Dear VLDB Associate Editor, Dear Reviewers:

We thank you for your insightful comments that have helped us to improve the paper significantly! We improved the presentation, expanded our implementation to handle new benchmarks such as TPC-H and JCC-H, and included various new experiments. As suggested by Reviewer 5 and due to lack of space, we included some of the new content in an extended technical report [42], available at https://github.com/jxiw/ADOPT/blob/main/report/ADOPT.pdf. Compared to our original submission, this report adds roughly eight pages of new material in the appendix (including experiments, examples, and explanations). We believe that the revised version addresses all points raised by the reviewers.

Meta Reviews

D1. Addition of the new experiments e.g., new workloads and benchmarks, scalability of number of attributes, incurred overhead.

We added three types of experiments: (1) Runtime experiments for all engines for two new benchmarks (TPC-H, JCC-H in Table 1); (2) Experiments to verify the scalability of the engines as we increase the number of attributes (Appendix H in the extended technical report [42]); Experiments to measure different types of overheads, such as memory use, index creation time, and synchronization overheads (Appendix E, F, and G in the extended technical report).

D2. Clarity of the algorithm and justification of the decision choices (i.e. expansion of the search tree).

We clarified the description of the algorithms in Section 3. In particular, we explain the expansion of the search tree in much more detail, supported by a corresponding example (Example 3.3) and by Figure 5, now showing the episodes at which specific search tree nodes were added (red labels). Also, we added detailed explanations of the leapfrog triejoin (Appendix I in our extended technical report [42]) and the specific implementation used in ADOPT (Appendix J).

D3. Discuss the challenges of the approach and provide a clear motivating example.

We added a motivating example in the introduction (Example 1.2), outlining a specific domain where clique queries with elevated number of joins are common. Also, we discuss the specific challenges of our scenario and how these motivate design decisions of ADOPT that are different from prior work in Section 6.

Review 5 Details

D1. Overall the paper is very easy to read and to understand. There is however an important part of the paper, which has to do with the way that ADOPT expands the search tree, which is not clear at all. Looking at Figure 5, there are the following case that the text should describe: Why are both AB and AC created, instead of expanding (for example) AB to ABC? Is there a priority to expand a node, where not all of his children have been created, rather than expanding a child of that node? If from A we had created AB, do we always expand all children of A, before expanding AB? If yes, why? If no, please explain the criterion for expanding a node better.

Thanks for your suggestion. We now explain search tree expansion in much more detail in Section 3.4. Also, the example in this section (Example 3.3), along with the associated figure (Figure 5), have been expanded to show which tree nodes are added in which episodes, given a specific sequence of selected attribute orders. In short: please note, first of all, that tree nodes are labeled with attribute orders (not attribute sets), transitions append one attribute. Expanding AB to ABC and expanding AC to ACB is not equivalent as it represents two different attribute orders with possibly different performance properties. Which tree nodes are expanded depends on the attribute orders selected by the UCT algorithm. Given a selected attribute order, corresponding to a path from root to leaf in the fully expanded tree, the UCT algorithm adds the first “missing” node on that path in the partially expanded tree. In doing so, over time, the UCT algorithm collects more reward statistics (associated with nodes) for parts of the search space that are frequently visited.

D2. The number of attributes of each joined table at the experiments seems small. Which is the maximum number of join attributes at a table? What happens at the relative performance of the algorithms when this number increases? If there is insufficient space available, some experiments could be added at a full version of the paper, which will be made available to the reviewers.

Our experiments in Section 5 feature benchmarks with up to four join attributes per table (e.g., queries 19c in the join order benchmark). Also, we added experiments in Appendix H (Figure 12) of our extended technical report [42], reporting results for the Loomis-Whitney queries that join n tables, where each table has $n - 1$ join attributes. The experiment shows that ADOPT clearly outperforms its competitors for the evaluation of Loomis Whitney queries over the ego-Twitter graph. As we increase n from 3 to 5, the performance gap of ADOPT relative to its competitors increases as well.

D3. At the experiments, there is a lot of emphasis when ADOPT outperforms other algorithms, but not much discussion when the reverse happens. For example, at Figure 7 there are cases where ADOPT is 4 times slower than a competitor. I would assume that queries with smaller "n" are more common than queries with large "n", so the paper needs to provide additional motivation that n-cycle or n-click queries with large n are common and useful in applications.

We added a new example in the introduction (Example 1.2), outlining concrete use cases in the context of social networks where large graph queries need to be processed. Also, in the field of bioinformatics, protein-protein interaction networks represent the physical interactions between proteins in a cell. Finding large k-cliques in these networks can help identify dense regions of proteins that interact closely with each other. E.g., in [43], they analyze protein network with sizes from $k = 4$ to $k = 6$. Besides graphs, one of our newly added benchmarks (JCC-H) shows that ADOPT performs generally well if data is sufficiently correlated.

Besides that, we discuss cases where ADOPT is slower than competitors in much more detail in Section 5.2 (e.g., Hypothesis 3 and the following discussion).

D4. At the introduction it is not entirely clear why a new technique could improve upon prior algorithms. There is a key phrase: "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." but this should be expanded.

We expanded the corresponding explanation in the introduction. The statement refers to the issue referenced, e.g., in the paper by Veldhuizen: "Choosing a good variable ordering is crucial for performance, in practice, but immaterial for the worst-case complexity analysis presented here." [41]. I.e., an analysis of worst-case optimality does not need to consider the attribute order since, given pessimistic assumptions on the database content, all attribute orders are asymptotically equivalent. However, in practice, the attribute order choice matters a lot for performance.

D5. There are experiments where the aggregate function is the COUNT function, which is easier for ADOPT, but it would be nice to also have experiments with other aggregate functions.

We added experiments with TPC-H and JCC-H queries which feature a diverse set of aggregates (as well as many other SQL features). The queries of the join order benchmark also use the MIN aggregate function.

D6. Providing pseudocode of the used algorithms is great. It would be even better though if the text provided references to specific lines of the algorithms, in order to further help the reader.

We added references to specific lines in the text accompanying the algorithms in Section 3. Thanks for the suggestion!

D7. The Volume covered is used as a useful measure by the algorithm. The Volume covered could be larger if fewer join results are produced, or if there are fewer tuples in these subcubes. It would be nice to explain more why the chosen criterion is good in both cases.

We added corresponding explanations in Section 3.5. In short: the volume covered per time unit depends on the part of the data that is processed (e.g., due to a non-homogeneous density of result tuples and other factors). However, the best attribute order covers most volume per time unit *in average*, averaging over the entire data. The goal of ADOPT is to converge to the attribute order that works best in total, considering the entire data. Having varying volumes per time unit, even for a fixed attribute order, makes it harder to identify the optimal one. However, reinforcement learning algorithms like the UCT algorithm can deal with stochastic reward functions (i.e., action sequences are associated with a probability distribution over rewards, rather than a fixed value). Covering varying amounts of volume for a fixed order translates into a stochastic reward function from the perspective of the UCT algorithm. Despite that "noise", UCT converges to optimal decisions.

D8. "ADOPT stores data as sorted index arrays and this preparation has a low overhead (up to 60 seconds) and is not reported." It is not clear why this overhead is not reported, and why it is fair that this overhead is not reported.

We now report index creation overheads in Appendix F of the extended technical report [42]. For the results in Table 1, we do not count index creation overheads for any of the compared systems. We report index creation overheads separately from query evaluation overheads since index creation only happens once. If executing enough queries, the index creation overheads become negligible compared to query evaluation overheads.

D9. The paper mentions that ADOPT uses novel data structures, but it is not clear which data structures it refers to and why they are novel.

This statement refers to the data structure storing unprocessed cubes and enabling operations over them, in particular removal of partially processed cubes, the focus of Section 3.3. We clarified the meaning of "novel data structures" in Section 6.

Review 7 Details

W1. It is unclear how much of the performance improvement comes from ADOPT being a lean research prototype compared to full-fledged systems (e.g., Postgres) with significantly more functionality. It would be stronger to implement these techniques inside an existing system for a true apples-to-apples comparison.

The high-level approach of ADOPT comes with very specific requirements on the execution engine. For instance, these include support for high-frequency attribute order switching in a worst-case optimal join processing framework, detailed processing statistics collected at run time (to inform the reinforcement learning optimizer), infrastructure for bookkeeping to avoid redundant work etc. This makes it hard to integrate this approach into classical SQL processing engines, motivating the design of a customized system instead. We clarified these considerations at the beginning of Section 2.

