaluation"). In our experiments, we find the following: First, regret-bounded query evaluation leads to robust performance even for difficult queries, given enough data to process. Second, performance gains by robust join ordering can outweigh learning overheads for benchmarks where optimization is difficult. Third, to realize the full potential of our approach, an (in-query) learning-based optimizer must be paired with a specialized execution engine.

We see multiple avenues for future work. First, the join algorithms presented in this article are sequential. To leverage emerging many-core hardware architectures, we must extend them to leverage parallelism. Second, we are investigating methods to combine inter- with intra-query learning. For instance, we could use knowledge gained from previous queries as initial priors in intra-query learning. A simple variant could select join orders according to a metric that combines UCT upper confidence values with a quality estimate based on past queries. As query execution time increases, we can decrease weight of the component that is not based on the current query. Ideally, this enables us to find optimal join orders for long running queries. For short running queries, it may help to find a reasonable join order faster. Finally, we plan to explore options for finding optimal join orders for data subsets, rather than the entire dataset as a whole.

REFERENCES

[1] A. Aboulnaga, P. Haas, M. Kandil, S. Lightstone, G. Lohman, V. Markl, I. Popivanov, and V. Raman. 2004. Automated statistics collection in DB2 UDB. In *PVLDB*. 1169–1180. DOI: https://doi.org/10.1145/1066157.1066293

[2] Mert Akdere and Ugur Cetintemel. 2011. Learning-based query performance modeling and predection. In *ICDE*. 390–401. DOI: ftp://ftp.cs.brown.edu/pub/techreports/11/cs11-01.pdf

[3] Khaled Hamed Alyoubi. 2016. *Database Query Optimisation Based on Measures of Regret*. Ph.D. Dissertation. Birkbeck University of London.

[4] Khaled H. Alyoubi, Sven Helmer, and Peter T. Wood. 2015. Ordering selection operators under partial ignorance. In *CIKM*. 1521–1530. DOI: https://doi.org/10.1145/2806416.2806446

[5] Ron Avnur and J. M. Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *SIGMOD*. 261–272. DOI: https://doi.org/10.1145/342009.335420

[6] Brian Babcock and S. Chaudhuri. 2005. Towards a robust query optimizer: A principled and practical approach. In *SIGMOD*. 119–130. Retrieved from http://dl.acm.org/citation.cfm?id=1066172.

[7] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *SIGMOD*. 107–118. DOI: https://doi.org/10.1145/1066157.1066171

[8] Peter Boncz, Angelos Christos Anatiotis, and Steffen Kläbe. 2018. JCC-H: Adding join crossing correlations with skew to TPC-H. *LNCS* 10661 (2018), 103–119. DOI: https://doi.org/10.1007/978-3-319-72401-0_8

[9] Nicolas Bruno and Surajit Chaudhuri. 2002. Exploiting statistics on query expressions for optimization. In *SIGMOD*. 263–274. DOI: https://doi.org/10.1145/564720.564722

[10] Surajit Chaudhuri and Vivek Narasayya. 2001. Automating statistics management for query optimizers. In *ICDE*. 7–20. DOI: https://doi.org/10.1109/69.908978

[11] Sophie Cluet and Guido Moerkotte. 1995. On the complexity of generating optimal left-deep processing trees with cross products. In *ICDT*. 54–67. DOI: http://link.springer.com/chapter/10.1007/3-540-58907-4{_}6 .

[12] Anne Condon, Amol Deshpande, Lisa Hellerstein, and Ning Wu. 2009. Algorithms for distributional and adversarial pipelined filter ordering problems. *ACM Trans. Algor.* 5, 2 (2009), 1–34. DOI: https://doi.org/10.1145/1497290.1497300

[13] Pierre-Arnaud Coquelin and Rémi Munos. 2007. Bandit algorithms for tree search. In *Uncertainty in Artificial Intelligence*. AUAI Press, 67–74. arXiv:arXiv:cs/0703062v1.

[14] D. Harish, Pooja N. Darera, and Jayant R. Haritsa. 2008. Identifying robust plans through plan diagram reduction. *PVLDB* 1, 1 (2008), 1124–1140. Retrieved from http://dl.acm.org/citation.cfm?id=1453976.

