ADOPT: Adaptively Optimizing Attribute Orders for Worst-Case Optimal Join Algorithms via Reinforcement Learning

Junxiong Wang
Cornell University
Ithaca, NY, USA
junxiong@cs.cornell.edu

Immanuel Trummer
Cornell University
Ithaca, NY, USA
itrummer@cornell.edu

Ahmet Kara
University of Zurich
Zurich, Switzerland
kara@ifi.uzh.ch

Dan Olteanu
University of Zurich
Zurich, Switzerland
olteanu@ifi.uzh.ch

ABSTRACT

The performance of worst-case optimal join algorithms depends on the order in which the join attributes are processed. Selecting good orders before query execution is hard, due to the large space of possible orders and unreliable execution cost estimates in case of data skew or data correlation. We propose ADOPT, a query engine that combines adaptive query processing with a worst-case optimal join algorithm, which uses an order on the join attributes instead of a join order on relations. ADOPT divides query execution into episodes in which different attribute orders are tried. Based on run time feedback on attribute order performance, ADOPT converges quickly to near-optimal orders. It avoids redundant work across different orders via a novel data structure, keeping track of parts of the join input that have been successfully processed. It selects attribute orders to try via reinforcement learning, balancing the need for exploring new orders with the desire to exploit promising orders. In experiments with various data sets and queries, it outperforms baselines, including commercial and open-source systems using worst-case optimal join algorithms, whenever queries become complex and therefore difficult to optimize.

PVLDB Reference Format:
Junxiong Wang, Immanuel Trummer, Ahmet Kara, and Dan Olteanu. ADOPT: Adaptively Optimizing Attribute Orders for Worst-Case Optimal Join Algorithms via Reinforcement Learning. PVLDB, 16(11): 2805 - 2817, 2023.
doi:10.14778/3611479.3611489

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jxiw/ADOPT.

1 INTRODUCTION

The area of join processing has recently been revolutionized by worst-case optimal join algorithms [29, 45]. LeapFrog TrieJoin (LFTJ) is a prime example of a worst-case optimal join algorithm [45].

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611489

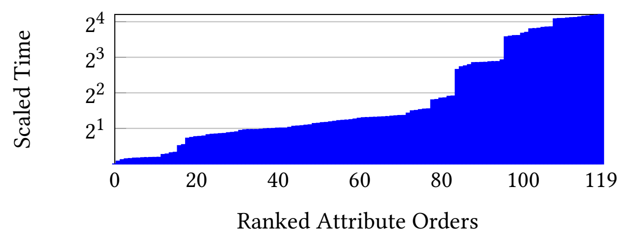

Figure 1: Execution time of different attribute orders for the five-clique query on the ego-Twitter graph.

Such algorithms guarantee asymptotically worst-case optimal performance. Those formal guarantees set them apart from traditional join algorithms, which are known to be sub-optimal [30]. In practice, they often translate into orders of magnitude runtime improvements, specifically for cyclic queries, compared to traditional approaches. They are incorporated in several recent query engines: for factorized databases [32, 33], graph processing [2, 15] and general query processing [11], in-database machine learning [36], and in the commercial systems LogicBlox [4] and RelationalAI [3].

As pointed out in prior work [45], in practice, the performance of worst-case optimal join algorithms often depends heavily on the order in which join attributes (i.e., groups of join columns that are linked by equality constraints) are processed. Yet this is not reflected in the formal analysis of worst-case optimal join algorithms [45]. Worst-case optimality is defined with regards to worst-case assumptions about the database content. Under these assumptions, different attribute orders have asymptotically equivalent time complexity. On the other side, given the actual database content, some attribute orders may perform much better than others in practice. Similar to the classical join ordering problem, it is therefore important to aim for the instance-optimal order, e.g., using data statistics.

Example 1.1. Figure 1 illustrates the need for accurate attribute order selection. It compares LFTJ execution times (scaled to the time of the fastest order) for different attribute orders and the same query that asks for the number of cliques of five distinct nodes. 120 attribute orders (on the x-axis) are ranked by execution time. The performance gap between the best and worst attribute orders is more than 16x. The choice of an attribute order has thus significant impact on performance and near-optimal orders are sparse.

Execution engines using worst-case optimal join operators (e.g., the LogicBlox system [4]) typically select attribute orders via a query optimizer. Similar to traditional query optimizers selecting join orders, such optimizers exploit data statistics and simplifying cost models to pick an attribute order. This approach is however risky. Erroneous cost estimates (e.g., due to data skew not represented in data statistics) can lead to highly sub-optimal attribute order choices. Incorrect cost estimates are known to cause significant overheads in traditional query optimization [23]. The experiments in Section 5 show that this case appears in the context of optimization for worst-case optimal join algorithms as well. In particular, this applies to queries on non-uniform data with an elevated number of predicates, increasing the potential for inter-predicate correlations that make size and cost predictions hard.

Example 1.2. In social network analysis, analysts are often interested in finding people who are mutually connected in cliques via links in the graph representing the social network [17, 19, 34]. Specifically, prior analysis often considers cliques of up to five or six [17, 34], or more [19] members. The experiments in Section 5 show that such queries already create challenges in cost prediction, making methods that are robust to prediction errors preferable.

To overcome these challenges, we propose an adaptive execution strategy for worst-case optimal join algorithms. The goal of adaptive processing is to enable attribute order switches, during query processing. The processing time is divided into episodes and in each episode we may choose an attribute order for the execution of the query over a fragment of the input data. By measuring execution speed for different attribute orders, the adaptive processing framework converges to near-optimal attribute orders over time. To the best of our knowledge, this is the first adaptive processing strategy for worst-case optimal join algorithms.

Adaptive processing for query processing based on attribute orders leads, however, to new challenges, discussed in the following.

First, we must limit overheads due to attribute order switching. In particular, we must avoid redundant processing when applying multiple attribute orders to the same data. We solve this challenge by a task manager, capturing execution progress achieved by different attribute orders. Join result tuples are characterized by a value combination for join attributes. Hence, we generally represent execution progress by (hyper)cubes within the Cartesian product of value ranges over all join attributes. Having processed a cube implies that all contained result tuples, if any, have been generated. Data processing threads query the task manager to retrieve cubes not covered previously. Also, the task manager is updated whenever new results become available. It ensures that different threads process non-overlapping cubes, independently of the current attribute order. Query processing ends once all processed cubes, in aggregate, cover the full space of join attribute value ranges.

Second, we need a metric to compare different attribute orders, based on run time feedback. This metric must be applicable even when executing attribute orders for a very short amount of time. The number of result tuples generated per time unit may appear to be a good candidate metric. However, it is not informative in case of small results. Instead, we opt for a metric analyzing the size of the hypercube (within the Cartesian product of join attribute values) covered per time unit. Even if no result tuples are generated, this metric rewards attribute orders that quickly discard subsets of the output space.

Third, we must choose, in each episode, which attribute order to select next. This choice is challenging as it is subject to the so called *exploration-exploitation dilemma*. Choosing attribute orders that obtained good scores in past invocation (exploitation) may seem beneficial to generate a full query result as quickly as possible. However, executing attribute orders about which little is known (exploration) may be better. It may lead to even better attribute orders that can be selected in future episodes. To balance between these two extremes in a principled manner, we employ methods from the area of reinforcement learning. Under moderately simplifying assumptions, based on the guarantees offered by these methods, we can show that ADOPT converges to optimal attribute orders.

We have integrated our approach for adaptive processing with worst-case optimal join algorithms into ADOPT (ADaptive wOrst-case oPTimal joins), a novel, analytical SQL processing engine. We compare ADOPT to various baselines, including traditional database systems such as PostgreSQL and MonetDB, prior methods for adaptive processing such as SkinnerDB [42], as well as commercial and open-source database engines that use worst-case optimal join algorithms. We evaluate all systems on acyclic and cyclic queries from public benchmarks, TPC-H, JCC-H [8], join order [13] and SNAP graph data [21, 31] workloads. For complex queries on skewed data, ADOPT outperforms all competitors. In particular, it improves over a commercial database engine using the same worst-case optimal join algorithm as ADOPT. This demonstrates the benefit of adaptive attribute order selections.

In summary, the contributions in this paper are the following:

- We propose the first adaptive processing strategy for worst-case optimal join algorithms using reinforcement learning.
- We describe specialized data structures, progress metrics, and learning algorithms that make adaptive processing in this scenario practical.
- We formally analyze worst-case optimality guarantees and convergence properties.
- We compare ADOPT experimentally against various baselines, showing that it outperforms them for a variety of acyclic and cyclic queries and datasets.

The remainder of this paper is organized as follows. Section 2 presents an overview of the ADOPT system. Section 3 describes the algorithm used for adaptive processing in detail. Section 4 analyzes the approach formally while Section 5 reports experimental results. Finally, Section 6 discusses prior related work.