W2. Space overhead is discussed (very) briefly in Section 5.1. It would be good to include a more detailed analysis of the space/memory consumption of each system in the experimental evaluation.

We report space and memory consumption in Tables 4 and 5 of our technical report. Table 4 shows that all systems require similar disk space. As shown in Table 5, ADOPT's memory usage is comparable to System-X (i.e., ADOPT's relative memory consumption is within a factor of 0.6 to 1.7, compared to System X), a commercial system using worst-case optimal joins. Systems using non worst-case optimal joins and non-adaptive processing (e.g., MonetDB) have often significantly higher memory consumption than ADOPT since they store large intermediate results in main memory, either due to intrinsic limitations of binary joins (producing large intermediate results for cyclic queries) or due to sub-optimal choices of the join order, leading to unnecessarily large intermediate results (for queries on highly skewed data, e.g., JCC-H).

W3. As the description in Sections 2 and 3.1 suggests, ADOPT seems to require the full materialization of intermediate results. Related to W2, how does this impact space overhead during query processing? Can the algorithm be adapted to enable pipelined execution?

Like LFTJ on which it is built, ADOPT does not require the materialization of intermediate results for join processing (intermediate results in the sense of tuples resulting from joining a subset of tables in a multiway join). Only the final results of a multiway join, i.e., results from the join between all tables, are stored (result set R in Algorithm 3). We clarified this in Section 2.

On the other hand, ADOPT applies unary predicates in a preprocessing stage and stores the resulting tables (which could be considered intermediate results). However, those tables tend to be much smaller than the original ones. Also, when processing complex queries using the new ADOPT version, ADOPT materializes the results of sub-queries in tables. We added clarifying comments on materialization in Section 2. Also, Appendix I in our extended technical report [42] describes the LFTJ join in detail whereas Appendix J contains details about the implementation in ADOPT. Finally, we analyze memory consumption in Appendix E.

W4. Section 3.3 states that ADOPT enables parallelism via equal-width range partitioning. Is this strategy guaranteed to give good/equal partitions, or is it just a simple heuristic?

Equal-width range partitioning is only used for the very first episode. At the end of each episode, defined by a timeout, threads return unprocessed parts of cubes they were processing to a common "pool" of unprocessed cubes (represented as variable U in Algorithm 4). Other threads can freely select cubes from that pool. This mechanism avoids potentially skewed processing overheads across threads (which would be possible with a static assignment from threads to cubes). We added a clarifying statement in Section 3.3.

D1. The notation in equations 1 and 2 is somewhat unclear. Volume(c) seems like it should be Volume(q), and similarly for Reward(P,q), since c is never used.

Thank you for pointing this out! You are right, this was a typo. We replaced Volume(c) by Volume(q) in that equation.

Review 8 Details

W1. For the proposed technique, the adaptive processing strategy is loosely connected with worst case optimal join algorithms. The discussed challenges in Section 1 (limiting overheads, metric to compare orders and how to select retribute orders) are mainly related to the adaptive processing. The reader would doubt that simply combining existing adaptive processing strategy (e.g., SkinnerDB) with worst case optimal join could lead to most of the claimed benefit.

W2. The novelty compared with the existing adaptive processing strategy (e.g., SkinnerDB) seems limited. They both adopt UCT to decide on query order for adaptive processing. The main difference is that ADOPT adopts worst-case optimal join. And ADOPT uses hypercube data decomposition while SkinnerDB uses an alternative design (shared prefix+offset progress tracker).

D1. Please discuss the challenges of combining the existing adaptive processing strategy with the worst-case optimal join. And analyze how ADOPT are costumed for the worst case optimal join algorithms.

Our goal in ADOPT is to combine guarantees on worst-case optimal processing with guarantees of converging to optimal attribute orders. To guarantee worst-case optimal joins in the context of adaptive processing (i.e., we repeatedly switch the attribute order), we first need to bound the amount of redundant work across attribute orders. Otherwise, even if using a worst-case optimal join for each attribute order, we may lose the guarantee of worst-case optimality due to redundant work when switching between orders. SkinnerDB reduces but does not completely avoid redundant work across join orders (it uses a tree-based data structure that shares progress between similar join orders but is less effective for dissimilar join orders). Such a data structure is therefore a bad basis for the formal guarantees that we want to give.

This motivated our work towards a completely different data structure with associated operators that avoids *any* redundant work when switching attribute orders. As you rightfully point out, this is one of the major novelties, compared to SkinnerDB. At the same time, this data structure can be used to avoid redundant work across different threads as well, enabling efficient parallelization. Besides that, ADOPT uses a different anytime join algorithm than SkinnerDB, and solves a different learning problem via reinforcement learning. The definition of states, actions, and the reward function differ from the ones used in SkinnerDB as ADOPT optimizes attribute (not join) orders, and measures progress (and reward) using volumes in the space of attribute value combinations (rather than the space of tuple combinations). We discuss those points in more detail in the related work section of the revised paper.

W3. Join Order Benchmark (JOB) is widely used for benchmarking join order selection, which is characterized with complex join (each query has 3 16 joins). However, ADOPT is inferior to a traditional engine, MonetDB on JOB workload. Would such a result imply that the adaptive processing or worst-case optimal join is only advantageous on graph workload with extremely large queries?

D2. I encourage the author to evaluate ADOPT on other workloads (e.g., TPC-H, TPC-DS) except the graph workload. And it would be meaningful to discuss why ADOPT performs worse than MonetDB that employs traditional query plans on the JOB workload.

We added experiments with the TPC-H and JCC-H benchmarks for scaling factor ten, as shown in Table 1. While both benchmarks have the same database schema, TPC-H uses synthetically generated data with a uniform distribution, while JCC-H uses highly skewed data. In the TPC-H benchmark, MonetDB outperforms ADOPT. This is consistent with prior work [1] showing that other systems using the LFTJ, the same join algorithm ADOPT is based upon, perform worse than MonetDB on TPC-H queries. On the other hand, on JCC-H, ADOPT is several orders of magnitude faster than MonetDB. Here, estimating sizes of intermediate results and, therefore, query planning is difficult due to highly skewed data. ADOPT wins due to adaptive processing. This shows that the combination of features implemented by ADOPT is advantageous beyond graph workloads.

On graph workloads with cyclic queries, ADOPT benefits from using an LFTJ variant, compared to systems implementing traditional, binary joins. Compared to other systems implementing LFTJ but no adaptive processing, ADOPT starts benefiting for clique and cycle queries joining an elevated number of tables that is however well within the range of query sizes that are used in practice (see our answer to D3 of Reviewer 5 as well as Example 1.2 for concrete examples). With the growing number of tables, taking into account correlations and therefore query planning becomes harder. We discuss the pros and cons of ADOPT's design and its impact on relative performance on various benchmarks in more detail in Section 5.2. Thank you for the suggestion!

D3. Adding a concrete example for worst-case optimal join algorithm could improve the readability of the paper for

the audience who are not familiar with the worst-case optimal join. What are the requirements for implementing the worst-case optimal join, compared with other common join algorithms?

We added detailed examples of the worst-case optimal join algorithm LFTJ on which ADOPT is based in Appendix I of the extended technical report [42], as well as implementation details about the LFTJ variant used by ADOPT in Appendix J.

D4. SkinnerD outperforms the traditional data engine (i.e., MonetDB and Postgres) greatly in its original paper. It would be better to explain the reason for the inconsistent performance.

It depends on the benchmark and scenario. The following discussion focuses on benchmarks that appear in prior papers about SkinnerDB and in the current publication.

In the original SIGMOD 2019 paper, SkinnerDB outperforms MonetDB on the join order benchmark (in terms of total execution time) if all engines are restricted to single-threaded evaluation (Table 1) but performs about 30% worse than MonetDB in a multi-threaded setting (Table 2). In the 2021 TODS paper, a re-implementation of SkinnerDB is compared to a newer version of MonetDB. Again, MonetDB performs better on the join order benchmark (Figure 12) when using parallel evaluation. This is consistent with the results in this paper where all compared engines use multithreading. As in the TODS 2021 paper, SkinnerDB performs better than Postgres on the join order benchmark. None of the prior papers reported results evaluating SkinnerDB on graph benchmarks.

