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(12): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

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 [24, 37]. LeapFrog TrieJoin (LFTJ) is a a prime example of a worst-case optimal join algorithm [37]. 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 [25]. 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 [27, 28], graph processing [1, 12] and general query processing [9], in-database machine learning [30], and in the commercial systems LogicBlox [3] and RelationalAI [2].

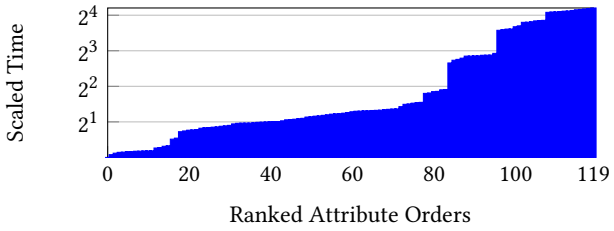

Figure 1: Execution time of different attribute orders for the five-clique query on the ego-Twitter graph.

The performance of any worst-case optimal join algorithm crucially depends, however, on the order in which join attributes (i.e., groups of join columns that are linked by equality constraints) are processed. Even though all orders are equally good to achieve optimality in the *worst case*, their performance may differ significantly in practice. Not unlike the join order for traditional join algorithms, the optimal attribute order depends on the database state. For a fixed database state, the performance gap between good and bad attribute orders is significant, even for moderately sized data sets. Whereas worst-case optimality guarantees (asymptotically) optimal performance, relative to a *worst-case database state*, it does not guarantee optimality with regards to the *actual state*. Hence, the theoretical guarantees offered by worst-case optimal algorithms do not remove the need for careful attribute order selection.

Example 1.1. Figure 1 shows an extract from the experiments, illustrating the need for accurate attribute order selection. It compares execution times achieved by LFTJ (scaled to the time of the fastest order) for different attribute orders and the same query that asks for the number of tuples of five distinct nodes that form a clique. Different (120) attribute orders are shown on the x-axis,

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:XX.XX/XXX.XX

ranked by execution time. The performance gap between the best and worst attribute order is more than 16x. Clearly, the choice of an attribute order has significant impact on performance and near-optimal orders are sparse.

Execution engines using worst-case optimal join operators (e.g., the LogicBlox system [3]) 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 [18]. The experiments in Section 5 show that this case appears in the context of optimization for worst-case optimal join algorithms as well.

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 [35], 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, join order [11] and SNAP graph data [17, 26] workloads. For small queries, adaptive processing does not result in performance improvements over competitors. However, for larger and more complex queries, 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 shows an overview of the ADOPT system, illustrating its primary components. ADOPT processes SPJA queries. It performs in-memory data processing and uses a columnar data layout. The implementation uses Java and supports 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 a given query, ADOPT first performs a pre-processing step to filter the tables using unary predicates from the query. After that, the following join phase is executed on the filtered tables.

The worst-case optimal join algorithm used by ADOPT is based on LeapFrog TrieJoin (LFTJ). This algorithm 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 a high frequency.

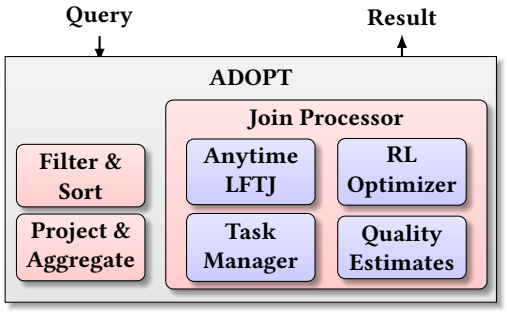

Figure 2: Overview of ADOPT system components.

This enables the adaptive processing strategy, allowing ADOPT to identify near-optimal attribute orders, based on run time feedback.

Besides the join algorithm itself, ADOPT uses a reinforcement learning based optimizer. 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). However, by averaging over different invocations for the same attribute order, ADOPT obtains more and 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.

The LFTJ requires index data structures on the join input tables. More precisely, it requires trie data structures, enabling quick access to tuples in a table with specific values in join columns. The desired structure of the trie depends on the attribute order. The original version of LFTJ uses one single attribute order, thereby requiring only a single trie per joined table. ADOPT, however, aims to explore various attribute orders during the processing of a single query. Hence, it creates new tries on input tables as the need arises. Specifically, whenever a new attribute order is selected by the reinforcement learning algorithm, ADOPT checks for the presence of required tries. If some of them are missing, it first creates corresponding data structures, before resuming join execution with the selected order. An input table can have as many tries as the possible total orders on its join attributes; e.g., for a table with two join attributes, there can be at most two tries.

Join processing terminates for the query 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 queries, ADOPT integrates join processing with aggregation and does not need to perform a post-processing stage.

ADOPT supports parallel processing via multi-threading in several of the aforementioned processing phases. Specifically, ADOPT parallelizes the join preparation phase (i.e., unary predicates are evaluated on different data partitions in general) and sorts data (to enable trie-based access) in parallel. During the join phase itself, ADOPT exploits multiple threads by 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

Algorithm 1 is used by ADOPT to process queries. The input is a query, joining m relations via equality joins (possibly with additional unary predicates), the number of data processing threads, and the number of computational steps spent to evaluate a selected attribute order.