[15] Amol Deshpande. 2004. An initial study of overheads of eddies. *SIGMOD Rec.* 33, 1 (2004), 44–49. DOI: https://doi.org/10.1145/974121.974129

[16] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2006. Adaptive query processing. *Found. Trends Datab.* 1, 1 (2006), 1–140. DOI: https://doi.org/10.1561/1900000001

[17] Carmel Domshlak and Zohar Feldman. 2013. To UCT, or not to UCT? In *SoCS*. 1–8. DOI: http://www.aaai.org/ocs/index.php/SOCS/SOCS13/paper/view/7268

[18] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *SIGMOD*. 337–348. DOI: https://doi.org/10.1145/1989323.1989359

[19] Anshuman Dutt. 2014. QUEST: An exploratory approach to robust query processing. *PVLDB* 7, 13 (2014), 5–8.

[20] Anshuman Dutt and Jayant Haritsa. 2014. Plan bouquets: Query processing without selectivity estimation. In *SIGMOD*. 1039–1050. DOI: https://doi.org/10.1145/2588555.2588566

[21] Amr El-Helw, Ihab F. Ilyas, and Calisto Zuzarte. 2009. StatAdvisor: Recommending statistical views. *PVLDB* 2, 2 (2009), 1306–1317. DOI: http://www.vldb.org/pvldb/2/vldb09-525.pdf

[22] Zohar Feldman and Carmel Domshlak. 2014. Simple regret optimization in online planning for Markov decision processes. *J. Artif. Intell. Res.* 51 (2014), 165–205. DOI: https://doi.org/10.1613/jair.4432

[23] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. 2009. Predicting multiple metrics for queries—Better decisions enabled by machine learning. In *ICDE*. 592–603.

[24] Sylvain Gelly, L. Kocsis, and Marc Schoenauer. 2012. The grand challenge of computer go: Monte Carlo tree search and extensions. *Commun. ACM* 3 (2012), 106–113. DOI: http://dl.acm.org/citation.cfm?id=2093574

[25] Andrey Gubichev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015), 204–215.

[26] P. J. Haas and A. N. Swami. 2011. Sampling-based selectivity estimation for joins using augmented frequent value statistics. In *ICDE*. 522–531. DOI: https://doi.org/10.1109/ICDE.1995.380361

[27] Peter J. Haas and Arun N. Swami. 1992. Sequential sampling procedures for query size estimation. *SIGMOD Rec.* 21, 2 (1992), 341–350. DOI: https://doi.org/10.1145/141484.130335

[28] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. DeepDB: Learn from data, not from queries! *PVLDB* 13, 7 (2019), 992–1005. DOI: https://doi.org/10.14778/3384345.3384349

[29] Konstantinos Karanasos, Andrey Balmin, Marcel Kutsch, Fatma Ozcan, Vuk Ercegovac, Chunyang Xia, and Jesse Jackson. 2014. Dynamically optimizing queries over large scale data platforms. In *SIGMOD*. 943–954. DOI: https://doi.org/10.1145/2588555.2610531

[30] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*. Retrieved from http://arxiv.org/abs/1809.00677.

[31] Levente Kocsis and C. Szepesvári. 2006. Bandit based monte-carlo planning. In *ECML*. 282–293. DOI: http://www.springerlink.com/index/D232253353517276.pdf

[32] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. Retrieved from http://arxiv.org/abs/1808.03196.

[33] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2018. QTune: A queryaware database tuning system with deep reinforcement learning. *PVLDB* 12, 12 (2018), 2118–2130. DOI: https://doi.org/10.14778/3352063.3352129

[34] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. Robust estimation of resource consumption for SQL queries using statistical techniques. *PVLDB* 5, 11 (2012), 1555–1566. Retrieved from http://dl.acm.org/citation.cfm?id=2350229.2350269.