For the newly added benchmarks TPC-H and JCC-H, the results are also consistent with prior publications (Figure 12 in the TODS 2021 version, albeit for a different scaling factor than in this publication). SkinnerDB performs significantly better than MonetDB and Postgres on JCC-H, due to highly correlated data which makes optimization hard, but performs worse than MonetDB on TPC-H where query optimization is relatively easy (making adaptive processing unnecessary).

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 [27, 41]. LeapFrog TrieJoin (LFTJ) is a prime example of a worst-case optimal join algorithm [41]. 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 [28]. 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 [30, 31], graph processing [2, 14] and general query processing [11], in-database machine learning [34], and in the commercial systems LogicBlox [4] and RelationalAI [3].

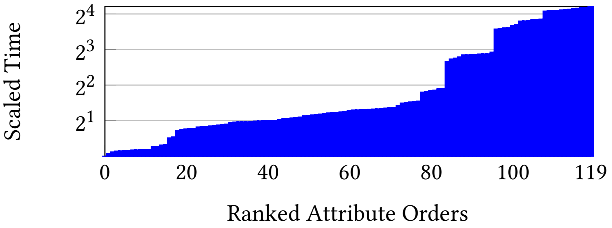


Figure 1: Execution time of different attribute orders for the five-clique query on the ego-Twitter graph.

As pointed out in prior work [41], 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 [41]. 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.

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

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 [21]. 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 [16, 18, 32]. Specifically, prior analysis often considers cliques of up to five or six [16, 32], or more [18] 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 [39], 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 [20, 29] 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

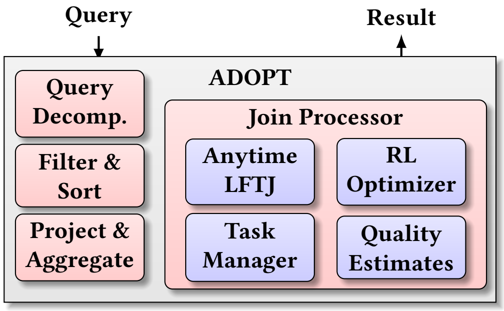


Figure 2: Overview of ADOPT system components.

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 [25, 38]. 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. Appendix I of our technical report [42] provides further details on the original LFTJ algorithm that ADOPT is based upon, including several examples. Also, Appendix J in the report discusses details on the LFTJ variant used for ADOPT. In particular, it discusses 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:     $\{R_1, \ldots, R_m\}$ ←PREP.UNARYFILTER($q$)
 6:     // Initialize join result set
 7:     $R \leftarrow \emptyset$
 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:         $o \leftarrow$ 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 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 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:     $r \leftarrow 0$
 6:     // Execute in parallel for all threads
 7:     for $1 \leq t \leq n$ in parallel do
 8:         // Initialize remaining cost budget
 9:         $l_t \leftarrow b$
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:             $\langle P_t, s_t \rangle$ ←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:             $r \leftarrow r + Reward(P_t, q)$
20:             // Update remaining budget
21:             $l_t \leftarrow l_t - s_t$
22:         end while
23:     end for
24:     // Return accumulated reward
25:     return $r$
26: end function
```

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 [41], 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 [42].

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
```

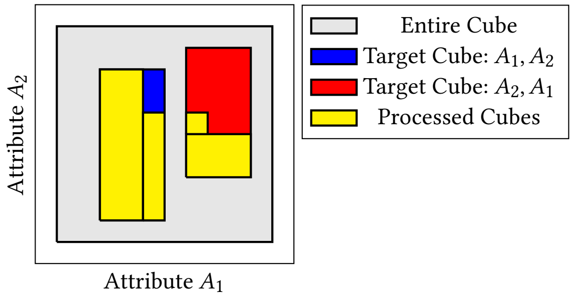


Figure 3: Illustration of containment relationships between hypercubes when processing a query with two attributes.

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 [41] to efficiently iterate over satisfying values. This mechanism is described in detail in Appendix I of the technical report [42]. 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.

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

16: Output: Returns an unprocessed hypercube.
17: function TM.Retrieve
18:     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:     $U \leftarrow U \setminus c$
25:     // Add complement of processed cubes as unprocessed
26:     for $p \in P$ do
27:         // Get dimensions where $p$ fully covers $c$
28:         $F \leftarrow$ 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:         $S \leftarrow$ indexes $i$ s.t. $p.l_i = p.u_i$
31:         // Get single remaining dimension
32:         $d \leftarrow$ single remaining dimension not in $F$ or $S$
33:         Create new cube $u$ s.t.
34:             $u_d = (p.u_d, c.u_d]$; $\forall f \in F : u_f = p_f$; $\forall s \in S : u_s = p_s$
35:         // Add newly created cube to unprocessed cubes
36:         if $u$ is not empty then
37:             $U \leftarrow U \cup \{u\}$
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 $U = \emptyset$
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.

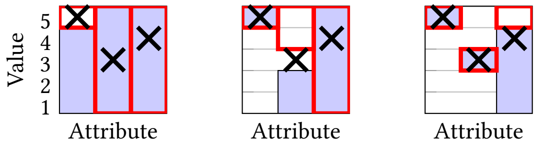


Figure 4: Illustrating cube processing in Example 3.2: The initial target cube $([1,5],[1,5],[1,5])$ **is processed up to** $(5,3,4)$ **(marked by X). Processed cubes are represented by blue rectangles, complementary unprocessed cubes by red rectangles.**

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 [17] 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 [37] 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.

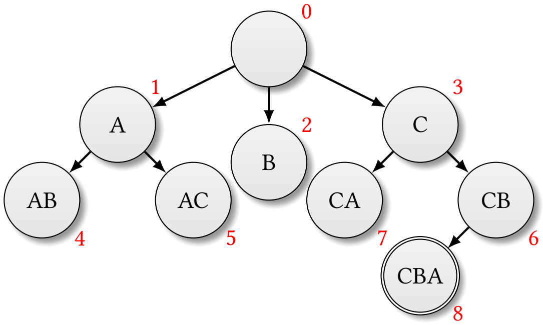


Figure 5: UCT search tree for a query with three attributes: nodes are labeled with partial attribute orders, transitions append one attribute. Red numbers next to nodes represent the episode number at which they are added when selecting attribute orders ABC, BCA, CBA, ABC, ACB, CBA, CAB, and CBA (in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

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 [17]. 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 [17]. 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 [42]. 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 [17]. 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 [20, 29] 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
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
SkinnerDB	65 (1.44x)	> 69,366 (15.71x)	> 129,741 (33.00x)	> 95,374 (10.29x)	> 101,392 (3.85x)	173 (1.23x)	320

lack of space, we defer to a technical report [42] 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 [29] and consider the evaluation of clique and cycle queries over the binary edge relations of four graph datasets from the SNAP network collection [20]. 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 [36] that employ traditional join plans; a commercial engine System-X (implemented in C++) that uses the worst-case optimal LFTJ algorithm [41]; the open-source engine EmptyHeaded that uses a worst-case optimal join algorithm [2]; and SkinnerDB [38] (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 Appendix F of the extended technical report [42]). 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 [38].

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 1.48x (1.14x) for 5-clique (5-cycle). This is remarkable, given that ADOPT tries out several attribute orders and switches between