First, ADOPT filters query tables via unary predicates. After that, the only remaining predicates are equality join predicates. Next, the algorithm initializes the set of join result tuples, the reinforcement learning algorithm, and the task manager. The reinforcement learning is initialized by specifying the search space of attribute orders (which depends on query properties). The task manager is initialized with the input query as well as the number of processing threads. 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 until that termination condition is satisfied. In each iteration, ADOPT first selects an attribute order via reinforcement learning. Then,

Algorithm 1 Main function of ADOPT, processing queries.

```
1:  Input: Query q, number of threads n, per-episode budget b
2:  Output: Result R
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 ¬ CS.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 join result
22:     return R
23: end function
```

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
```

it executes that order, in parallel, for a fixed number of steps. By executing the attribute order, result tuples may get added into the result set (R). Also, executing an attribute order yields reward values, representing execution progress per time unit. Those reward values are used to update statistics, maintained internally by the reinforcement learning optimizer, to guide attribute order selections in future iterations. Once all input is processed, the algorithm returns the full set of result tuples.

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 steps until its computational budget is depleted. First, it retrieves an unprocessed cube, the target cube, from the task manager. Then, it uses a sub-function (an anytime version of the LFTJ) to process the retrieved target cube. 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 (they will be excluded from further consideration). Also, a reward value is calculated that represents progress towards generating a full join result. 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.

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 [37], a classical, worst-case optimal join algorithm. 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 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. 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 Line 8, Algorithm 3 creates an iterator over values for the current attribute that appear in all input relations (as indicated by the expression "matching values"). It focuses only on attribute values 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. More precisely, those data structures allow ADOPT to efficiently

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
2:  Effect: Iterates over attribute values and possibly adds results to R
3:  procedure JoinOneCubeRec(q, b, o, c, R, a)
4:      if a ≥ |q.A| then // Check for completed result tuples
5:          Insert tuple with current attribute values into R
6:      else
7:          // Initialize value iterator (do not evaluate it!)
8:          V ← iterator over matching values for o_a in [c.l_{o_a}, c.u_{o_a}]
9:          // Iterate over values until timeout
10:         for v ∈ V do
11:             // Select values for remaining attributes
12:             JoinOneCubeRec(q, l, o, c, R, a + 1)
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)
25:     // Retrieve state from JoinOneCubeRec invocation
26:     s ← Number of computational steps spent
27:     v ← Vector s.t. v_a is last value considered for attribute o_a
28:     // Calculate processed cubes
29:     P ← ∅
30:     for 0 ≤ a < |q.A| do
31:         Create new cube p s.t.
32:             ∀i < a : p_i = [v_i, v_i];
33:                 p_a = [c.l_{o_a}, v_a);
34:                     ∀a < i : p_i = [c.l_{o_i}, c.u_{0_i}]
35:         P ← P ∪ {p}
36:     end for
37:     return ⟨P, s⟩
38: end function
```

intersect values from different relations for the same join attribute. We refer to the original publication for a detailed discussion [37]. Note that ADOPT, instead of creating all potentially required data structures a-priori (which depend on the attribute order), it creates them on-demand, only if required for executing a new order.

Join processing via Procedure JoinOneCube terminates once the computational budget is depleted, 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

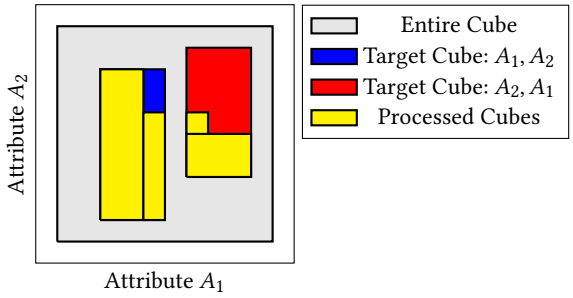

Figure 3: Illustration of containment relationships between hypercubes when processing a query with two attributes.

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 $[p.l_i, p.u_i] = [v_i, v_i]$).

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 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, then their value ranges. 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. To do so, it uses the attribute with maximal value domain, dividing its range equally across threads.

Whenever a worker threads requests a cube to work on, a randomly selected cube from the set of unprocessed cubes is returned. 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).