2 OVERVIEW

Figure 2 overviews the ADOPT system, illustrating its primary components. ADOPT supports SPJAG queries with sub-queries, covering the majority of TPC-H queries (see Section 5.1 for details). It performs in-memory data processing and uses a columnar data layout. The highly specific requirements of the ADOPT approach (e.g., support for high-frequency attribute order switching in a worst-case optimal join processing framework) motivate a customized system, rather than the integration into classical SQL execution engines. The implementation uses Java and supports

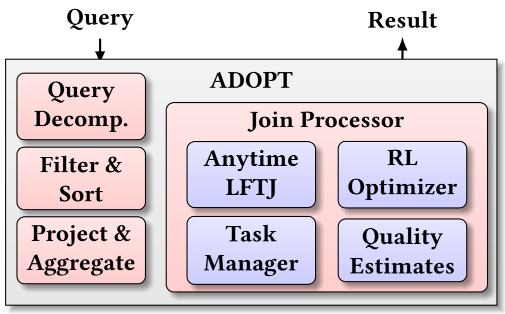

Figure 2: Overview of ADOPT system components.

multi-threading via the Java ExecutorService API. It uses a worst-case optimal algorithm to process joins and selects attribute orders via reinforcement learning.

For complex queries (i.e., queries with sub-queries), ADOPT first decomposes them into a sequence of simple SPJAG queries, using decomposition techniques proposed in prior work [27, 41]. After decomposition, it executes the resulting queries, storing query results in temporary tables that are referenced by later queries in the query sequence (as input tables). For each simple query, ADOPT first performs a pre-processing step to filter the tables using unary predicates from the query (the resulting tables are typically much smaller than the original ones). After that, the following join phase is executed on the filtered tables.

For worst-case optimal computation of equality joins, ADOPT uses LeapFrog TrieJoin (LFTJ). LFTJ considers join attributes in a fixed order to find value combinations that satisfy all join predicates. ADOPT uses an anytime version of this algorithm, so it can suspend and resume execution with high frequency. This enables the adaptive processing strategy, allowing ADOPT to identify near-optimal attribute orders, based on run time feedback. Similar to LFTJ, ADOPT does not materialize intermediate join results: LFTJ stores at most one tuple, containing one value per attribute, as intermediate state and adds complete tuples directly to the join result. This makes suspend and resume operations very efficient. Our technical report [48] provides further details on the original LFTJ algorithm that ADOPT is based upon, including several examples. It also discusses details on the LFTJ variant used for ADOPT, such as how ADOPT uses and maintains data structures enabling the system to perform fast seek operations on the input tables, retrieving tuples that satisfy inequality conditions on their attributes.

Besides the join algorithm itself, ADOPT uses an optimizer based on reinforcement learning. The optimizer selects attribute orders, balancing the need for exploration (i.e., trying out new attribute orders) with the need for exploitation (i.e., trying out attribute orders that performed well in the past). Each selected attribute order is only executed for a limited number of steps, enabling ADOPT to try thousands of attribute orders per second. To compare different attribute orders, ADOPT generates quality estimates. These estimates judge the performance achieved via an attribute order during a single invocation. Performance may vary, for the same order, across different invocations (e.g., due to heterogeneous data distributions). However, by averaging over different invocations for the same attribute order, ADOPT obtains increasingly more precise quality estimates over time.

Switching between attribute orders makes it challenging to avoid redundant work. ADOPT uses a task manager to keep track of remaining parts of the join input to process. More precisely, the task manager manages (hyper)cubes in the Cartesian product space, formed by value ranges of all join attributes. Each cube represents a part of the input space that still has to be processed by some attribute order (i.e., corresponding result tuples, if any, have not been added into a shared result set yet). The execution of the anytime LFTJ is restricted to cubes that have not been processed yet. More precisely, data processing threads query the task manager for cubes, called *target cubes* in the following, that do not overlap with any cubes processed previously or concurrently (by other threads). Threads process the target cube until completion or until reaching the per-episode limit of computational steps. The task manager is notified of processed parts of the target cube (if the step limit is reached, only a subset of the target cube, represented by a small set of cubes contained in the target cube, was processed). The task manager removes processed cubes from the set of remaining cubes.

Join processing terminates once the entire input (i.e., the hypercube representing the full Cartesian product of join attribute values) has been covered. This can be verified efficiently using the task manager. If no unprocessed cubes remain, a complete result has been generated. Depending on the type of query, ADOPT executes a post-processing stage in which group-by clauses and aggregates are executed. Specifically for (count, max, min, sum, and avg) aggregates without grouping, ADOPT integrates join processing with aggregation and does not need to perform a post-processing stage.

Several processing phases of ADOPT can be parallelized. Specifically, ADOPT parallelizes the join preparation phase (i.e., unary predicates are evaluated on different data partitions in general) and sorts data in parallel. During the join phase, ADOPT assigning non-overlapping hypercubes to different threads. Hence, using the same mechanism that avoids redundant work across attribute orders, ADOPT avoids redundant work across different threads as well.

3 ALGORITHM

We discuss the algorithm used by ADOPT in detail. Section 3.1 discusses the top-level function, used to process queries. Section 3.2 introduces ADOPT's parallel anytime join algorithm with worst-case optimality guarantees. Section 3.3 discusses the mechanism by which ADOPT avoids redundant work across different attribute orders. Section 3.4 describes how ADOPT selects attribute orders via reinforcement learning. Finally, Section 3.5 describes the reward metric used to guide the learning algorithm.

3.1 Main Function

ADOPT uses Algorithm 1 to process simple SPJAG queries (i.e., without sub-queries). In addition to the query, the algorithm also takes as input a number of data processing threads and a number of computational steps spent to evaluate a selected attribute order.

First, ADOPT filters the tables with the unary predicates (Line 5). ADOPT supports hash indexes on single columns and uses them, if available, to retrieve rows satisfying unary equality predicates. Without indexes, it scans and filters data, exploiting multi-threading.

Algorithm 1 Main function of ADOPT, processing queries.

```
1: Input: Query q, number of threads n, per-episode budget b
2: Output: Query result
3: function ADOPT(q, n, b)
4:     // Filter input tables via unary predicates
5:     {R_1, ..., R_m} ←Prep.UnaryFilter(q)
6:     // Initialize join result set
7:     R ← ∅
8:     // Initialize reinforcement learning
9:     RL.Init(q)
10:    // Initialize constraint store
11:    TM.Init(q, n)
12:    // Iterate until result is complete
13:    while ¬ TM.Finished do
14:        // Select attribute order via UCT algorithm
15:        o ← RL.Select
16:        // Use order for limited join steps
17:        reward ←AnytimeWCOJ(q, o, n, b, R)
18:        // Update UCT statistics with reward
19:        RL.Update(o, reward)
20:    end while
21:    // Return result after post-processing
22:    return Post(q, R)
23: end function
```

After that, the only remaining predicates are then join predicates (including equality and other join predicates). Next, the algorithm initializes the set of join result tuples, the reinforcement learning algorithm by specifying the search space of attribute orders (which depends on the query), and the task manager with the input query and the number of processing threads (Lines 6 to 11). Internally, the task manager initializes the hypercube representing the total amount of work for each thread. More precisely, it divides the cube, representing the Cartesian product of all join attribute ranges, into equal shares for each thread.

The task manager keeps track of cubes processed by the worker threads. Hence, query processing finishes once all processed cubes, in aggregate, cover the full input space. Iterations continue (Lines 13 to 20) until that termination condition is satisfied. In each iteration, ADOPT first selects an attribute order via reinforcement learning (Line 15). Then, it executes that order, in parallel, for a fixed number of steps (Line 17). By executing the attribute order, the result set (R) may get updated. Note that R only contains complete result tuples (mapping each attribute to a value) or partial values for aggregates. However, it does not contain any intermediate result tuples. Besides updating results, executing an attribute order yields reward values, representing execution progress per time unit. Those reward values are used to update statistics (Line 19), maintained internally by the reinforcement learning optimizer, to guide attribute order selections in future iterations. Once the join finishes, the algorithm performs post-processing (e.g., calculating per-group aggregates for group-by queries, based on join results in R) and returns the result (Line 22).

3.2 Anytime Join Algorithm

Algorithm 2 is the (worst-case optimal) join algorithm, used to execute a given attribute order for a fixed number of steps. Execution proceeds in parallel: different worker threads operate on non-overlapping cubes. Each worker thread iterates the following

Algorithm 2 Parallel anytime version of worst-case optimal join algorithm.

```
1: Input: Query q, attribute order o, number of threads n, per-episode
   budget b, Result set R
2: Output: Reward r
3: function AnytimeWCOJ(q, o, n, b, R)
4:     // Initialize accumulated reward
5:     r ← 0
6:     // Execute in parallel for all threads
7:     for 1 ≤ t ≤ n in parallel do
8:         // Initialize remaining cost budget
9:         l_t ← b
10:        // Iterate until per-episode budget spent
11:        while l_t > 0 do
12:            // Retrieve unprocessed target cube
13:            c_t ←TM.Retrieve
14:            // Process cube until timeout, add results
15:            ⟨P_t, s_t⟩ ←JoinOneCube(q, l_t, o, c_t, R)
16:            // Update constraints via processed cube
17:            TM.Remove(c_t, P_t)
18:            // Update accumulated reward (see Section 3.5)
19:            r ← r + Reward(P_t, q)
20:            // Update remaining budget
21:            l_t ← l_t − s_t
22:        end while
23:    end for
24:    // Return accumulated reward
25:    return r
26: end function
```

steps until its computational budget is depleted (Lines 11 to 22). First, it retrieves an unprocessed cube, the target cube, from the task manager (Line 13). Then, it uses a sub-function (an anytime version of the LFTJ) to process the retrieved target cube (Line 15). In practice, it is often not possible to process the entire target cube under the remaining computation budget. Hence, the result of the triejoin invocation (Function JoinOneCube) reports the set of cubes, contained within the target cube, that were successfully processed. In addition, it returns the number of computation steps spent. The task manager is notified of successfully processed cubes which will be excluded from further consideration (Line 17). Also, a reward value is calculated that represents progress towards generating a full join result (Line 19). We postpone a detailed discussion of the reward function to Section 3.4. Finally, Algorithm 2 returns the reward value, accumulated over all threads and iterations (Line 25).