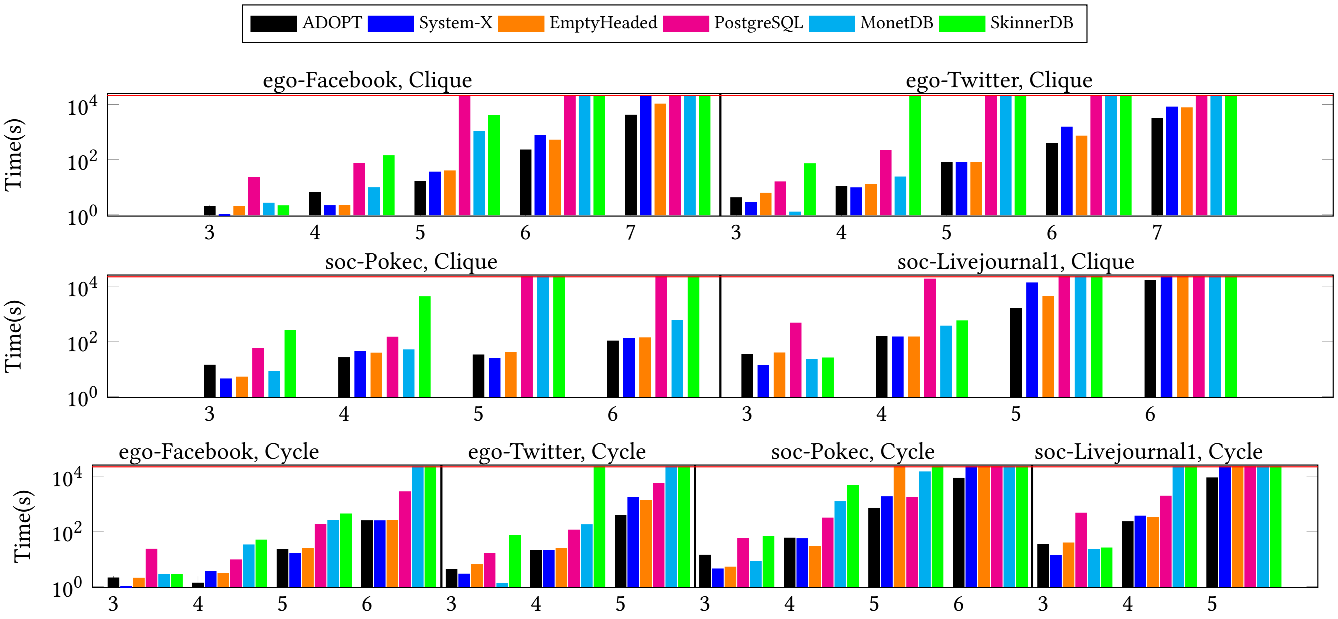
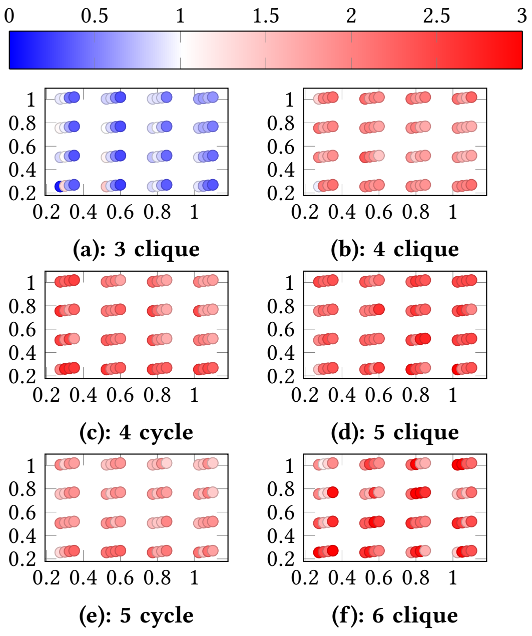


Figure 6: Execution (wall-clock) time for clique and cycle queries on four graphs (x axis represents query size).



(a): 3 clique

(b): 4 clique

(c): 4 cycle

(d): 5 clique

(e): 5 cycle

(f): 6 clique

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.

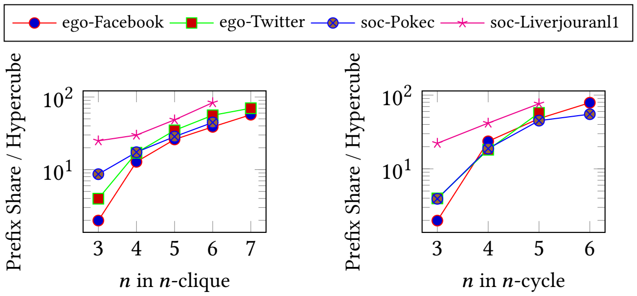
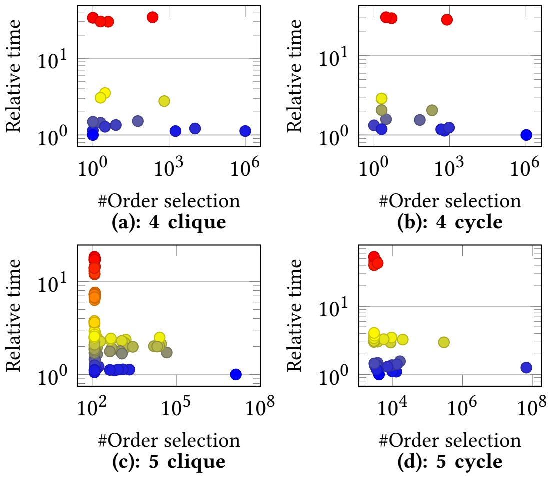


Figure 8: Speedup of using our hypercube approach versus using prefix+offset share progress tracker in ADOPT.



(a): 4 clique

(b): 4 cycle

(c): 5 clique

(d): 5 cycle

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

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.

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

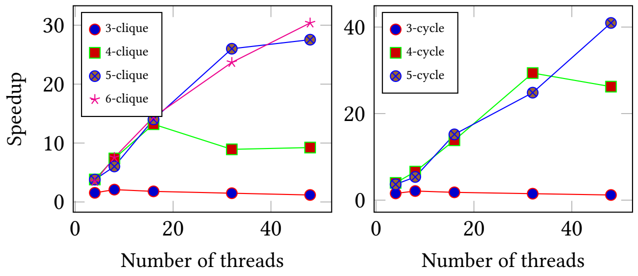


Figure 10: Speedup of multi-threaded ADOPT over single-threaded ADOPT for clique and cycle queries on ego-Twitter.

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 [35]. 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, 21]. Our experiments demonstrate that it appears in the context of worst-case optimal join algorithms as well.

Adaptive processing [6, 10, 33, 40, 45] 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 [6, 10, 33, 40] (where query execution times are assumed to be longer), adaptive processing has recently also gained traction for classical query processing [24, 38]. SkinnerDB [38] 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 recent works that employ learning for query optimization [19, 22, 23, 44]. 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 [31]. The latest development extends this line of work with data degree constraints and histograms [26]. 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 [15]. 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*. IEEE, 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 arXiv:1503.02368

[3] 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

[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. *Proc. VLDB Endow.* 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 *Performance Evaluation and Benchmarking for the Analytics Era: 9th TPC Technology Conference, TPCTC 2017, Munich, Germany, August 28, 2017, Revised Selected Papers 9*. Springer, 103–119.

[9] Peter A Boncz, Martin L Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85.

[10] Amol Deshpande. 2004. An initial study of overheads of eddies. *SIGMOD Record* 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. *Proceedings of the VLDB Endowment* 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.

[14] 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

[15] 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

[16] 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

[17] 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

[18] Valdis E Krebs. 2002. Mapping Networks of Terrorist Cells. *Connections* 24, 3 (2002), 43–52. arXiv:0309488 [cond-mat] http://www.insna.org/pubs/connections/v24.html

[19] 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

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

[21] Guy Lohman. 2014. Is query optimization a "solved" problem? *SIGMOD Blog* (2014).

[22] 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

[23] 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

[24] 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

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

[26] 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

[27] 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

[28] 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

[29] 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

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

[31] 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

[32] 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.

[33] 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

[34] 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

[35] 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

[36] Michael Stonebraker and Lawrence A Rowe. 1986. The design of Postgres. *ACM Sigmod Record* 15, 2 (1986), 340–355.

[37] 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

[38] 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.

[39] 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

[40] Kostas Tzoumas, Timos Sellis, and Christian S Jensen. 2008. *A reinforcement learning approach for adaptive query processing*. Technical Report.

[41] 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

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

[43] Haiyuan Yu, Alberto Paccanaro, Valery Trifonov, and Mark Gerstein. 2006. Predicting interactions in protein networks by completing defective cliques. *Bioinformatics* 22, 7 (2006), 823–829.

[44] 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

[45] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking ahead makes query plans robust. In *SIGMOD*. 889–900.

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. Our analysis is based on the worst-case optimality properties of LFTJ and makes the same assumptions as the corresponding proof [41] (e.g., restriction to equality joins). We consider the number of threads a constant. Additionally, our analysis makes 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 [41]. 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. □

C STATISTICS ABOUT GRAPH DATASETS

Table 3 describes the graph datasets used in our experiments.

Table 3: 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	689,937,732

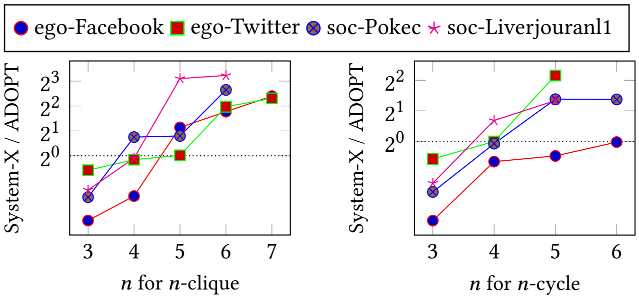


Figure 11: Relative speedup of ADOPT over System-X.