Algorithm 4 Managing cubes representing unprocessed join input.

```
1:  U ← ∅ // 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:      A ← attributes that appear in q in equality join conditions
6:      [l_a, u_a] ← attribute value ranges for all attributes a ∈ A
7:      // Identify attribute with largest value domain
8:      a* ← arg max_{a∈A} (u_a − l_a)
9:      // Use full value range for all but that attribute
10:     f ← ⨉_{a∈A:a≠a*} [l_a, u_a]
11:     // Divide largest value domain into per-thread ranges
12:     δ ← (u_{a*} − l_{a*})/n
13:     // Form one unprocessed cube per thread
14:     U ← {f × [l_{a*} + i · δ, l_{a*} + (i + 1) · δ | 0 ≤ i < n]}
15: end procedure

16: Output: Returns an unprocessed hypercube.
17: function TM.Retrieve
18:     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:     // Subtract target cube from unprocessed cubes
24:     U ← U \ c
25:     // Add complement of processed cubes as unprocessed
26:     for p ∈ P do
27:         // Get dimensions where p fully covers c
28:         F ← indexes i s.t. p.l_i = c.l_i and p.u_i = c.u_i
29:         // Get dimensions where p's bounds collapse
30:         S ← indexes i s.t. p.l_i = p.u_i
31:         // Get single remaining dimension
32:         d ← single remaining dimension not in F or S
33:         Create new cube u s.t.
34:             u_d = (p.u_d, c.u_d]; ∀f ∈ F : u_f = p_f; ∀s ∈ S : u_s = p_s
35:         // Add newly created cube to unprocessed cubes
36:         if u is not empty then
37:             U ← U ∪ {u}
38:         end if
39:     end for
40: end procedure

41: Output: True iff no unprocessed cubes are left.
42: function TM.Finished
43:     return true iff U = ∅
44: end function
```

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 (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

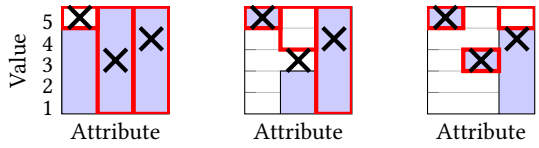

Figure 4: Illustrating cube processing in Example 3.2: The initial target cube $([1,5],[1,5],[1,5])$ is processed up to $(5,3,4)$ (marked by X). Processed cubes are represented by blue rectangles, complementary unprocessed cubes by red rectangles.

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. 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.

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 [15] 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

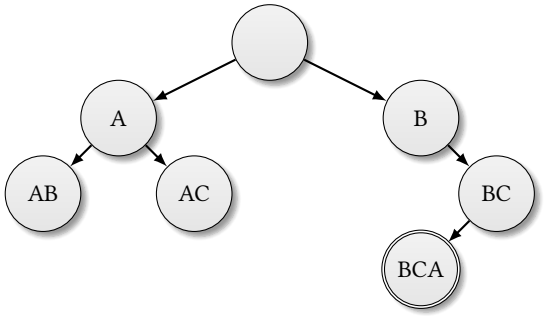

Figure 5: UCT search tree for a query with three attributes.

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 [33] 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 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 [15]. After selecting a path to a leaf and calculating the associated reward, the UCT algorithm updates confidence bounds for each node on that path.

In our scenario, it is crucial to avoid generating the entire search tree at once, as this may cause non-negligible overheads. The number of possible attribute orders for a given query can be very large. Instead, we use a UCT variant that builds the search tree gradually. Specifically, starting from a tree containing only the root node, the algorithm expands the search tree by at most one node per sample. If the algorithm reaches an MDP state that has no associated node in the search tree, it performs random transitions until reaching an end state. In our scenario, this means that the remaining attributes are selected in random order. The resulting reward value corresponds to a uniform random sample for rewards in the corresponding sub-tree. If the algorithm encounters states with no associated tree nodes, it creates the first "missing" nodes and adds it to the tree. As time progresses, the tree becomes most refined along paths associated with interesting attribute orders.

Example 3.3. Figure 5 shows an example for an UCT search tree, referring to a query with three attributes (called A, B, and C in the figure). Each node is labeled with the associated attribute prefix. Edges connect prefixes that can be reached by a one-attribute extension. Note that the UCT tree is not fully built. This is realistic, as the tree is only expanded by at most one node per sample.

3.5 Estimating Order Quality

The reinforcement learning, described in Section 3.4, is guided by reward values. Those values serve as quality samples, judging the quality of an attribute order when processing one specific partition of the data. While quality estimates may vary for the same order, across different invocations, ADOPT converges to the order with maximal average quality over time.

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:

$$Volume(c) = \prod_i (c.u_i - c.l_i) \quad (1)$$

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. 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):

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

4 ANALYSIS

In this section, we prove that ADOPT converges to optimal attribute orders. Two other properties of ADOPT, correctness and worst-case optimality, are analyzed in the appendix of our technical report [1]. First, we show that ADOPT must finish processing once 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 attribute value cube covered, scaled to the size of the full cube. Hence, accumulating a reward sum of one means that a volume equal to the entire query 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 entire query 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, processing finishes once accumulated rewards reach one (according to Theorem 4.1). Therefore, the average reward r_o per episode for an order 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. According to Theorem 4.2, the order with highest average reward is, at the same time, the fastest one to process. However, the UCT algorithm used by ADOPT, converges to a solution with maximal expected reward [15]. Hence, ADOPT converges to an attribute order minimizing 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 public join order [11] and graph data [17, 26] 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. 5.6); hypercube data decomposition (Sec. 5.4); and domain parallelism (Sec. 5.7).

[1] https://github.com/jxiw/ADOPT/blob/main/report/ADOPT.pdf

Table 1: Graphs used in the experiments.

Graph	#Vertices	#Edges
ego-Facebook	4,039	88,234
ego-Twitter	81,306	2,420,766
soc-Pokec	1,632,803	30,622,564
soc-LiveJournal1	4,847,571	68,993,7732

5.1 Experimental Setup

We benchmark the query engines on acyclic and cyclic queries.