Algorithm 3 describes the sub-function, used to process a single cube, at a high level of abstraction. The actual join is performed by Procedure JoinOneCubeRec. This procedure is based on the leapfrog triejoin [45], a classical, worst-case optimal join algorithm[1]. For conciseness, the pseudo-code describes the algorithm as a recursive function (whereas the actual implementation does not use recursion). The input to the algorithm is the join query, the remaining computational budget, an attribute order, a target cube to process, the result set, and the index of the current attribute. The algorithm considers query attributes sequentially, in the given attribute order. The attribute index marks the currently considered

[1]A detailed example of the LFTJ execution is given in our technical report [48].

Algorithm 3 Worst-case optimal join algorithm with timeout, joining a single cube.

1: **Input:** Query q, remaining budget b, attribute order o, target cube to process c, result set R, attribute counter a, value mappings M
2: **Effect:** Iterates over attribute values and possibly adds results to R
3: **procedure** JoinOneCubeRec(q, b, o, c, R, a, M)
4: **if** $a \geq |q.A|$ **then** // Check for completed result tuples
5: Insert tuple with current attribute values M into R
6: **else**
7: // Initialize value iterator (do not evaluate it!)
8: $V \leftarrow$ iterator over values for o_a in $[c.l_{o_a}, c.u_{o_a}]$ that satisfy all applicable join predicates in q.
9: // Iterate over values until timeout
10: **for** $v \in V$ **do**
11: // Select values for remaining attributes
12: JoinOneCubeRec($q, l, o, c, R, a+1, M \cup \{\langle o_a, v \rangle\}$)
13: // Check for timeouts
14: **if** Total computational steps $> b$ **then**
15: **Break**
16: **end if**
17: **end for**
18: **end if**
19: **end procedure**

20: **Input:** Query q, remaining budget b, attribute order o, target cube to process c, result set R
21: **Output:** Processed cube p, computational steps performed s
22: **function** JoinOneCube(q, b, o, c, R)
23: // Resume join for fixed number of steps
24: JoinOneCubeRec($q, b, o, c, R, 0, \emptyset$)
25: // Retrieve state from JoinOneCubeRec invocation
26: $s \leftarrow$ Number of computational steps spent
27: $v \leftarrow$ Vector s.t. v_a is last value considered for attribute o_a
28: // Calculate processed cubes
29: $P \leftarrow \emptyset$
30: **for** $0 \leq a < |q.A|$ **do**
31: Create new cube p s.t.
32: $\forall i < a : p_i = [v_i, v_i]$;
33: $p_a = [c.l_{o_a}, v_a)$;
34: $\forall a < i : p_i = [c.l_{o_i}, c.u_{0_i}]$
35: $P \leftarrow P \cup \{p\}$
36: **end for**
37: **return** $\langle P, s \rangle$
38: **end function**

attribute. Once the attribute index reaches the total number of attributes (represented as $q.A$), the algorithm has selected one value for each attribute. Furthermore, at that point, it is clear that the combination of attribute values satisfies all applicable join conditions. Hence, the algorithm adds the corresponding result tuple into the result set (Line 5). As a variant (not shown in Algorithm 3), for queries with simple aggregates without grouping, ADOPT does not store result tuples but merely updates partial aggregate values for each aggregate. If the attribute index is below the total number of attributes, the algorithm iterates over values for that attribute (i.e., attribute o_a where o is the order and a the attribute index) in the loop from Line 10 to 17.

In Line 8, Algorithm 3 creates an iterator over values for the current attribute that satisfy all *applicable* join predicates and are within the target cube, i.e., values contained in the interval $[c.l_{o_a}, c.u_{o_a}]$ for attribute number a within order o ($c.l$ and $c.u$ designate vectors, indexed by attribute, that represent lower and upper target cube bounds respectively). It should be well understood that the algorithm does not assemble the full set of matching values before iterating (as that would create significant overheads when switching attribute orders before being able to try all collected values). Instead, Line 8 is meant to represent the initialization of data structures that allow iterating over matching values efficiently. Join predicates are applicable if, beyond the current attribute o_a, they only refer to attributes whose values have been fixed previously (i.e., a corresponding value assignment is contained in M). For equality join predicates, ADOPT uses the same mechanism as LFTJ [45] to efficiently iterate over satisfying values. This mechanism is described in detail in the extended technical report [48]. It is based on data structures that support fast seek operations on query relations. Whenever required data structures are not available, ADOPT dynamically creates them at run time. For base relations, but not for relations filtered via unary predicates, ADOPT caches and reuses those data structures across queries.

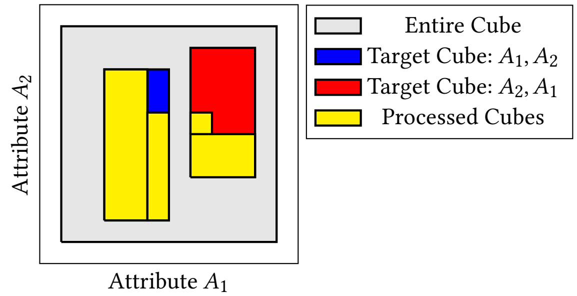

Figure 3: Illustration of containment relationships between hypercubes when processing a query with two attributes.

Join processing via Procedure JoinOneCube terminates once the computational budget is depleted (check in Line 14), or if the current cube is entirely processed. Function JoinOneCube retrieves the number of computational steps, spent during join processing, as well as the last selected value for each attribute. It uses the latter to calculate the set of processed cubes (to be removed from the set of unprocessed cubes). Procedure JoinOneCubeRec does not advance from one value of an attribute to the next, unless all value combinations for the remaining attributes have been fully considered. Hence, if value range $c.l_{o_a}$ to v_a was covered for the current attribute a, the cube representing processed value combinations reaches the full cube dimensions for all attributes that appear later than a in the order o, and is fixed to the currently selected value for all attributes appearing before a in o. Note that the pseudo-code uses a shortcut to assign both cube bounds at once (e.g., $p_i = [v_i, v_i]$ is equivalent to $[p.l_i, p.u_i] = [v_i, v_i]$) in Lines 32 to 34.

Example 3.1. Figure 3 illustrates the containment relationships between different cubes when processing a query with two attributes. Processed cubes are contained within target cubes and target cubes are contained within the entire query cube. The figure represents target cubes that were processed, in different episodes, according to both possible attribute orders. The first one (left) was processed using order A_1, A_2. Hence, values for the first attribute

Algorithm 4 Managing cubes representing unprocessed join input.

1: $U \leftarrow \emptyset$ // Global variable representing unprocessed cubes

2: **Input:** Query q, number of threads n.
3: **Effect:** Initialize set of unprocessed cubes.
4: **procedure** TM.INIT(q, n)
5: $\quad A \leftarrow$ attributes that appear in q in equality join conditions
6: $\quad [l_a, u_a] \leftarrow$ attribute value ranges for all attributes $a \in A$
7: $\quad$ // Identify attribute with largest value domain
8: $\quad a^* \leftarrow \arg\max_{a \in A}(u_a - l_a)$
9: $\quad$ // Use full value range for all but that attribute
10: $\quad f \leftarrow \bigtimes_{a \in A: a \neq a^*} [l_a, u_a]$
11: $\quad$ // Divide largest value domain into per-thread ranges
12: $\quad \delta \leftarrow (u_{a^*} - l_{a^*})/n$
13: $\quad$ // Form one unprocessed cube per thread
14: $\quad U \leftarrow \{f \times [l_{a^*} + i \cdot \delta, l_{a^*} + (i+1) \cdot \delta | 0 \leq i < n]\}$
15: **end procedure**

16: **Output:** Returns an unprocessed hypercube.
17: **function** TM.RETRIEVE
18: $\quad$ **return** Randomly selected cube from U
19: **end function**

20: **Input:** Target cube c to subtract, processed cube set P.
21: **Effect:** Updates set of unprocessed cubes.
22: **procedure** TM.REMOVE(c, P)
23: $\quad$ // Subtract target cube from unprocessed cubes
24: $\quad U \leftarrow U \setminus c$
25: $\quad$ // Add complement of processed cubes as unprocessed
26: $\quad$ **for** $p \in P$ **do**
27: $\quad\quad$ // Get dimensions where p fully covers c
28: $\quad\quad F \leftarrow$ indexes i s.t. $p.l_i = c.l_i$ and $p.u_i = c.u_i$
29: $\quad\quad$ // Get dimensions where p's bounds collapse
30: $\quad\quad S \leftarrow$ indexes i s.t. $p.l_i = p.u_i$
31: $\quad\quad$ // Get single remaining dimension
32: $\quad\quad d \leftarrow$ single remaining dimension not in F or S
33: $\quad\quad$ Create new cube u s.t.
34: $\quad\quad\quad u_d = (p.u_d, c.u_d]; \forall f \in F : u_f = p_f; \forall s \in S : u_s = p_s$
35: $\quad\quad$ // Add newly created cube to unprocessed cubes
36: $\quad\quad$ **if** u is not empty **then**
37: $\quad\quad\quad U \leftarrow U \cup \{u\}$
38: $\quad\quad$ **end if**
39: $\quad$ **end for**
40: **end procedure**

41: **Output:** True iff no unprocessed cubes are left.
42: **function** TM.FINISHED
43: $\quad$ **return true** iff $U = \emptyset$
44: **end function**

change only after trying all values for the second attribute. Therefore, processed cubes fill the target cube "column by column". The other target was processed using the order A_2, A_1. Hence, processed cubes fill the target cube "row by row".