D RELATIVE PERFORMANCE COMPARISON OF ADOPT OVER SYSTEM-X

Figure 11 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.

E DISK AND MEMORY CONSUMPTION

We report the disk space and memory usage of different systems in Table 4 and 5 respectively. We use the command `du -sh` to measure disk space of the respective data folder. For measuring maximal memory consumption for each benchmark, we use the `ps -p pid -o rss=` command. ADOPT is implemented in Java. Hence, we increased the default settings for the Xmx and Xms parameters of the Java virtual machine as follows: `-Xmx15G -Xms15G` on ego-Facebook, `-Xmx20G -Xms20G` on ego-Twitter, `-Xmx25G -Xms25G` on soc-Pokec, `-Xmx50G -Xms50G` on soc-Livejournal1, and `-Xmx80G -Xms80G` on JOB, TPC-H, and JCC-H. For all other systems, to maximize their performance in terms of run time (results reported in Table 1), we increased buffer space to 350 GB, the amount of main memory available on our test machine (e.g., for Postgres, we increased the setting for the shared_buffer_pool parameter). Note that all systems typically exploit only a small part of the total buffer space available to them. As discussed in Section 5 in more detail, some systems were only evaluated on a subset of benchmarks (missing values are marked by "-" in the tables).

Table 5 shows that ADOPT consumes an amount of main memory that is approximately comparable to System X, i.e., ADOPT's main memory consumption is within a factor of 0.6 to 1.7 of the corresponding value for System X for all evaluated benchmarks. This seems reasonable as the execution engine of System X is the most similar to the one of ADOPT (due to the use of LFTJ variants). EmptyHeaded, another worst-case optimal system, consumes more main memory than ADOPT. MonetDB consumes significantly more main memory than ADOPT on the graph benchmarks. Here, using non worst-case optimal joins, MonetDB produces large intermediate results for cyclic queries that are stored in main memory. The use of worst-case optimal join algorithms avoids these overheads. Also, MonetDB consumes more main memory on benchmarks that use skewed data (in particular JCC-H). Here, large intermediate results can be avoided by using the right join order. However, due to data skew, reliably identifying near-optimal query plans without adaptive processing is hard. On the other hand, MonetDB consumes only moderate amounts of main memory on TPC-H. Here, worst-case optimal join algorithms are unnecessary to avoid large intermediate results and query planning is easier (due to uniform data distributions). SkinnerDB incurs high memory overhead on some of the graph benchmarks due to cyclic queries. Postgres incurs high memory overheads for some of the graph benchmarks and for the JCC-H benchmark, due to binary joins and non-adaptive optimization.

ADOPT stores several data structures in main memory that are specific to its adaptive approach: different sort orders for each table to support the LFTJ (each sort order is stored as one integer array with row indexes, of the same length as the table), the set of unprocessed cubes, maintained by the task manager (and represented as variable U in Algorithm 4), and data structures used by the reinforcement learning algorithm, in particular the UCT search tree with associated reward statistics. Table 6 shows the amount of memory consumed by each of these data structures for each benchmark (in addition to the total main memory consumption). For most benchmarks, sort orders consume most main memory, among all auxiliary data structures, reaching up to 11% of total memory consumption for one benchmark. Main memory consumption for storing cubes and UCT statistics is lower by several orders of magnitude, making their contribution to total main memory consumption negligible.

F INDEX CREATION TIME

We report on index creation overheads, referring to the indexes created before the experiments discussed in Section 5.2. For Postgres, we created indexes on all primary and foreign key columns. In addition, for TPC-H and JCC-H, we additionally index the o_orderdate column of the Orders table and build an index with composite search key on the l_shipdate, l_discount, and l_quantity columns of the Lineitem table (the additional indexes improved performance). For SkinnerDB, we run the "index all" command, indexing all suitable columns and thereby optimizing its query evaluation times. For ADOPT, we index the same columns as SkinnerDB via hash indexes (supporting evaluation of unary equality predicates), except for join columns (which SkinnerDB indexes via hash indexes as well). Additionally, ADOPT creates data structures representing different sort orders of base tables (see Appendix J for details). System X and MonetDB create indexes automatically, based on observed queries (note

Table 4: Disk space of different systems on each benchmark.

Systems	JOB	ego-Facebook	ego-Twitter	soc-Pokec	soc-Livejournal1	TPC-H	JCC-H
ADOPT	3.2G	836K	48M	405M	1.1G	8.5G	8.5G
System-X	2.8G	1.6M	46M	387M	759M	-	-
EmptyHeaded	-	6.0M	44M	496M	1.3G	-	-
PostgreSQL	5.4G	11M	101M	1.1G	2.3G	20GB	20GB
MonetDB	2.9G	3.5M	58M	247M	554M	8.5G	8.5G
SkinnerDB	3.1G	834K	47M	403M	882M	8.4G	8.4G

Table 5: Memory space of different systems on each benchmark.

Systems	JOB	ego-Facebook	ego-Twitter	soc-Pokec	soc-Livejournal1	TPC-H	JCC-H
ADOPT	22G	10G	17G	22G	45G	52G	57G
System-X	38G	16G	16G	16G	26G	-	-
EmptyHeaded	-	68G	74G	85G	89G	-	-
PostgreSQL	28G	15G	17G	25G	110G	56G	110G
MonetDB	42G	122G	243G	345G	345G	18G	280G
SkinnerDB	38G	26G	69G	89G	125G	86G	86G

that MonetDB supports the "create index" command but it is only treated as a suggestion, according to the online manual[2]). To give those two systems the opportunity to create suitable indexes, we ran each benchmark once before starting the actual measurements. As those systems interleave query execution and index creations, making it hard to measure index creation time separately, we do not report indexing overheads for those systems. EmptyHeaded creates all relevant indexes in a pre-processing step.

Table 7 reports corresponding results. ADOPT has typically higher index creation overheads than SkinnerDB due to the added overhead for creating sort orders. On the other hand, ADOPT's index creation overheads are below the ones of Postgres. EmptyHeaded incurs higher index creation overheads, compared to ADOPT, as it creates indexes for all permutations of table columns. On JOB, this approach incurs very high index generation overheads, making the approach impractical. Table 8 shows a breakdown of ADOPT's index generation overheads into two components: time for generating indexes representing row orders (to support LFTJ) and time for generating hash indexes on single columns (to support evaluation of unary equality predicates). Clearly, time for generating row orders dominates, even though hashing time is non-negligible for the TPC-H and JCC-H benchmarks.

Table 6: Breakdown of ADOPT's main memory consumption for each benchmark: total memory consumption, memory for storing sort orders, unprocessed cubes, and the UCT search tree.

Benchmark	Total	Sort	Cubes	UCT
JOB	22G	2G	37K	21M
ego-Facebook	10G	689K	445K	4M
ego-Twitter	17G	19M	878K	4M
soc-Pokec	22G	233M	3M	4M
soc-Livejournal1	45G	5G	11M	4M
TPC-H	52G	2G	104K	109K
JCC-H	57G	2G	105K	110K

G SORTING AND SYNCHRONIZATION OVERHEADS

Hypothesis 8. *The times required by ADOPT for sorting and thread synchonization are small relative to the total execution time.*

To implement efficient seek operations in the context of its LFTJ variant, ADOPT requires data structures representing different table sort orders (see Appendix J for details). For base tables, ADOPT creates those sort orders at pre-processing time, corresponding overheads are reported in Appendix F. However, ADOPT creates temporary tables during query evaluation, representing base tables after filtering via unary predicates. For those tables, ADOPT creates all required sort orders at run time. We measured the relative overhead of run time sorting, compared to total query evaluation time. Over all queries and benchmarks, sorting overheads reach at most 2.5% of total query evaluation time. This means that time required for run time sorting is fairly modest (which is explained, in part, by the fact that tables resulting from filter operations tend to be quite small, compared to the source tables).

For each episode and thread, we can sum up the time spent by that thread in processing the LFTJ join on different cubes (considering all cubes processed by the thread during the episode). When measuring the total duration of an episode, the episode time typically exceeds the accumulated join processing time. The difference is due to various bookkeeping and synchronization overheads, e.g., waiting for the lock on the data structure containing unprocessed cubes (locking is necessary to avoid redundant work across threads). Considering all queries and benchmark, the maximal percentage of

[2] https://www.monetdb.org/documentation-Sep2022/user-guide/sql-summary/#create-index

Table 7: Index creation time (in seconds) of different systems for each benchmark it was evaluated on (- if the corresponding system was not evaluated on the benchmark).