Benchmark for acyclic queries. The join order benchmark (JOB) [11] 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.

Benchmark for cyclic queries. We follow prior work on benchmarking worst-case optimal join algorithms against traditional join plans [26] and consider the evaluation of clique and cycle queries over the binary edge relations of four graph datasets from the SNAP network collection [17]. Table 1 describes these datasets. 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 [7] (Database Server Toolkit v11.39.7, Oct2020-SP1) and PostgreSQL 10.21 [32] that employ traditional join plans; a commercial engine System-X (implemented in C++) that uses the worst-case optimal LFTJ algorithm [37]; the open-source engine EmptyHeaded that uses a worst-case optimal join algorithm [1]; and SkinnerDB [35] (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

Table 2: 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. ADOPT outperforms its competitors on all graph benchmarks by 2-30x. For the JOB benchmark, ADOPT is close (0.91x) to MonetDB and outperforms the others by 1.4-6.3x.

Systems	JOB	ego-Facebook	ego-Twitter	soc-Pokec	soc-Livejournal1
ADOPT	45	**4,414**	**3,931**	**9,268**	**26,350**
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)
MonetDB	**41** (0.91x)	> 66,165 (14.99x)	> 86,596 (22.03x)	> 59,131 (7.23x)	> 96,222 (3.84x)
SkinnerDB	65 (1.44x)	> 69,366 (15.71x)	> 129,741 (33.00x)	> 95,374 (10.29x)	> 101,392 (3.85x)

disk. ADOPT, EmptyHeaded, MonetDB, SkinnerDB, and System-X were set to run in memory. By default, all engines use 64 threads. Each system incurs overhead for ingesting the data into their own internal format. ADOPT stores data as sorted index arrays and this preparation has a low overhead (up to 60 seconds) and is not reported. Its memory overhead is also very low: Besides the data arrays, each hypercube needs a start and end value index per attribute. Each thread either consumes an entire hypercube or returns up to n disjoint hypercubes (for n join attributes). EmptyHeaded takes more than five days to construct indices for non-binary relations.

5.2 Runtime Performance

Hypothesis 1. *ADOPT demonstrates comparable or better performance than its competitors for both cyclic and acyclic queries.*

Table 2 reports the total time in seconds for different systems and benchmarks. In the join order benchmark (JOB), MonetDB is the fastest, followed very closely by ADOPT. This shows that ADOPT is competitive with the fastest competitor that employs traditional query plans for acyclic queries. For System-X, Table 2 only reports the overall runtime for 39 out of the 113 queries: These queries do not have IS/NOT NULL and IN predicates as they are not supported by System-X. ADOPT thus outperforms System-X by at least six times. 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.

For the graph benchmarks, ADOPT outperforms its competitors. MonetDB, PostgreSQL, and SkinnerDB, which use traditional query plans, perform consistently worse on cyclic queries than ADOPT, EmptyHeaded, and System-X, which use worst-case optimal join algorithms. This follows the asymptotic complexity gap and confirms the reported runtime gap between traditional query plans and worst-case optimal plans for cyclic queries [26]. Yet ADOPT outperforms EmptyHeaded and System-X by at least 1.5x on ego-Facebook and 2.5x on the other graphs. We also benchmarked the BAO learned query optimization for PostgreSQL [20] on ego-Twitter graph. There was no noteworthy runtime improvement.

Hypothesis 2. *ADOPT outperforms competitors for large queries.*

Figure 6 reports the runtime for: n-cliques on the ego-Facebook and ego-Twitter graphs (top) and respectively on the soc-Pokec and soc-Livejournal1 graphs (middle), and for n-cycles on the four graphs (bottom). The red top line in each figure represents the timeout (six hours = 21,600 seconds). For small cliques ($n = 3$), System-X is the best; it uses sensitivity indices to speed up the leapfrogging that intersects sorted lists in LFTJ. For larger cliques, ADOPT significantly outperforms its competitors. The reason is the much larger search space for a good (attribute or join) order. It is therefore much less likely to find a good order for the competitors in contrast to ADOPT, which tries many orders during query execution. For instance, for 6- and 7-clique queries, ADOPT is at least 2x faster than the runner-up system for all graphs. A similar behavior can be observed for n-cycle queries in Figure 6. Notably, ADOPT is the only system to complete 6-cycle on soc-Pockec and 5-cycle on Livejournal1 before timeout.

EmptyHeaded and System-X are the two competitors that use worst-case optimal join algorithms and outperform the other systems on the graph benchmarks. In the following experiments, we give a more detailed comparison of ADOPT against System-X on the graph benchmarks. One reason for choosing System-X over EmptyHeaded is that both ADOPT and System-X use LFTJ as the underlying join algorithm. Furthermore, the performance of these two competitors on the graph benchmarks is very close (within 4.8% of each other). Finally, the query language of System-X supports unary predicates, which are needed for the robustness experiment in Section 5.3.