3.3 Avoiding Redundant Work

ADOPT changes between different attribute orders over the course of query processing. This creates the risk of redundant work across different orders. ADOPT avoids redundant work by keeping track of cubes, in the space of join attribute values, that have not been considered yet. More precisely, ADOPT keeps track, at any point in time, of remaining, i.e. unprocessed, cubes. Whenever one of the processing threads requests a new cube to work on, ADOPT returns an unprocessed cube, thereby avoiding redundant work.

Algorithm 4 gives functions used to manipulate cubes. At the beginning (Procedure TM.INIT), it initializes the set of unprocessed cubes to cover the entire attribute space. To do so, ADOPT first retrieves all join attributes (Line 5), then their value ranges (Line 6). Forming one single cube (i.e., the Cartesian product of all value ranges) diminishes chances for parallelization, at least at the start of query processing. Hence, ADOPT divides the attribute value space into equal-sized cubes with one cube per thread (Lines 7 to 14). To do so, it uses the attribute with maximal value domain, dividing its range equally across threads (Line 12). Note that, as discussed in the following, threads are not restricted to processing cubes initially assigned to them over the entire course of query evaluation. Instead, at the end of each episode, unprocessed parts of cubes assigned to a specific thread may get re-assigned to other threads.

Whenever a worker threads requests a cube to work on (Line 13 in Algorithm 2), a randomly selected cube from the set of unprocessed cubes is returned (Line 18 in Algorithm 4). Note that the pseudo-code is slightly simplified, compared to the implementation, by omitting checks used to avoid concurrent changes to the set of unprocessed cubes (by multiple threads).

Whenever a worker threads finished processing, it registers a set of cubes that was processed. It calls Procedure TM.REMOVE to update the set of unprocessed cubes. This function takes two parameters, representing the set of processed cubes as well as the target cube, as input. All processed cubes are contained within the target cube and have a special structure, explained in the following. As a first step, ADOPT removes the target cube from the set of unprocessed cubes in Line24 (the target cube was selected by an invocation of the TM.RETRIEVE function and is therefore contained in the set U). If the set of processed cubes, in aggregate, do not cover the target cube (in general, that is the case), the set of unprocessed cubes is now missing all cubes contained in the target cube but not covered by the processed cubes. Hence, ADOPT adds more unprocessed cubes to reflect the difference.

Each processed cube has a special form, due to the structure of the join algorithm generating it (Lines 23 to 28 in Algorithm 3). All processed cubes are generated according to the same attribute order and based on the same, final values selected for each attribute. Consider one single processed cube, using the selected attribute values v_s for a prefix S of the attribute order, the range of values up to the selected value v_d for a single attribute d, and the full target cube range for the remaining attributes F. Clearly, given the selected values for attributes S, none of the values greater than v_d for attribute d has been considered by the join algorithm (instead, such value combinations would have been considered later by the join algorithm). Hence, the corresponding cube is added to the set of unprocessed cubes (Line 37). Also note that these unprocessed cubes cannot overlap (as, for each pair of unprocessed cubes, there is at least one attribute a for which one cube fixes a value v_a, the other cube covers only values greater than v_a). This preserves the invariant that elements of U, representing unprocessed cubes, do not overlap. It also means that work done by different threads does not overlap. The processing finishes (Procedure TM.FINISHED) whenever no unprocessed cubes are left.

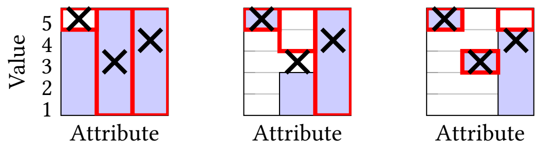

Figure 4: Illustrating cube processing in Example 3.2: The initial target cube $([1,5],[1,5],[1,5])$ **is processed up to** $(5,3,4)$ **(marked by X). Processed cubes are represented by blue rectangles, complementary unprocessed cubes by red rectangles.**

Example 3.2. Figure 4 illustrates the processing of a target cube $([1,5],[1,5],[1,5])$ for an attribute order (A_0,A_1,A_2). In each subplot, the x-axis represents attributes while the y-axis represents attribute values. Assume the timeout for this episode occurs after considering the values $(5,3,4)$ (marked by X). This means that we managed to process the following sub-cubes, left: $([1-4],[1-5],[1-5])$, middle: $(5,[1-2],[1-5])$, right: $(5,3,[1-4])$. We infer the remaining unprocessed sub-cubes that complement these processed sub-cubes with respect to the target cube, left: $(5,[1-5],[1-5])$, middle: $(5,[4-5],[1-5])$, right:$(5,3,5)$.

3.4 Learning Attribute Orders

ADOPT uses reinforcement learning to learn near-optimal attribute orders, over the course of a single query execution. At the beginning of each time slice, ADOPT selects an attribute order that maximizes the tradeoff between exploration and exploitation. It uses the Upper Confidence Bounds on Trees (UCT) algorithm [18] to choose an attribute order. This requires mapping the scenario (of attribute order selection) into a Markov-Decision Problem. Next, we discuss the algorithm as well as the problem model.

An episodic Markov Decision Process (MDP) is generally defined by a tuple $\langle s_0, S, A, T, R\rangle$ where S is a set of states, $s_0 \in S$ the initial state in each episode, A a set of actions, and $T : S \times A \rightarrow S$ a transition function, linking states and action pairs to target states. Component R represents a reward function, assigning states to a reward value. In our scenario, the transition function is deterministic while the reward function is probabilistic (i.e., states are associated with a probability distribution over possible rewards, rather than a constant reward that is achieved, every time the state is visited). The transition function is partial, meaning that certain actions are not available in certain states. Implicitly, we assume that all states without available actions are end states of an episode. After reaching and end state, the current episode ends and the next episode starts (from the initial state s_0 again). Given an MDP, the goal in reinforcement learning [39] is to find a policy, describing behavior that results in maximal (expected) reward. In order to leverage reinforcement learning algorithms for our scenario, we must therefore map attribute order selection into the MDP formalism.

Our goal is to learn a policy that describes an attribute order. The policy generally recommends actions to take in a specific state. Here, we introduce one action for each query attribute. States are associated with attribute order prefixes (i.e., each state represents an order for a subset of attributes). To simplify the notation, we will refer to states by the prefix they represent, to actions by the attribute they correspond to. The transition function connects a first state s_1 to a second state s_2 via action a, if the second state can

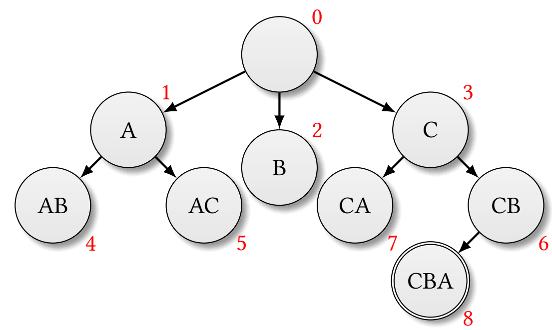

Figure 5: UCT search tree for a query with three attributes: nodes are labeled with partial attribute orders, transitions append one attribute. Red numbers next to nodes represent the episode number at which they are added when selecting attribute orders ABC, BCA, CBA, ABC, ACB, CBA, CAB, and CBA (in that order).

be reached by appending the attribute, represented by the action, to the prefix represented by the first state. More precisely, using the notation introduced before, the transition function links the state-action pair $\langle s_1, a\rangle$ to state $s_2 = s_1 \circ a$ (where $\circ$ represents concatenation). Each state represents a prefix of an attribute order in which each attribute appears at most once. Hence, the actions available in a state correspond to attributes that do not appear in the prefix represented by the state. This means that all states representing a complete attribute order are end states, implicitly. As a further restriction, we do not allow actions representing attributes that do not connect to any attributes in the prefix represented by the current state. This is similar to the heuristic of avoiding Cartesian product joins, used almost uniformly in traditional query optimizers. The reward function is set to zero for all states, except for end states. States of the latter category represent complete attribute orders. Upon reaching such a state, ADOPT executes the corresponding attribute order for a limited number of steps, measuring execution progress. The process by which execution process is measured is described in the following subsections.