Systems	JOB	ego-Facebook	ego-Twitter	soc-Pokec	soc-Livejournal1	TPC-H	JCC-H
ADOPT	63	0.2	1.6	28	64	155	153
EmptyHeaded	> 432000	17	24	38	115	-	-
PostgreSQL	78	0.36	14	144	328	169	172
SkinnerDB	23	0.4	1.1	19	25	82	84

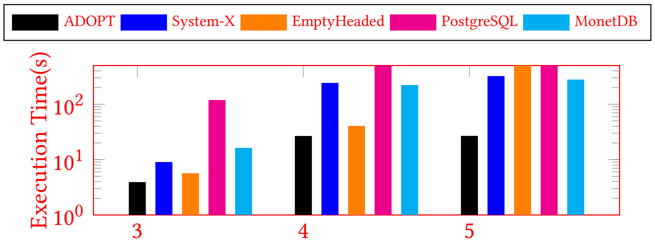


Figure 12: Execution (wall-clock) time for Loomis-Whitney queries on ego-Twitter (x-axis has the number of join attributes in each table).

such overheads, relative to total query execution time, was 7%. This means that the largest part of run time is spent doing useful work.

H SCALABILITY IN THE NUMBER OF JOIN ATTRIBUTES PER RELATION

Many of the benchmarks presented so far have an elevated number of tables and join attributes (e.g., up to 16 tables for JOB). However, the number of join attributes per table is typically small (two join attributes per table for graph benchmarks, reaching up to four attributes for some JOB queries). The number of attributes per table influences the number of sort orders ADOPT has to maintain, possibly influencing its relative performance. Next, we study the impact of the number of attributes per table on the relative performance of ADOPT.

We use Loomis-Whitney queries with varying degree for this purpose. A Loomis-Whitney query with degree n (i.e., the number of join attributes in each table is $n-1$) is defined as,

$$\begin{aligned}&edge(a_1, a_2, \cdots, a_{n-2}, a_{n-1}), edge(a_2, \cdots, a_{n-2}, a_{n-1}, a_n),\\&edge(a_1, a_3, \cdots, a_{n-1}, a_n), edge(a_1, a_2, a_4, \cdots, a_n), \cdots,\\&edge(a_1, a_2, \cdots, a_{n-2}, a_n)\end{aligned}$$

We use the query result of Loomis-Whitney with degree n as the input table of Loomis-Whitney query with degree $n+1$ (truncating the query result to 200M rows as, otherwise, none of the compared systems finish for the highest degree within the timeout of ten minutes). For the different degrees, in ascending order, the table sizes are (approximately) 13M rows, 105M rows, and 200M rows.

The wall-clock execution time for Loomis-Whitney queries on ego-Twitter is depicted in Figure 12. ADOPT performs better than the other baselines and the relative performance gap grows as the number of join attributes increases. Clearly, without adaptive processing, finding good query plans becomes harder as queries become more complex. In addition, we measure overheads for index creation before run time for all systems separating index creation from query evaluation (see Appendix F for details). For Postgres, we create indexes on each column of the input table. EmptyHeaded automatically selects indexes to create. For ADOPT, we create indexes to support all possible attribute orders. Table 9 reports corresponding results. For both baselines implementing worst-case optimal join algorithms (ADOPT and EmptyHeaded), index creation time is higher than for Postgres and grows faster with increasing degree (note that, as discussed previously, the size of the input data increases as well). However, among the two baselines with worst-case optimal join algorithms, ADOPT generates indexes significantly faster. For all systems, the resulting indexes can be reused across all future queries.

Table 8: Breakdown of index generation overheads for ADOPT: total indexing time, time for generating sort orders, and time for generating hash indexes.

Benchmark	Total	Sorting	Hashing
JOB	63	42	21
ego-Facebook	0.2	0.2	0
ego-Twitter	1.6	1.6	0
soc-Pokec	28	28	0
soc-Livejournal1	64	64	0
TPC-H	155	97	58
JCC-H	153	94	59

Table 9: Index generation times for different systems (in seconds), preparing evaluation of Loomis-Whitney queries with varying degree.

System	**Degree 3**	**Degree 4**	**Degree 5**
Postgres	14	161	407
ADOPT	3	81	1003
EmptyHeaded	99	949	3719

$r(a)$	$s(a)$	$t(a)$	join of $r, s,$ and t
0	0	2	8
1	2	4	
3	6	5	
4	7	8	
5	8	10	
6	9		
7			
8			
9			
11			

Figure 13: Unary relations and their join.

I ILLUSTRATION OF LEAPFROG TRIEJOIN

In this section, we illustrate the LFTJ algorithm. In Section I.1, we describe leapfrog join on unary relations, which is the basic building block of LFTJ. We explain in Section I.2 how LFTJ traverses non-unary relations. In Sections I.3 and I.4, we illustrate LFTJ for an acyclic and respectively a cyclic query.

I.1 Leapfrog Join on Unary Relations

Assume we want to join several unary relations over the same attribute. This amounts to computing the intersection of the relations. The leapfrog join algorithm navigates each relation using an iterator that sees the relation as an ordered list. The iterators provide the following operations: $next()$ moves the iterator to the next position in the list, or to **EOF** if no such position exists; given a value v, $seek(v)$ moves the iterator to the position with the least value w such that $w \geq v$, or to **EOF** if no such value exists. At the beginning, each iterator is at the first position of its list. As long as all values at the current positions of the iterators do not match, the leapfrog join algorithm proceeds as follows. Given that the largest value at the current positions of the iterators is v, the algorithm calls $seek(v)$ for one of the iterators with the smallest current value. In case the values at the current iterator positions match, the common value is added to the output. Then, the algorithm calls $next()$ for one of the list and repeats the above strategy to find the next common value. The algorithm stops once one of the iterators reaches **EOF**.

Example I.1. We illustrate leapfrog join for the query $q(a) = r(a), s(a), t(a)$ that computes the intersection of the three unary relations depicted in Figure 13. Figure 14 visualizes how the iterators traverse the three relations. The iterators start at the initial positions of the ordered lists representing the relations. The values at the initial positions do not match and the largest such value is the value 2 in t. Hence, the algorithm calls $seek(2)$ for r, which moves the iterator of r to the position with value 3. Now, the largest value of the current iterator positions is 3, so the algorithm calls $seek(3)$ for s. This operation moves the iterator of s to the position with value 6. Then, it calls $seek(6)$ for t, which moves t's iterator to the position with value 8. Afterwards, it calls $seek(8)$ for r and then for s, upon which the iterators of both relations move to their respective positions holding value 8. Now, all iterators point to the value 8, so we have a match. The algorithm adds 8 to the output and moves the iterator of t to the next position, which holds the value 10. Then, it calls $seek(10)$ for r, which moves r's iterator to the position with value 11. Finally, it calls $seek(11)$ for s, upon which the iterator of s moves to **EOF**, so the algorithm stops.

We conclude that the only output value is 8.

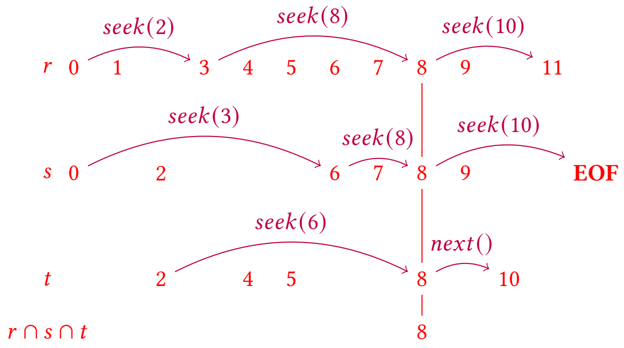


Figure 14: Joining the unary relations in Figure 13 using leapfrog join. The only value in the result is 8.

I.2 Navigation over Non-Unary Relations

LFTJ navigates non-unary relations using iterators that interpret the relations as tries that follow attribute orders. Each level in a trie corresponds to one attribute. The iterators support the following operations: $open()$ moves the iterator to the first child node of the current node; $up()$ returns the iterator to the parent node; $next()$ moves the iterator to the next sibling or **EOF** if no such sibling exists; given a value v, $seek(v)$ moves the iterator to the sibling with the least value w such that $w \geq v$, or to **EOF** if no such value exists.