Figure 7 examines the relative performance of ADOPT over System-X. The speedup of ADOPT over System-X increases with the query parameter n; System-X times out for large n. This speedup reaches: 4x for both 5- and 6-clique on ego-Twitter and soc-Pokec; 8x for 5- and 6-clique on soc-Livejournal1; 2x for 5-cycle on ego-Twitter and 4x on both soc-Pokec and soc-Liverjournal1. A reason for this speedup increase is the difficulty of System-X's optimizer to pick a good attribute order for increasingly larger queries. The average performance of the attribute orders used by ADOPT is better than the one attribute order of System-X.

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

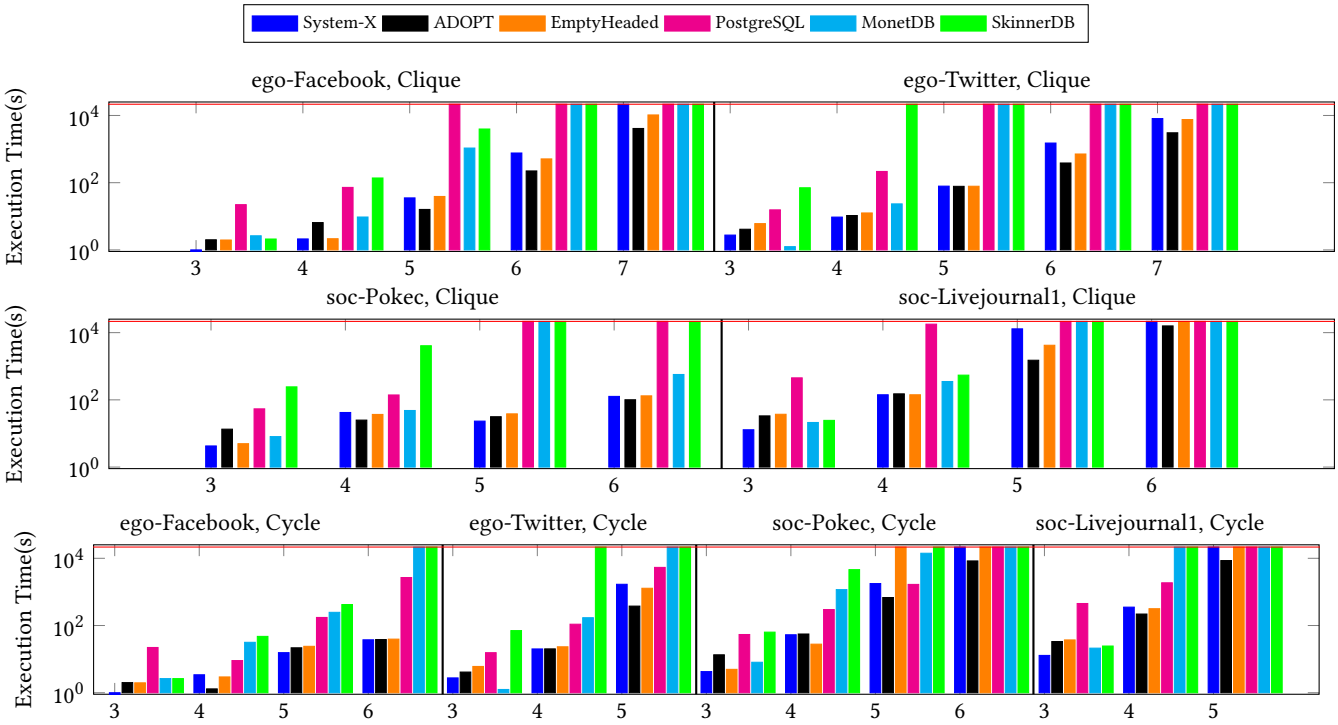

Figure 6: Execution (wall-clock) time for clique and cycle queries on four graphs (x axis represents query size).

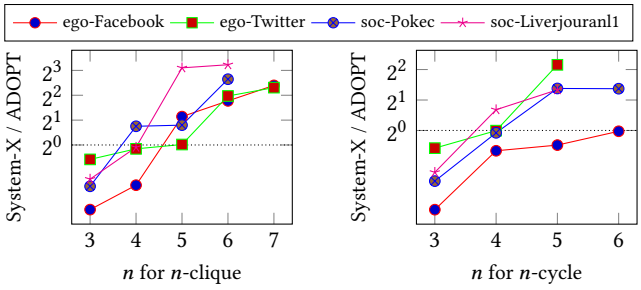

Figure 7: Relative speedup of ADOPT over System-X.