ADOPT applies the UCT algorithm to solve the resulting MDP. As the MDP represents the problem of attribute ordering, linking rewards to execution progress, solving the MDP (i.e., finding a policy with maximal expected reward) yields a near-optimal attribute order. The UCT algorithm represents the state space as a search tree. Nodes represent states while tree edges represent transitions. Tree nodes are associated with statistics, establishing confidence bounds on the average reward associated with the sub-tree rooted at that node. Confidence bounds are updated as new reward samples become available. In each episode, the UCT algorithm selects a path from the search tree root to one of the leaf nodes. At each step, the UCT algorithm selects the child node with maximal upper confidence bound (hence the name of the algorithm). This approach converges to optimal policies [18]. After selecting a path to a leaf and calculating the associated reward, the UCT algorithm updates confidence bounds for each node on that path.

ADOPT grows the UCT search tree gradually over the course of query execution. At the start of execution, the tree only contains the root node. Then, in each episode, the tree is expanded by at most one node. Which nodes are added depends on the selected attribute orders. Each attribute order corresponds to a sequence of states in the MDP (a state represents an attribute order, each state appending one attribute, compared to its predecessor). In the fully grown search tree, each state is associated with one node. If, for the currently selected attribute order, some of the states do not have associated nodes in the tree yet, ADOPT expands the tree by adding a node for the first such state. ADOPT uses the partial tree to select attribute orders as follows. Given a state for which all possible successor states have associated nodes in the tree (i.e., reward statistics are available), ADOPT uses the aforementioned principle and selects the attribute that maximizes the upper confidence bound on reward values. If some of the successor states do not have associated nodes yet, ADOPT transitions to a randomly selected state among them (which will create a corresponding node). As a special case, if no nodes are available for any of the successor states, ADOPT selects the next attribute with uniform random distribution.

Example 3.3. Given a query with three attributes (A, B, and C), assume that ADOPT selects the following attribute orders in the first episodes (some orders are selected in multiple episodes): ABC, BCA, CBA, ABC, ACB, CBA, CAB, CBA. Figure 5 shows the UCT search tree after those episodes. Nodes represent partial attribute orders and edges represent the addition of one attribute. Next to each node, in red, the figure shows the number of the episode in which the node was added. Initially (episode zero), the tree contains only the root node. In the first episode, ADOPT selects order ABC, adding a node for the first prefix (A) without corresponding node in the tree. Later, in episode four, ADOPT selects order ABC and, again, adds a node for the first prefix (AB) for which no node has been created. Once nodes are added, ADOPT starts collecting reward statistics for all attribute orders extending the corresponding prefix. These statistics are used to select attribute orders in future episodes.

3.5 Estimating Order Quality

The reinforcement learning, described in Section 3.4, is guided by reward values. Next, we discuss the definition of the reward function. Before that, we introduce an auxiliary function, measuring the volume of a cube as the product of range sizes over all dimensions:

$$Volume(c) = \prod_{i}(c.u_i - c.l_i) \tag{1}$$

With a slight abuse of notation, we write $Volume(q)$ to denote the volume of the cube, spanned by all join attributes of a query q.

In order to fully process a query, ADOPT must cover the cube representing the entire space of attribute value combinations. Hence, the more volume of that cube we cover per time unit, the faster query processing is. Of course, even for a fixed attribute order, the volume processed per time unit may vary across different parts of the data (e.g., since the number of result tuples per volume varies). However, the fastest order processes most volume in average, averaging over the entire data set, and the UCT algorithm converges to decisions with highest average reward, even if the reward function is noisy [18]. This implies that volume covered is a useful measure of progress. The reward function, presented next, follows that intuition. Given a set of processed cubes P for query q, it uses the aggregate volume covered, scaled to the total volume to process (scaling ensures reward values between zero and one, consistent with the requirements of the UCT algorithm):

$$Reward(P, q) = (\sum_{p \in P} Volume(p))/Volume(q) \tag{2}$$

4 ANALYSIS

In this section, we prove that ADOPT converges to optimal attribute orders. Two further properties, correctness and worst-case optimality, are analyzed in the appendix of our extended technical report [48]. First, we show that ADOPT must finish processing once the accumulated rewards reach a precise threshold.

THEOREM 4.1. *Join processing finishes once the sum of accumulated rewards over all threads and episodes reaches one.*

PROOF. Reward is proportional to the volume of the cube covered, scaled to the size of the full cube. Hence, accumulating a reward sum of one means that a volume equal to the full cube has been processed. Furthermore, ADOPT avoids covering overlapping cubes by different threads and in different episodes (independently of the attribute order). Hence, once the accumulated reward reaches one, processed cubes must cover the full cube. □

This implies that the reward function is a good measure of attribute order quality indeed.

THEOREM 4.2. *The attribute order with the highest average reward per episode minimizes the number of computational steps.*

PROOF. For any attribute order o, processing finishes once the accumulated rewards reach one (Theorem 4.1). Therefore, the average reward r_o per episode for o is inversely proportional to the number of episodes e_o needed by o, i.e. $r_o = 1/e_o$. Also, the number of computational steps per episode is constant. Therefore, minimizing the number of episodes needed maximizes the average reward. □

This implies convergence to optimal attribute orders.

COROLLARY 4.3. *ADOPT converges to an optimal attribute order.*

PROOF. Following Theorem 4.2, the order with the highest average reward is also the fastest one to process. Furthermore, the UCT algorithm used by ADOPT converges to a solution with maximal expected reward [18]. Hence, ADOPT converges to an attribute order that minimizes the number of processing steps. □

5 EXPERIMENTAL EVALUATION

We confirm experimentally that ADOPT outperforms a range of competitors for both acyclic and cyclic queries from the join order benchmark [13], standard decision support benchmarks (i.e., TPC-H and JCC-H [8]), and graph data [21, 31] workloads. The robustness of ADOPT's query evaluation becomes more evident for queries with an increasingly larger number of joins and with filter conditions whose joint selectivity is hard to assess correctly at optimization time. The superior performance of ADOPT over its competitors is due to the interplay of its four key features: worst-case optimal join evaluation; reinforcement learning that eventually converges to near-optimal attribute orders (Sec. 5.5); hypercube data decomposition (Sec. 5.4); and domain parallelism (Sec. 5.6). For

Table 1: Overall runtime (in seconds) to compute all queries for each benchmark. For the JOB benchmark, ">" indicates the time is only for some of the 113 queries. For the four graph datasets, ">" indicates the time exceeded the six-hour (21,600 seconds) timeout for some of the cyclic queries. The multiplicative factors in parentheses after the runtimes of systems are the speedups of ADOPT over these systems.

Systems	JOB	ego-Facebook	ego-Twitter	soc-Pokec	soc-Livejournal1	TPC-H	JCC-H
ADOPT	45	**4,414**	**3,931**	**9,268**	**26,350**	141	**194**
System-X	> 287 (6.38x)	> 22,459 (5.09x)	11,384 (2.90x)	> 23,623 (2.55x)	> 63,878 (2.42x)	–	–
EmptyHeaded	–	6,783 (1.54x)	10,381 (2.64x)	> 43,444 (4.69x)	> 55,144 (2.09x)	–	–
PostgreSQL	285 (6.33x)	> 67,774 (15.35x)	> 70,515 (17.94x)	> 67,016 (7.23x)	> 101,193 (3.84x)	182 (1.53x)	> 216,122 (1,114x)
MonetDB	**41** (0.91x)	> 66,165 (14.99x)	> 86,596 (22.03x)	> 59,131 (7.23x)	> 96,222 (3.84x)	**17** (0.12x)	> 216,035 (1,114x)
SkinnerDB	65 (1.44x)	> 69,366 (15.71x)	> 129,741 (33.00x)	> 95,374 (10.29x)	> 101,392 (3.85x)	173 (1.23x)	320 (1.6x)

lack of space, we defer to a technical report [48] further experiments on: memory consumption, scalability with the number of join attributes per table, sorting and synchronization overhead, and the performance comparison of ADOPT and System-X.

5.1 Experimental Setup

We benchmark the query engines on acyclic and cyclic queries.

Benchmark for acyclic queries. The join order benchmark (JOB) [13] consists of 113 queries over the highly-correlated IMDB real-world dataset. This benchmark shows an orders-of-magnitude performance gap between different join orders for the same query. TPC-H (JCC-H [8]) is a benchmark used for decision support, comprising of 22 queries that incorporate standard SQL predicates. In TPC-H, data is synthetically generated with uniform distribution, whereas in JCC-H, the data is highly skewed, which makes JCC-H a harder benchmark to optimize. In our experiments, we use TPC-H/JCC-H with scaling factor ten. We omit four queries in TPC-H (JCC-H) queries, Q2, Q13, Q15, and Q22, for lack of support for non-integer join columns, outer joins, views, and substring functions.

Benchmark for cyclic queries. We follow prior work on benchmarking worst-case optimal join algorithms against traditional join plans [31] and consider the evaluation of clique and cycle queries over the binary edge relations of four graph datasets from the SNAP network collection [21]. The considered queries are as follows:

- n-clique: Compute the cliques of n distinct vertices. Such a clique has an edge between any two of its vertices. For instance, the 3-clique is the triangle:
  $edge(a, b), edge(b, c), edge(a, c), a < b < c$
- n-cycle: Compute the cycles of n distinct nodes. Such a cycle has an edge between the i-th and the $(i + 1)$-th vertices for $1 \leq i < n$ and an edge between the first and the last vertices. For instance, the 4-cycle query is:
  $edge(a, b), edge(b, c), edge(c, d), edge(a, d), a < b < c < d$

The inequalities in the above queries enforce that each node in the clique/cycle is distinct. Instead of returning the list of all distinct cliques/cycles, all systems are instructed to return their count. ADOPT counts the result tuples as they are computed. The reason for returning the count is to avoid the time to list the result tuples and only report the time to compute them.