$r(a, b, c)$		
0	3	4
0	3	5
0	4	0
0	4	1
0	4	3
0	5	2
1	5	2

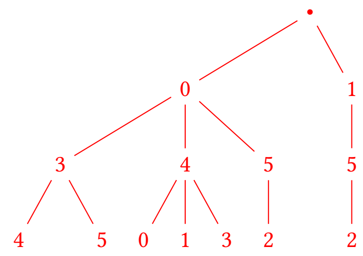


Figure 15: A relation and its interpretation as a trie following the attribute order $a - b - c$.

Example I.2. Figure 15 depicts a relation r and its interpretation as a trie that follows the attribute order $a - b - c$. The children of each node are sorted. The children of the root carry the a-values of r, which are 0 and 1. The children of the a-value 0 carry the values 3, 4, and 5, which are the b-values paired with 0 in r. The children of the c-value 3 carry the values 4 and 5, which are the c-values paired with 0 and 3 in r. The rest of the trie is organized analogously. Figure 16 visualizes how an iterator traverses the values in relation r via the operation sequence $open()$, $open()$, $next()$, $open()$, $seek(2)$, $up()$.

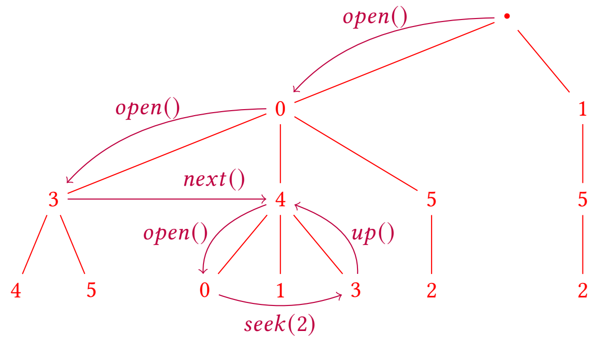


Figure 16: Traversal over the values of relation r in Figure 15.

I.3 Leapfrog Triejoin for an Acyclic Query

Given a set of relations, a global attribute order is an ordering of all attributes appearing in the relations. LFTJ requires that all input relations can be navigated following a global attribute order. Consider a set of relations and a global attribute order $att_1, \ldots, att_n$. To create all tuples in the join result, LFTJ proceeds as follows. It uses leapfrog join to fix the first att_1-value that appears in all relations containing attribute att_1. Given that the values for $att_1, \ldots, att_i$ with $i < n$ are already fixed to values $a_1, \ldots, a_i$, it uses again leapfrog join to fix the first att_{i+1}-value that appears in all relations containing att_{i+1} when restricted to $a_1, \ldots, a_i$. Once all attributes are fixed to values $a_1, \ldots, a_n$, it means that $(a_1, \ldots, a_n)$ constitutes a tuple in the join result, so the algorithm adds it to the output. Then, it triggers leapfrog join to traverse the remaining att_n-values that appear in all relations containing att_n when restricted to $a_1, \ldots, a_{n-1}$. For each such att_n-value a'_n, it adds $(a_1, \ldots, a'_n)$ to the output. Once all att_n-values are exhausted, it backtracks and searches for the next att_{n-1}-value that appears in the join in the context of $(a_1, \ldots, a_{n-2})$, and so on. In the following, we illustrate how LFTJ computes an acyclic join.

$r(a,b,c)$		
0	2	1
0	3	1
1	5	2

$s(a,c)$	
0	1
0	2
2	3

$t(b)$
0
2
3

$u(b,c)$	
0	0
0	2
2	1

join of r, s, t, and u		
0	2	1

Figure 17: Acyclic join of four relations.

Example I.3. Consider the acyclic query $q(a,b,c,) = r(a,b,c)$, $s(a,c)$, $t(b)$, $u(b,c)$ joining the four relations depicted in Figure 17. As shown in the figure, the only tuple in the join result is $(0,2,1)$. Figure 20 shows how LFTJ traverses the four relations following the global attribute order $a - b - c$ to compute the join result. First, the algorithm calls $open()$ for r and s, since these are the only relations containing attribute a. The iterators of these relations move to nodes carrying 0, which means that we have a match for the a-values (second row in Figure 20). Next, the algorithm calls $open()$ for r, t, and u, since these are the relations that have attribute b. The current value of the iterator for r becomes 2 while the current value of the iterators for t and u become 0 (third row in Figure 20). This means that the b-values do not match yet. Since the largest current b-value is 2, the algorithm calls $seek(2)$ for t, which moves the iterator of t to the sibling node with value 2 (fourth row in Figure 20). In this situation, the b-values still do not match and the largest current b-value is 2. The algorithm calls $seek(2)$ for u, moving the iterator of u to the sibling node with value 2 (fifth row in Figure 20). Now, the iterators of r, t, and u point to 2, so the algorithm fixes the b-value to 2. Calling $open()$ for r, s, and u moves their iterators to child nodes with value 1. Hence, the algorithm fixes the c-value to 1 (sixth row in Figure 20). It follows that $(0,2,1)$ constitutes the first result tuple, which is added to the output. After backtracking, the algorithm realizes that there is no further tuple in the join result and stops.

I.4 Leapfrog Triejoin for a Cyclic Query

The next example illustrates that traditional join algorithms are suboptimal in the sense that they can produce intermediate results that are larger than the final result. In contrast, LFTJ does not produce intermediate results and constructs one output tuple at a time. The example considers the triangle query and showcases a database that was used in prior work to demonstrate the suboptimality of traditional join algorithms on skewed data [28].

Example I.4. Consider the triangle query $q(a,b,c) = r(a,b), s(a,c), t(b,c)$, which joins the three relations r, s, and t depicted in Figure 18. Each relation has two values of degree $m+1$ and $2m$ values of degree 1. For instance, in relation r, each of the values a_0 and b_0 is paired with $m+1$ distinct values and each of the remaining values is paired with exactly one value. Each input relation has $2m+1$ tuples while the result (Figure 18 right) has $3m+1$ tuples. Hence, the size of the result is linear in m.

$r(a,b)$	
a_0	b_0
a_0	$\ldots$
a_0	b_m
a_1	b_0
$\ldots$	b_0
a_m	b_0

$s(a,c)$	
a_0	c_0
a_0	$\ldots$
a_0	c_m
a_1	c_0
$\ldots$	c_0
a_m	c_0

$t(b,c)$	
b_0	c_0
b_0	$\ldots$
b_0	c_m
b_1	c_0
$\ldots$	c_0
b_m	c_0

join of r, s, and t		
a_0	b_0	c_0
a_0	b_0	$\ldots$
a_0	b_0	c_m
a_0	b_1	c_0
a_0	$\ldots$	c_0
a_0	b_m	c_0
a_1	b_0	c_0
$\ldots$	b_0	c_0
a_m	b_0	c_0

Figure 18: Relations of the triangle join.

Traditional join algorithms first join two of the three input relations and then join in the remaining one. Figure 19 shows the result of any pairwise join. In each case, the result contains $(m+1)^2 + m$ tuples, which is quadratic in m. This means that any join plan that first joins two of the three input relations needs at least quadratic time, while the size of the final result is linear in m.

LFTJ does not produce any intermediate result and its computation time is proportional to the size of the final result. Figure 21 visualizes how LFTJ traverses the three relations following the

join of r and s			join of r and t			join of s and t		
a_0	b_0	c_0	a_0	b_0	c_0	a_0	b_0	c_0
a_0	b_0	...	a_0	b_0	...	a_0	...	...
a_0	b_0	c_m	a_0	b_0	c_m	a_0	b_m	c_0
...	...	...	...	...	...	...	...	...
a_0	b_m	c_0	a_m	b_0	c_0	a_m	b_0	c_0
a_0	b_m	...	a_m	b_0	...	a_m	...	c_0
a_0	b_m	c_m	a_m	b_0	c_m	a_m	b_m	c_0
a_1	b_0	c_0	a_0	b_1	c_0	a_0	b_0	c_1
...	b_0	c_0	a_0	...	c_0	a_0	b_0	...
a_m	b_0	c_0	a_0	b_m	c_0	a_0	b_0	c_m

Figure 19: Pairwise joins of the relations r, s, and t from Figure 18.

global attribute order $a - b - c$. The last column in the Figure shows how the join result is produced. Just as in Example I.3, LFTJ uses leapfrog join to compute the intersection of a-values, the intersection of b-values in the context of a given a-value, and the intersection of c-values in the context of a given (a, b)-pair. Hence, in the sequel we focus more on the order in which complete result tuples are produced.