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

Figure 8 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

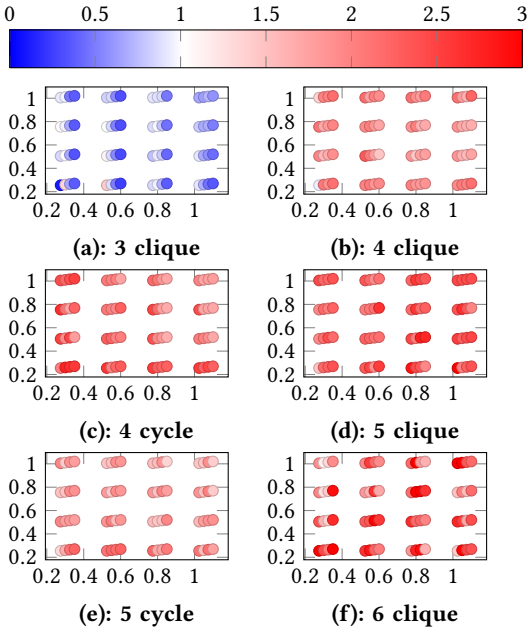

Figure 8: Speedup of ADOPT over System-X when varying the selectivity of newly added unary predicates on three randomly chosen attributes (along the x-axis, y-axis, and the circles for an (x,y)-point). More intense red (blue) means higher (lower) speedup. All queries are executed on ego-Twitter.

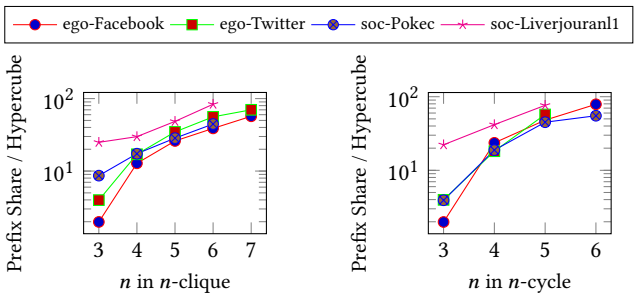

Figure 9: Speedup of using our hypercube approach versus using prefix+offset share progress tracker in ADOPT.

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 [34].

Hypothesis 4. *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 9 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 5. *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

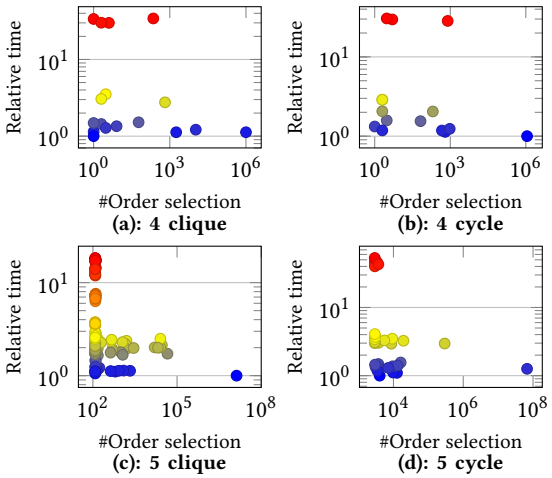

Figure 10: Selections of orders with different quality on ego-Twitter.

Table 3: Execution time (seconds) of clique and cycle queries on on ego-Twitter for ADOPT (1st column) and LFTJ with optimal attribute order (2nd column). Average runtime of LFTJ over all attribute orders (3rd column). Relative speedup of OPT over ADOPT (4th 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

we select and their quality relative to the optimal orders (i.e., with lowest execution time) in Figure 10. 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 3 compares ADOPT and LFTJ with an optimal attribute order: The runtime gap decreases from 2.52x for 3-clique/cycle to 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 3 also shows that ADOPT takes significantly less time than the average runtime of LFTJ over all attribute orders.

5.6 Sorting and synchronization overhead

Hypothesis 6. *The times required by ADOPT for sorting and thread synchonization are small relative to the total execution time.*

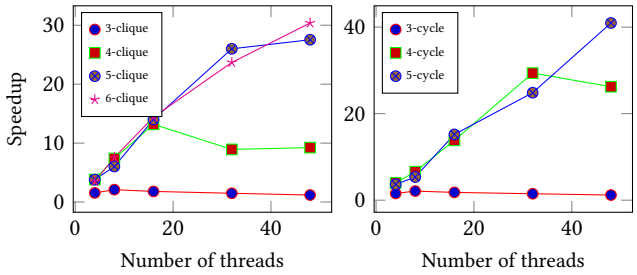

Figure 11: Speedup of multi-threaded ADOPT over single-threaded ADOPT for clique and cycle queries on ego-Twitter.

Whenever ADOPT starts a new episode with an attribute order different from the order used in the previous episodes, it might need to re-sort the tables so as to support the intersection of sorted lists as required by the leapfrog triejoin. In our experiments, the time for sorting only accounts for at most 2.5% of the total execution time of ADOPT for all queries. Sorting has thus negligible overhead, even though ADOPT frequently switches orders.

At the start of a new episode, ADOPT assigns a thread to handle an unprocessed hypercube. This process involves some synchronization overhead to manage the hypercubes. The time required for synchronization is at most 7% of the total execution time.

5.7 Parallelization

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

Figure 11 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 [31]. 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 [11, 18]. Our experiments demonstrate that it appears in the context of worst-case optimal join algorithms as well.

Adaptive processing [5, 8, 29, 36] 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 has focused on stream data processing [5, 8, 29, 36] (where query execution times are assumed to be longer), adaptive processing has recently also gained traction for classical query processing [22, 35]. SkinnerDB [35] is the closest in spirit to ADOPT: they both use the reinforcement learning algorithm UCT [15]. However, unlike SkinnerDB, ADOPT achieves worst-case optimal join processing, using a combination of specialized join algorithms, novel data structures, and specialized learning methods.