Systems. ADOPT is implemented in JAVA (jdk 1.8). It uses 10,000 steps per episode and UCT exploration ratio 1E-6. The competitors are: the open-source engines MonetDB [9] (Database Server Toolkit v11.39.7, Oct2020-SP1) and PostgreSQL 10.21 [38] that employ traditional join plans; a commercial engine System-X (implemented in C++) that uses the worst-case optimal LFTJ algorithm [45]; the open-source engine EmptyHeaded that uses a worst-case optimal join algorithm [2]; and SkinnerDB [41] (implemented in Java jdk 1.8) that uses reinforcement learning to learn an optimal join order for traditional query plans.

Setup. We run each experiment five times and report the average execution time. We used a server with 2 Intel Xeon Gold 5218 CPUs with 2.3 GHz (32 physical cores)/384GB RAM/512GB hard disk. ADOPT, EmptyHeaded, MonetDB, SkinnerDB, and System-X were set to run in memory. By default, all engines use 64 threads. For all systems, we create indexes to optimize performance (index creation overheads are reported separately in the extended technical report [48]). For systems such as MonetDB that create indexes automatically, based on properties of observed queries, we perform one warm-up run before starting our measurements.

5.2 Runtime Performance

ADOPT puts together worst-case optimal join algorithm, which is primarily motivated by cyclic queries, and adaptive processing, which is motivated by scenarios in which size and cost prediction for query planning is difficult (e.g., due to data skew or complex queries). This motivates the following hypotheses.

Hypothesis 1. *ADOPT outperforms baselines without worst-case optimal join algorithms on cyclic queries.*

Hypothesis 2. *ADOPT outperforms non-adaptive baselines for complex queries on skewed data.*

Hypothesis 3. *ADOPT performs worse, compared to baselines, if queries are simple, acyclic, and are executed on uniform data.*

Table 1 reports the total time in seconds for different systems and benchmarks. ADOPT performs best for the four benchmarks on graphs, featuring cyclic queries. Figure 6 breaks those results down by query size and query type. Compared to other baselines using worst-case optimal joins, ADOPT's gains derive from larger queries with more predicates, creating the potential for inter-predicate correlations that are hard to predict. This makes it difficult to select optimal attribute orders before execution. PostgreSQL, MonetDB, and SkinnerDB suffer from over-proportionally large intermediate results when processing cyclic queries as they do not implement worst-case optimal joins.

The join order benchmark (JOB) features acyclic queries but non-uniform data (i.e., it contains some elements that should benefit ADOPT in the comparison and some that have the opposite effect). Here, ADOPT performs comparably but slightly worse to the best baseline: MonetDB. For System-X, Table 1 only reports time for executing a subset of the queries (39 out of 113). The remaining queries have IS/NOT NULL and IN predicates that are not supported by System-X. EmptyHeaded needs more than five days to construct the data indices (tries) it needs for the non-binary JOB tables so we were not able to report its runtime on the JOB queries.

TPC-H and JCC-H share the same query templates and database schema but differ in the database content: TPC-H uses uniform data whereas JCC-H uses highly correlated data. On TPC-H, MonetDB performs best and outperforms ADOPT significantly. This is consistent with prior work [1], showing that systems with worst-case optimal joins (specifically: the LFTJ that ADOPT uses internally) perform significantly worse than MonetDB on TPC-H. Given those prior results and limited support for TPC-H queries in System X and EmptyHeaded, we compare only to MonetDB as the strongest baseline. Besides drawbacks due to the join algorithm, ADOPT incurs overheads due to adaptive processing which is unnecessary on TPC-H: predicting sizes of intermediate results and plan execution cost is relatively easy due to uniform data.

On the other hand, ADOPT outperforms all other systems on JCC-H. Despite sharing the same query templates with TPC-H, JCC-H makes query optimization hard due to highly correlated data. Here, both adaptive baselines (SkinnerDB and ADOPT) benefit, with ADOPT being significantly faster, whereas all other systems reach the timeout of six hours. This means, even on acyclic queries, traditionally not considered the sweet spot for LFTJ-based joins [1], ADOPT is preferable if data is sufficiently correlated.

5.3 Robustness

ADOPT does not rely on query optimization to pick the best attribute order. This can be a significant advantage for queries with user-defined functions or selection predicates, for which there are no available selectivity estimates. Mainstream systems pick a query plan that may be arbitrarily off from a good one. In contrast, ADOPT may quickly realize that such a plan is subpar and switch to a different one. To benchmark this observation, we consider experiments to assess the *robustness* of ADOPT and System-X, which are the two systems we use that rely on attribute orders, when adding to the join queries very simple (unary) yet arbitrary selection conditions that can throw off standard query optimizers.

Hypothesis 4. *ADOPT outperforms System-X consistently when varying the selectivity of unary predicates.*

Figure 7 shows the relative speedup of ADOPT over System-X as we vary the selectivity of unary predicates (selections with constants) on three randomly chosen attributes: we choose the five selectivities 0.2, 0.4, 0.6, 0.8, and 1 for the three attributes along the x-axis, y-axis, and the circles for an (x,y)-point. The color of each 3D point in the plot varies from blue to red: The more intense the red is, the higher is the speedup of ADOPT over System-X. System-X mostly outperforms ADOPT for 3-cliques. ADOPT is up to three times faster than System-X for all other cliques and cycles. This is due to the difficulty of optimizers to pick a good query plan in the absence of selectivity estimates, here even for unary predicates.

5.4 Hypercube Data Partitioning

We next benchmark the effect of our hypercube partitioning scheme and verify that it indeed leads to faster execution time than SkinnerDB's alternatives called shared prefix+offset progress tracker [41].

Hypothesis 5. *Hypercube partitioning leads to faster execution than shared prefix progress tracker and offset progress tracker.*

SkinnnerDB shares progress between all join orders with the same prefix (iterating over all possible prefix lengths). Given a join order, it restores a state by comparing execution progress between the current join order and all other orders with the same prefix and by selecting the most advanced state. Offset progress tracker keeps the last tuples of each table that have been joined with all other tuples already. Using hypercube partitioning, ADOPT executes the episodes on disjoint parts of the input data so it can trivially compute distributive aggregates such as count. This is not the case for SkinnerDB's partitioning: To avoid recomputation of the same result in different episodes, it has to maintain a data structure (concurrent hash map). In the multi-thread environment, the prefix share progress tracker blocks the concurrent execution and causes significant synchronization overhead.

Figure 8 shows the speedup of using the hypercube partitioning over using the prefix+offset share progress tracker in ADOPT. The hypercube approach consistently has significant smaller overhead than prefix+offset share progress tracker. For larger (above 4) clique and cycle queries, the speedup is 10x to 100x.

5.5 Time Breakdown by Attribute Order

Hypothesis 6. *ADOPT spends most time on executing near-optimal attribute orders.*

We verified this hypothesis for n-clique and n-cycle queries with $n \in \{4, 5\}$, since for these queries it was feasible to generate and execute all possible attribute orders. This was necessary to understand which orders are better than others and assess whether ADOPT uses predominantly good or poor orders. We plot the orders that we select and their quality relative to the optimal orders (i.e., with lowest execution time) in Figure 9. The x-axis is the number of time slices that use an order: the larger the x-value, the more we use an order. The y-axis is execution time of an order relative to the optimal one: The smaller the y-value, the closer to the optimal the order is. For 4-clique and 4-cycle, ADOPT spends more than 10^6 (over 95% frequency) times on executing an order with near-optimal performance. For 5-cycle and 5-clique, ADOPT picks a near-optimal order more than 10^8 times (over 98% frequency). ADOPT thus quickly converges to a near-optimal order and then uses it for most of the processing, which confirms our hypothesis.