[35] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. 1990. Practical selectivity estimation through adaptive sampling. In *SIGMOD*. 1–11. DOI: https://doi.org/10.1145/93605.93611

[36] Guy Lohman. 2014. Is query optimization a "solved" problem? *SIGMOD Blog* (2014). https://wp.sigmod.org/?p=1075

[37] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2018. Neo: A learned query optimizer. *PVLDB* 12, 11 (2018), 1705–1718. DOI: https://doi.org/10.14778/3342263.3342644

[38] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *aiDM*. 3. Retrieved from arXiv:arXiv:1803.00055v2.

[39] Thomas Neumann and Cesar Galindo-Legaria. 2013. Taking the edge off cardinality estimation errors using incremental execution. In *BTW*. 73–92.

[40] Thomas Neumann and Alfons Kemper. 2015. Unnesting arbitrary queries. In *BTW*. 383–402. DOI: http://www.btw-2015.de/res/proceedings/Hauptband/Wiss/Neumann-Unnesting{_}Arbitrary{_}Querie.pdf.

[41] Hung Q. Ngo, Ely Porat, and Christopher Ré. 2012. Worst-case optimal join algorithms. In *PODS*. 37–48.

[42] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2019. An empirical analysis of deep learning for cardinality estimation. Retrieved from http://arxiv.org/abs/1905.06425.

[43] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2020. QuickSel: Quick selectivity learning with mixture models. In *SIGMOD*. 1017–1033. DOI: https://doi.org/10.1145/3318464.3389727

[44] Adrian Daniel Popescu, Andrey Balmin, Vuk Ercegovac, and Anastasia Ailamaki. 2013. PREDIcT: Towards predicting the runtime of large scale iterative analytics. *PVLDB* 6, 14 (2013), 1678–1689. DOI: https://doi.org/10.14778/2556549.2556553

[45] Li Quanzhong, Shao Minglong, Volker Markl, Kevin Beyer, Latha Colby, and Guy Lohman. 2007. Adaptively reordering joins during query execution. In *ICDE*. 26–35. DOI: https://doi.org/10.1109/ICDE.2007.367848

[46] Vijayshankar Raman, A. Deshpande, and J. M. Hellerstein. 2003. Using state modules for adaptive query processing. In *ICDE*. 353–364. DOI: https://doi.org/10.1109/ICDE.2003.1260805

[47] P. G. G. Selinger, M. M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In *SIGMOD*. 23–34. DOI: http://dl.acm.org/citation.cfm?id=582095.582099

[48] David Silver. 2009. *Reinforcement Learning and Simulation-based Search in Computer Go.* Ph.D. Dissertation. University of Alberta.

[49] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning optimizer. In *PVLDB*. VLDB, 19–28. http://www.vldb.org/conf/2001/P019.pdf.

[50] Ji Sun and Guoliang Li. 2020. An end-to-end learning-based cost estimator. In *VLDBJ*, Vol. 13. 307–319. DOI: https://doi.org/10.14778/3368289.3368296

[51] TPC. 2013. TPC-H Benchmark. Retrieved from http://www.tpc.org/tpch/.

[52] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-bounded query evaluation via reinforcement learning. In *SIGMOD*. 1039–1050.

[53] Kostas Tzoumas, Timos Sellis, and Christian S. Jensen. 2008. *A Reinforcement Learning Approach for Adaptive Query Processing*. Technical Report. Aalborg University.

[54] Todd L. Veldhuizen. 2012. Leapfrog Triejoin: A worst-case optimal join algorithm.

[55] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. 2003. Maximizing the output rate of multi-way join queries over streaming information sources. In *PVLDB*. 285–296. Retrieved from http://dl.acm.org/citation.cfm?id=1315451.1315477

[56] Lucas Woltmann, Claudio Hartmann, Maik Thiele, and Dirk Habich. 2019. Cardinality estimation with local deep learning models. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM)*. 1–8.

[57] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. 2016. Sampling-based query re-optimization. In *SIGMOD*. 1721–1736. DOI: http://arxiv.org/abs/1601.05748

[58] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-LSTM for join order selection. In *ICDE*. 1297–1308. DOI: https://doi.org/10.1109/ICDE48307.2020.00116

[59] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *SIGMOD*. 415–432. DOI: https://doi.org/10.1145/3299869.3300085

Received April 2020; revised February 2021; accepted May 2021