There is strong theoretical evidence that using several execution strategies for different parts of the data can lead to asymptotically lower complexity than using only one execution strategy for the entire input data [21]: The logical data partitioning put forward in this line of theoretical work is carefully crafted to observe the degree constraints in the data (e.g., whether the number of tuples with a certain value for an attribute exceeds a threshold). To achieve asymptotically low complexity, data partitioning and execution strategies are interleaved. PANDA [14] is the first algorithm to employ this theoretical approach, yet it is not implemented and its runtime depends on large constant factors. In contrast, ADOPT picks execution strategies (as variable orders) at random and logically partitions the data without observing degrees. This is simple and effective, as exploiting data degree constraints and synthesizing execution strategies tailored at them may take non-trivial time.

Our work uses reinforcement learning to select attribute orders. It relates to recent works that employ learning for query optimization [16, 19, 20, 38]. 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 [4], 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 [10] and for succinct factorized representations of query results [28]. The latest development extends this line of work with data degree constraints and histograms [23]. Classical approaches to query optimization based on heuristics [9] and data statistics [1, 3] 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 [6] and functional aggregate queries [13]. 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 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 arXiv:1503.02368
[2] Molham Aref. 2019. Relational Artificial Intelligence. In *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019 (CEUR Workshop Proceedings)*, Mario Alviano and Andreas Pieris (Eds.), Vol. 2368. CEUR-WS.org, 1. http://ceur-ws.org/Vol-2368/invited1.pdf
[3] 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
[4] 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
[5] Ron Avnur and Jm Hellerstein. 2000. Eddies: continuously adaptive query processing. In *SIGMOD*. 261–272. https://doi.org/10.1145/342009.335420
[6] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Zavodny. 2013. Aggregation and Ordering in Factorised Databases. *Proc. VLDB Endow.* 6, 14 (2013), 1990–2001. https://doi.org/10.14778/2556549.2556579
[7] Peter A Boncz, Martin L Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85.
[8] Amol Deshpande. 2004. An initial study of overheads of eddies. *SIGMOD Record* 33, 1 (2004), 44–49. https://doi.org/10.1145/974121.974129
[9] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting worst-case optimal joins in relational database systems. *Proceedings of the VLDB Endowment* 13, 11 (2020), 1891–1904. https://doi.org/10.14778/3407790.3407797
[10] 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
[11] Andrey Gubichev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015), 204–215.
[12] Guodong Jin, Nafisa Anzum, and Semih Salihoglu. 2022. GRainDB: A Relational-core Graph-Relational DBMS. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p57-jin.pdf
[13] 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
[14] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2017. What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts (Eds.). ACM, 429–444. https://doi.org/10.1145/3034786.3056105
[15] Levente Kocsis and C Szepesvári. 2006. Bandit based monte-carlo planning. In *European Conf. on Machine Learning*. 282–293. http://www.springerlink.com/index/D232253353517276.pdf
[16] 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. arXiv:1808.03196 http://arxiv.org/abs/1808.03196
[17] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[18] Guy Lohman. 2014. Is query optimization a "solved" problem? *SIGMOD Blog* (2014).
[19] 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 arXiv:1904.03711
[20] 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 *ACM SIGMOD Record*, Vol. 51. 5. https://doi.org/10.1145/3542700.3542702
[21] Dániel Marx. 2013. Tractable Hypergraph Properties for Constraint Satisfaction and Conjunctive Queries. *J. ACM* 60, 6 (2013), 42:1–42:51. https://doi.org/10.1145/2535926
[22] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. 2020. Permutable compiled queries: Dynamically adapting compiled queries without recompiling. *Proceedings of the VLDB Endowment* 14, 2 (2020), 101–113. https://doi.org/10.14778/3425879.3425882
[23] Hung Q. Ngo. 2022. On an Information Theoretic Approach to Cardinality Estimation (Invited Talk). In *25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference) (LIPIcs)*, Dan Olteanu and Nils Vortmeier (Eds.), Vol. 220. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:21. https://doi.org/10.4230/LIPIcs.ICDT.2022.1
[24] 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
[25] 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
[26] 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 *Proceedings of the Third International Workshop on Graph Data Management Experiences and Systems, GRADES 2015, Melbourne, VIC, Australia, May 31 - June 4, 2015*, Josep Lluís Larriba-Pey and Theodore L. Willke (Eds.). ACM, 2:1–2:8. https://doi.org/10.1145/2764947.2764948
[27] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Rec.* 45, 2 (2016), 5–16. https://doi.org/10.1145/3003665.3003667
[28] 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
[29] 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
[30] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning Linear Regression Models over Factorized Joins. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 3–18. https://doi.org/10.1145/2882903.2882939
[31] PG G Selinger, MM 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. http://dl.acm.org/citation.cfm?id=582095.582099
[32] Michael Stonebraker and Lawrence A Rowe. 1986. The design of Postgres. *ACM Sigmod Record* 15, 2 (1986), 340–355.
[33] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement learning, second edition: An introduction*. 532 pages. https://doi.org/10.1016/s1364-6613(99)01331-5 arXiv:1603.02199
[34] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data*. 1153–1170.
[35] 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 Transactions on Database Systems* 46, 3 (2021). https://doi.org/10.1145/3464389
[36] Kostas Tzoumas, Timos Sellis, and Christian S Jensen. 2008. *A reinforcement learning approach for adaptive query processing*. Technical Report.
[37] Todd L. Veldhuizen. 2014. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 96–106. https://doi.org/10.5441/002/icdt.2014.13
[38] 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

A CORRECTNESS

We show that ADOPT generates a complete and correct result.