Table 2 compares ADOPT and LFTJ with an optimal attribute order: The runtime gap decreases from 2.52x for 3-clique/cycle to

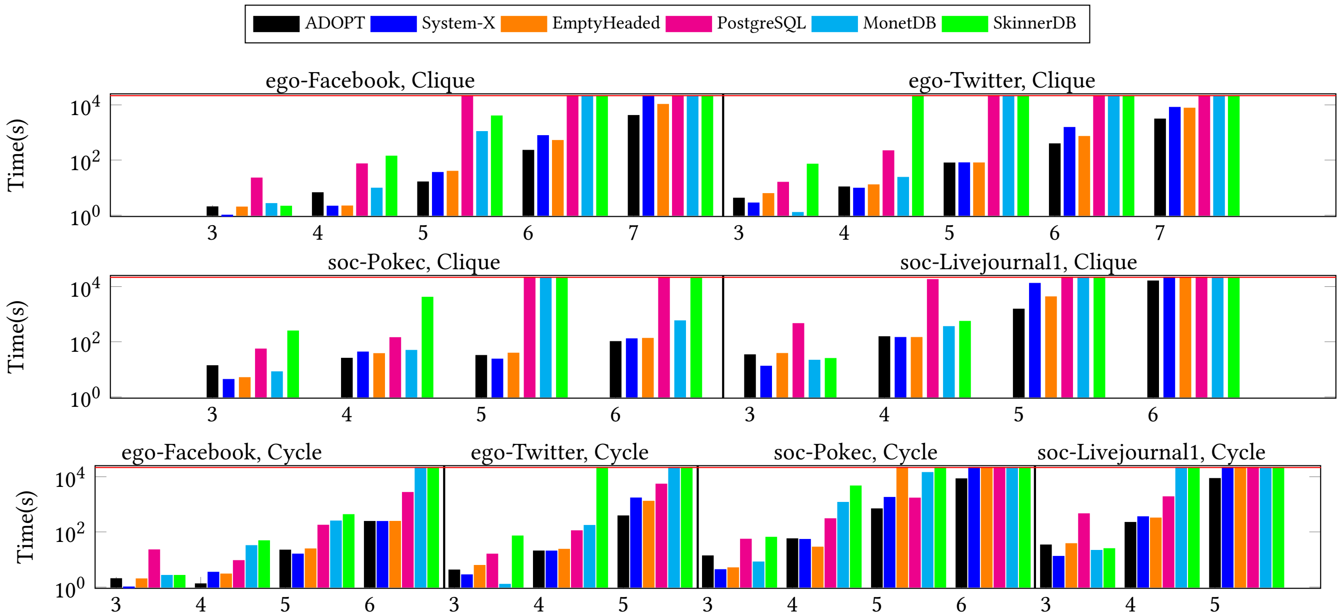

Figure 6: Execution (wall-clock) time for clique and cycle queries on four graphs (x axis represents query size).

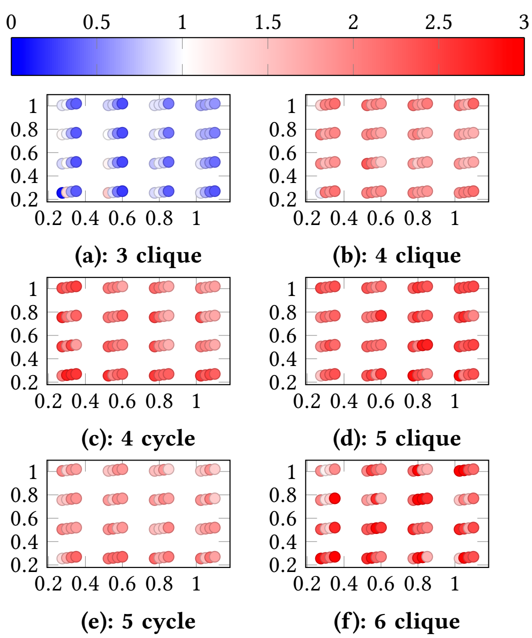

Figure 7: Speedup of ADOPT over System-X when varying the selectivity of newly added unary predicates on three randomly chosen attributes (along the x-axis, y-axis, and the circles for an (x,y)-point). More intense red (blue) means higher (lower) speedup. All queries are executed on ego-Twitter.

1.48x (1.14x) for 5-clique (5-cycle). This is remarkable, given that ADOPT tries out several attribute orders and switches between them, whereas LFTJ only uses one attribute order, which is optimal. Table 2 also shows that ADOPT takes significantly less time than the average runtime of LFTJ over all attribute orders.

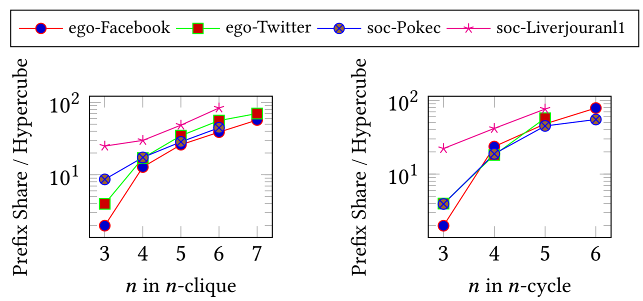

Figure 8: Speedup of using our hypercube approach versus using prefix+offset share progress tracker in ADOPT.

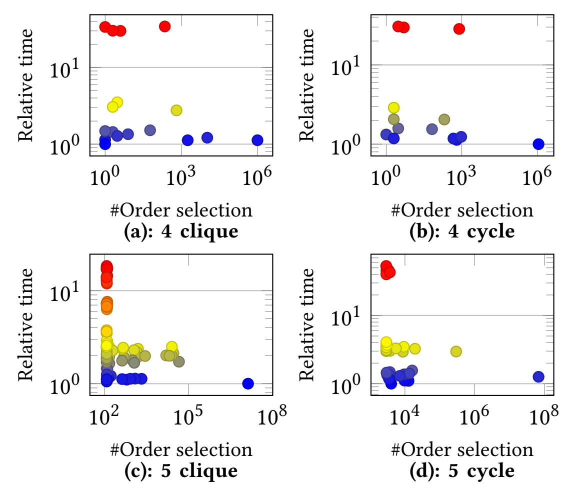

Figure 9: Selections of orders with different quality on the ego-Twitter dataset.

Table 2: Execution times (sec) for clique and cycle queries on ego-Twitter of: ADOPT, LFTJ with optimal attribute order (OPT), average runtime of LFTJ over all attribute orders (AVG). Relative speedup of OPT over ADOPT (last column).

	ADOPT	OPT	AVG	ADOPT/OPT
3 clique	4.1	1.6	3.7	2.52
4 clique	10.5	6.8	23.9	1.54
5 clique	77.9	52.5	275.6	1.48
3 cycle	4.1	1.6	3.5	2.52
4 cycle	20.1	17.4	58.9	1.16
5 cycle	377.9	328.8	3618.1	1.14

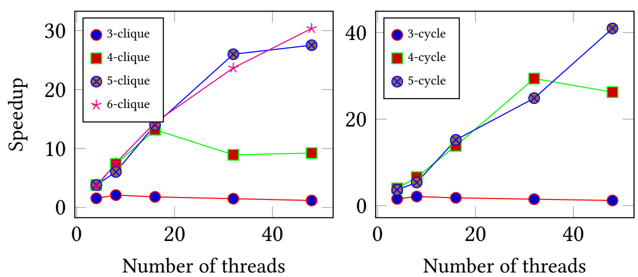

Figure 10: Speedup of multi-threaded ADOPT over single-threaded ADOPT for clique and cycle queries on ego-Twitter.

5.6 Parallelization

Hypothesis 7. *ADOPT achieves almost linear speedup for large cyclic queries.*

Figure 10 plots the speedup of ADOPT as a function of the number of threads. ADOPT achieves significant speedups for large clique and cycle queries. In particular, it achieves nearly 30x speedup on 5- and 6-clique, and 40x speedup on 5-cycle (with 48 threads). The main reason is that the hypercube approach partitions disjointly the workload across threads, minimizing synchronization overheads.

6 RELATED WORK

The choice of an attribute order, for worst-case optimal join algorithms, resembles the problem of join order selection for traditional join algorithms [37]. Both tuning decisions have significant impact on processing performance. At the same time, it is hard to find good attribute orders before query processing starts, mainly due to challenges in estimating execution cost for specific orders (e.g., due to challenges in estimating sizes of intermediate results). The latter problem has been well documented for traditional query optimizers [13, 23]. Our experiments demonstrate that it appears in the context of worst-case optimal join algorithms as well.

Adaptive processing [6, 10, 35, 43, 49, 52] has been proposed as a remedy to this problem, allowing the engine to switch to a different join order during query execution based on run time feedback. While early work focused on stream data processing [6, 10, 35, 43] (where query execution times are assumed to be longer), adaptive processing has recently also gained traction for classical query processing [26, 41]. SkinnerDB [41] is the closest in spirit to ADOPT: both use reinforcement learning and adaptive processing. However, ADOPT uses an anytime version of a worst-case optimal join algorithm, whereas SkinnerDB's join algorithm is not optimal. The learning problems (i.e., actions and states of the corresponding MDPs) differ between the systems as ADOPT optimizes attribute orders whereas SkinnerDB orders tables. Most importantly: ADOPT introduces a novel data structure, characterizing precisely the cubes in the space of attribute value combinations that have not been processed yet, along with operators for updating it after each episode. This data structure avoids redundant work across episodes and attribute orders as well as across threads. This property is crucial to be able to maintain optimality guarantees for equi-joins when switching between attribute orders. Instead, SkinnerDB uses a tree-based data structure that reduces but does not completely avoid redundant work across join orders that are dissimilar. As the amount of redundant work is hard to bound, it is difficult to maintain worst-case optimality guarantees with such mechanisms.

Our work uses reinforcement learning to select attribute orders. It relates to works that employ learning for database tuning [14, 22, 40, 44, 46, 47, 50] and in particular for query optimization [20, 24, 25, 51]. Our work differs as it focuses on learning and specialized data structures for worst-case optimal join algorithms.

Prior work on query optimization for worst-case optimal joins investigates "model-free" information-theoretic cardinality estimation. A seminal work, which enabled reasoning about worst-case optimal join computation, established tight bounds on the worst-case size of join results [5], the so-called AGM bound that is defined as the cost of the optimal solution of a linear program derived from the joins and the sizes of the input tables. This is further refined in the presence of functional dependencies [12] and for succinct factorized representations of query results [33]. The latest development extends this line of work with data degree constraints and histograms [28]. Classical approaches to query optimization based on heuristics [11] and data statistics [2, 4] have also been considered. To the best of our knowledge, ADOPT is the first adaptive approach for optimization in the context of worst-case optimal join algorithms. Our approach is free from cost-based heuristics.

7 CONCLUSION

Worst-case optimal join algorithms and adaptive processing strategies have been two of the most exciting advances in join processing over the past decades. Worst-case optimal joins enable efficient processing of cyclic queries. Adaptive processing allows handling complex queries where a-priori optimization is hard. For the first time, ADOPT brings together these two techniques, resulting in attractive performance for both acyclic and cyclic queries and in particular excellent performance for large cyclic queries.

ADOPT is an adaptive framework readily applicable to further query processing techniques, e.g., factorized databases [7] and functional aggregate queries [16]. These works combine worst-case optimal joins with effective techniques to push aggregates past joins to achieve the best known computational complexity for query evaluation. In future work, we plan to merge this line of work with ADOPT-style adaptivity.

REFERENCES

[1] Christopher Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Re. 2018. Levelheaded: A Unified Engine for Business Intelligence and Linear Algebra Querying. In *ICDE*. 449–460. https://doi.org/10.1109/ICDE.2018.00048

[2] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *SIGMOD*. 431–446. https://doi.org/10.1145/2882903.2915213

[3] Molham Aref. 2019. Relational Artificial Intelligence. In *Datalog*, Vol. 2368. 1. http://ceur-ws.org/Vol-2368/invited1.pdf

[4] Molham Aref, Balder Ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD*. 1371–1382. https://doi.org/10.1145/2723372.2742796

[5] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767. https://doi.org/10.1137/110859440

[6] Ron Avnur and Jm Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*. 261–272. https://doi.org/10.1145/342009.335420

[7] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Zavodny. 2013. Aggregation and Ordering in Factorised Databases. *PVLDB* 6, 14 (2013), 1990–2001. https://doi.org/10.14778/2556549.2556579

[8] Peter Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. 2018. JCC-H: Adding Join Crossing Correlations with Skew to TPC-H. In *TPCTC, Revised Selected Papers*. 103–119. https://doi.org/10.1007/978-3-319-72401-0_8

[9] Peter A Boncz, Martin L Kersten, and Stefan Manegold. 2008. Breaking the Memory Wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85. https://doi.org/10.1145/1409360.1409380

[10] Amol Deshpande. 2004. An Initial Study of Overheads of Eddies. *SIGMOD Rec.* 33, 1 (2004), 44–49. https://doi.org/10.1145/974121.974129

[11] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *PVLDB* 13, 11 (2020), 1891–1904. https://doi.org/10.14778/3407790.3407797

[12] Georg Gottlob, Stephanie Tien Lee, Gregory Valiant, and Paul Valiant. 2012. Size and Treewidth Bounds for Conjunctive Queries. *J. ACM* 59, 3 (2012), 16:1–16:35. https://doi.org/10.1145/2220357.2220363

[13] Andrey Gubichev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[14] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. 2019. Towards Learning a Partitioning Advisor with Deep Reinforcement Learning. *SIGMOD* (2019). https://doi.org/10.1145/3329859.3329876

[15] Guodong Jin, Nafisa Anzum, and Semih Salihoglu. 2022. GRainDB: A Relational-core Graph-Relational DBMS. In *CIDR*. https://www.cidrdb.org/cidr2022/papers/p57-jin.pdf

[16] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2017. Juggling Functions Inside a Database. *SIGMOD Rec.* 46, 1 (2017), 6–13. https://doi.org/10.1145/3093754.3093757

[17] Akram Khodadadi and Shahram Saeidi. 2021. Discovering the Maximum k-Clique on Social Networks Using Bat Optimization Algorithm. *Computational Social Networks* 8, 1 (2021). https://doi.org/10.1186/s40649-021-00087-y

[18] Levente Kocsis and C Szepesvári. 2006. Bandit Based Monte-Carlo Planning. In *ECML*. 282–293. https://doi.org/10.1007/11871842_29

[19] Valdis E Krebs. 2002. Mapping networks of terrorist cells. *Connections* 24, 3 (2002), 43–52.

[20] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2020. Learning to Optimize Join Queries with Deep Reinforcement Learning. In *aiDM*. 1–6. http://arxiv.org/abs/1808.03196

[21] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[22] 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. https://doi.org/10.14778/3352063.3352129

[23] Guy Lohman. 2014. Is Query Optimization a "Solved" Problem?. In *Proc. Workshop on Database Query Optimization*, Vol. 13. 10.

[24] 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. https://doi.org/10.14778/3342263.3342644

[25] Tim Marcus, Ryan and Negi, Parimarjan and Mao, Hongzi and Tatbul, Nesime and Alizadeh, Mohammad and Kraska. 2022. Bao: Making Learned Query Optimization Practical. In *SIGMOD Rec.*, Vol. 51. 5. https://doi.org/10.1145/3542700.3542702

[26] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. 2020. Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling. *PVLDB* 14, 2 (2020), 101–113. https://doi.org/10.14778/3425879.3425882

[27] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *BTW (LNI)*, Vol. P-241. 383–402. https://dl.gi.de/handle/20.500.12116/2418

[28] Hung Q. Ngo. 2022. On an Information Theoretic Approach to Cardinality Estimation (Invited Talk). In *ICDT*, Vol. 220. 1:1–1:21. https://doi.org/10.4230/LIPIcs.ICDT.2022.1

[29] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-Case Optimal Join Algorithms. *J. ACM* 65, 3 (2018), 16:1–16:40. https://doi.org/10.1145/3180143

[30] Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2013. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.* 42, 4 (2013), 5–16. https://doi.org/10.1145/2590989.2590991

[31] Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. 2015. Join Processing for Graph Patterns: An Old Dog with New Tricks. In *GRADES*. 2:1–2:8. https://doi.org/10.1145/2764947.2764948

[32] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (2016), 5–16. https://doi.org/10.1145/3003665.3003667

[33] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Trans. Database Syst.* 40, 1 (2015), 2:1–2:44. https://doi.org/10.1145/2656335

[34] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. 2005. Uncovering the Overlapping Community Structure of Complex Networks in Nature and Society. *Nature* 435, 7043 (2005), 814–818.

[35] Li Quanzhong, Shao Minglong, Volker Markl, Kevin Beyer, Latha Colby, and Guy Lohman. 2007. Adaptively Reordering Joins During Query Execution. In *ICDE*. 26–35. https://doi.org/10.1109/ICDE.2007.367848

[36] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *SIGMOD*. ACM, 3–18. https://doi.org/10.1145/2882903.2882939

[37] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD*. 23–34. https://doi.org/10.1145/582095.582099

[38] Michael Stonebraker and Lawrence A Rowe. 1986. The Design of Postgres. *Sigmod Rec.* 15, 2 (1986), 340–355. https://doi.org/10.1145/16894.16888

[39] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. 532 pages. https://doi.org/10.1016/s1364-6613(99)01331-5

[40] Immanuel Trummer. 2022. DB-BERT: A Database Tuning Tool that "Reads the Manual". In *SIGMOD*. 190–203. https://doi.org/10.1145/3514221.3517843

[41] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *SIGMOD*. 1153–1170. https://doi.org/10.1145/3299869.3300088

[42] 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. *TODS* 46, 3 (2021). https://doi.org/10.1145/3464389

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

[44] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-Based Automatic Configuration Tuning Services on Real-World Database Management Systems. *PVLDB* 14, 7 (2021), 1241–1253. https://doi.org/10.14778/3450980.3450992

[45] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*. 96–106. https://doi.org/10.5441/002/icdt.2014.13

[46] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. Demonstrating UDO: A Unified Approach for Optimizing Transaction Code, Physical Design, and System Parameters via Reinforcement Learning. In *SIGMOD*. 2794–2797. https://doi.org/10.1145/3448016.3452754

[47] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: Universal Database Optimization using Reinforcement Learning. *PVLDB* 14, 13 (2021), 3402–3414. http://arxiv.org/abs/2104.01744

[48] Junxiong Wang, Immanuel Trummer, Ahmet Kara, and Dan Olteanu. 2023. *ADOPT: Adaptively Optimizing Attribute Orders for Worst-Case Optimal Join Algorithms via Reinforcement Learning*. Technical Report. http://arxiv.org/abs/2307.16540.

[49] Ziyun Wei and Immanuel Trummer. 2022. SkinnerMT: Parallelizing for Efficiency and Robustness in Adaptive Query Processing on Multicore Platforms. *PVLDB* 16, 4 (2022), 905–917. https://doi.org/10.14778/3574245.3574272

[50] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, and Philip A. Bernstein. 2022. Budget-Aware Index Tuning with Reinforcement Learning. In *SIGMOD*. 1528–1541. https://doi.org/10.1145/3514221.3526128

[51] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *ICDE*, Vol. 2020-April. 1297–1308. https://doi.org/10.1109/ICDE48307.2020.00116

[52] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking ahead makes query plans robust. In *SIGMOD*. 889–900. https://doi.org/10.14778/3090163.3090167