Theorem A.1. *ADOPT does not produce incorrect join results.*

Proof. ADOPT inserts join results in Line 5 of Algorithm 3. For each attribute, Algorithm 3 only iterates over values that appear in all relations with that attribute (Line 10 in Algorithm 3). Hence, join results must satisfy all equality join conditions. Furthermore, Algorithm 3 is only applied to input tuples satisfying all unary predicates (due to the filter in Line 5 in Algorithm 1). Hence, join results satisfy all applicable predicates and are correct. □

Lemma A.2. *All results contained within processed cubes, returned by Algorithm 3, have been inserted into the result set.*

Proof. Assume there is a vector r of join attribute values, matching all join predicates, that is contained in a processed cube p but not in the join result. There is an attribute a such that r equals the last selected attribute values v up to attribute a (in attribute order), then takes a value below the last selected value for the next attribute. However, Algorithm 3 does not advance from one value to

the next for an attribute, before considering all value combinations for the remaining attributes. Hence, r must have been added to the result, leading to a contradiction. □

Lemma A.3. *The task manager only removes processed cubes.*

Proof. In each invocation of TS.Remove, the task manager removes only the target cube. Assume a vector l of attribute values, within the target cube, is "lost", i.e. it is neither contained in any processed cube nor in any of the newly added, unprocessed cubes. Denote by v the last selected values for all attributes in the join invocation, immediately preceding removal, and by o the corresponding attribute order. Assume l matches the values in v for some prefix (possibly of size zero) of order o. Denote by a the first attribute in o for which l does not match v. Denote by p the processed cube added when reaching a in the loop from Line 30 in Algorithm 3. If $l_a < v_a$ then l must be contained in p. However, if $l_a > v_a$, l must be contained in the unprocessed cube added when reaching p in the loop from Line 26 in Algorithm 4, leading to a contradiction. □

Theorem A.4. *ADOPT produces a complete join result.*

Proof. The join phase terminates only once no unprocessed cubes are left. Result tuples contained in processed cubes are inserted into the result set (Lemma A.2) and no unprocessed cubes are erroneously removed (Lemma A.3). Hence, processing cannot terminate before all result tuples are inserted. □

The next result follows immediately.

Corollary A.5. *ADOPT produces a correct join result.*

Proof. ADOPT produces all correct join results (Theorem A.4) without generating any incorrect tuples (Theorem A.1). □

B WORST-CASE OPTIMALITY

We analyze whether ADOPT maintains worst-case optimality guarantees. We focus on join processing overheads, neglecting without loss of generality the preparation overheads. These overheads include the sorting of the relations to support LFTJ leapfrogging following the orders of attributes picked by ADOPT. We consider the number of threads is constant. Additionally, our analysis is based on the following assumption.

Assumption 1. *We assume that the number of episodes is bounded by a constant that does not depend on the data size.*

The latter assumption can be ensured by increasing the number of steps per episode, proportional to the maximal output size.

Lemma B.1. *The number of cubes processed by ADOPT does not depend on the data size.*

Proof. Initially, the number of unprocessed cubes is proportional to the number of threads (i.e., constant). Each invocation of Procedure CS.Remove may create up to m cubes where m is the number of query attributes (i.e., a constant). Each episode may process multiple cubes, i.e. invoke that function multiple times. However, whenever the target cube was fully processed, no unprocessed cubes are added. There can be at most one target cube per episode and thread that was not fully processed. Hence, the number of unprocessed cubes added per episode is bounded by a constant. Due to Assumption 1, the number of generated (and processed) cubes is therefore bounded by a constant as well. □

Lemma B.2. *Time complexity of ADOPT's join phase is dominated by the complexity of the function JoinOneCube.*

Proof. The per-episode complexity of all operations of the reinforcement learning algorithm are bounded by the number of join attributes. Similarly, the number of operations required to retrieve or remove cubes is bounded by the number of join attributes. Hence, the time complexity for join processing dominates. □

We are now ready to prove our main result.

Theorem B.3. *ADOPT is worst-case optimal.*

Proof. The number of cubes processed by ADOPT does not depend on the input data size (Lemma B.1). Furthermore, time complexity for joins dominates (Lemma B.2). ADOPT uses the LFTJ algorithm to process cubes. This algorithm is worst-case optimal [37]. The time for processing the largest cube, which occurs over the execution of a query, is upper-bounded by the time required by LFTJ for processing the entire query cube. The number of cubes is independent of the data size. Hence, total processing overheads are asymptotically equivalent to the time required by LFTJ, therefore worst-case optimal. □