The first result tuple is constructed by fixing the attributes a, b, and c to the values a_0, b_0, and c_0, respectively (second row in Figure 21). The c-values in the context of a_0 in s and in the context of b_0 in t are teh same: $c_1, \ldots, c_m$. So, we obtain the result tuples (a_0, b_0, c_1), ..., (a_0, b_0, c_m) (third row in Figure 21). The b-values in the context of a_0 in r are the same as the b-values in t. All b-values in t have c_0 as a child, which is also child of a_0 in s. So, the algorithm produces the result tuples (a_0, b_1, c_0), ..., (a_0, b_m, c_0) (fourth row in Figure 21). At this point, all b- and c-values in the context of a_0 are exhausted. The algorithm moves to the next a-value a_1 in the intersection of the a-values in r and s. The value b_0 appears in t and in the context of a_1 in r. Moreover, the value c_0 appears in the context of a_1 in s and of b_0 in t. Since this is a match, the algorithm adds (a_1, b_0, c_0) to the output (fifth row in Figure 21). Next, the algorithm iterates over the a-values $a_2, \ldots, a_m$. Each of these values have b_0 as child in r and c_0 as child in s. Hence, the algorithm produces the result tuples (a_2, b_0, c_0), ..., (a_m, b_0, c_0) (sixth row in Figure 21).

J LFTJ IMPLEMENTATION IN ADOPT

ADOPT uses a variant of the LFTJ, described in Algorithm 3 at a relatively high level of abstraction. The loop from Lines 10 to 17 iterates over values for the current attribute that satisfy all applicable join predicates. Here, ADOPT uses the LFTJ approach, discussed in the preceding subsections, to efficiently identify the next attribute value that appears in all input relations. As discussed previously, identifying the next value (v in Algorithm 3) may involve repeated "seek" operations on all input relations that contain the current attribute. The LFTJ, as presented so far, focuses on equality join predicates. In addition, ADOPT processes other join predicates by simply skipping to the next attribute value if predicates evaluate to false with the current value. Also, ADOPT only considers values for each attribute that fall within the current target cube (by starting from the lower bound and terminating iterations once seek operations return values above the upper bound). Whenever ADOPT proceeds to a new attribute, it performs the equivalent of the "open" operation on all input tables containing the new attribute. Finally, whereas the original LFTJ focuses on set data, ADOPT supports multi-sets by iterating over all tuple combinations having the currently selected combination of values in all join columns (variable M in Algorithm 3), when updating the query result set (Line 5 in Algorithm 3).

LFTJ variants generally rely on data structures that enable efficient seek operations. Each table is *logically* organized as a trie, where each level corresponds to one attribute and the order of the levels follows the global attribute order. This logical organization can be supported *physically* by sorted tables as done by ADOPT, B$^+$-trees as in the original LFTJ implementation in LogicBlox [4], or nested hashing [11]. More precisely, assume we want to process the global attribute order a_1 to a_m. For a specific table, denote by a_{i_1} to a_{i_n} the subset of attributes that appear in that table, in the same order as they appear in the global attribute order. ADOPT sorts the rows in that table according to values for attributes a_{i_1} to a_{i_n}, prioritizing attribute values in this order during comparisons (e.g., rows are ordered according to their value for a_{i_1} as first priority, considering the value for a_{i_2} only when comparing rows with the same value for a_{i_1}). ADOPT avoids materializing the sorted table but merely stores the ordered row indexes in an integer array of the same length as the table. Since ADOPT is a column store and stores columns as arrays, data access via the row index is efficient. Having sorted rows enables ADOPT to implement seek operations efficiently.

The original LFTJ algorithm assumes that tables have been sorted during pre-processing before run time [41]. To try out different global attribute orders, ADOPT may require multiple alternative sort orders for the input relations. Whenever ADOPT selects a global attribute order, it determines which local sort orders are required for each table. If the corresponding order (i.e., the array containing sorted row indexes) is not cached, ADOPT creates the corresponding order at run time. In its caching policy, ADOPT distinguishes tables with and without unary predicates. For tables obtained after applying unary predicates, ADOPT stores associated sort orders only while processing the current query. Note that tables tend to be small after applying unary predicates, making sorting them relatively cheap. Tables without unary predicates tend to be large and sorting is more expensive. Here, ADOPT caches corresponding sort orders beyond the duration of the current query, reusing them for future queries if possible.

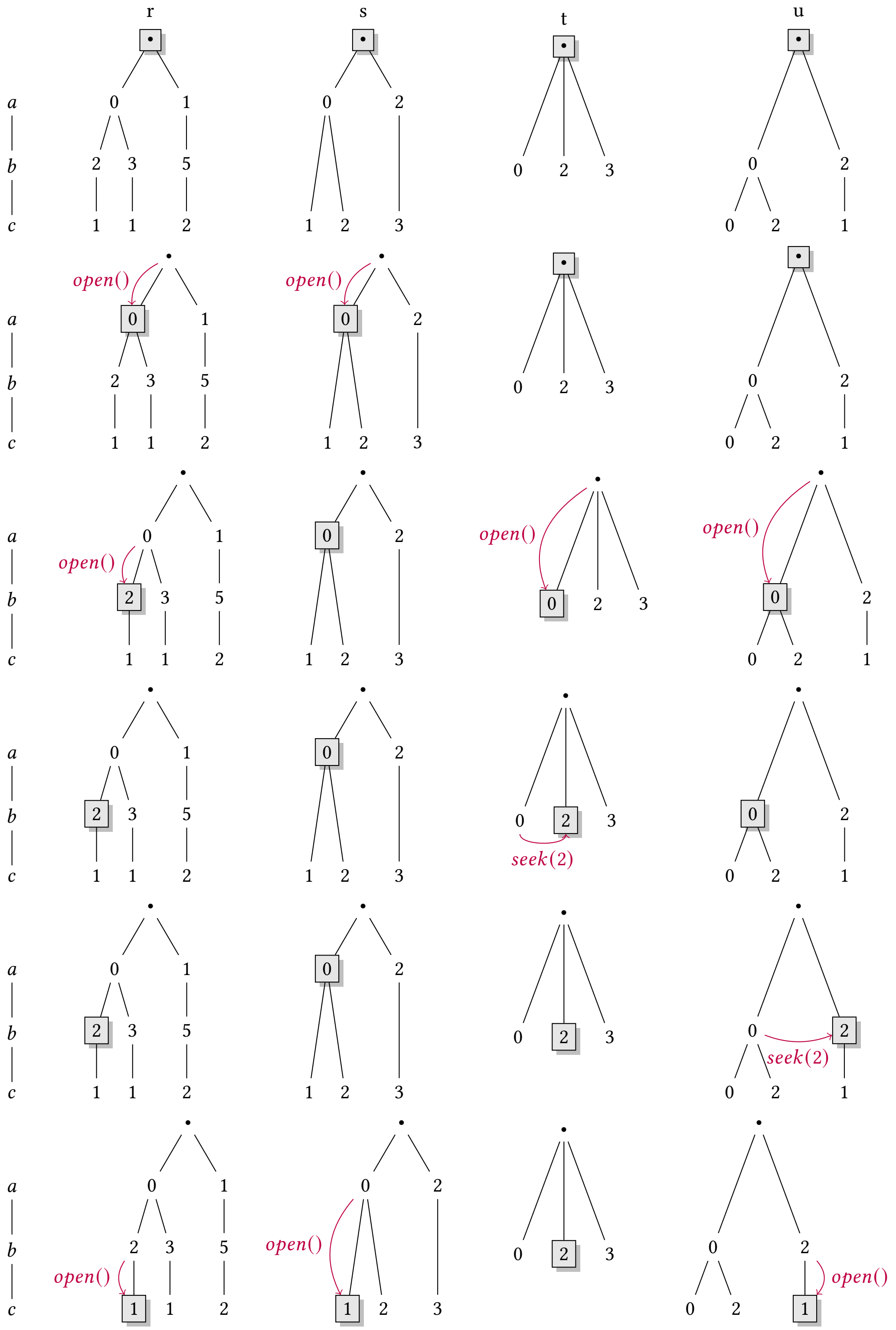


Figure 20: LFTJ execution for the input relations in Figure 17 following the attribute order $a - b - c$ (depicted to the left).

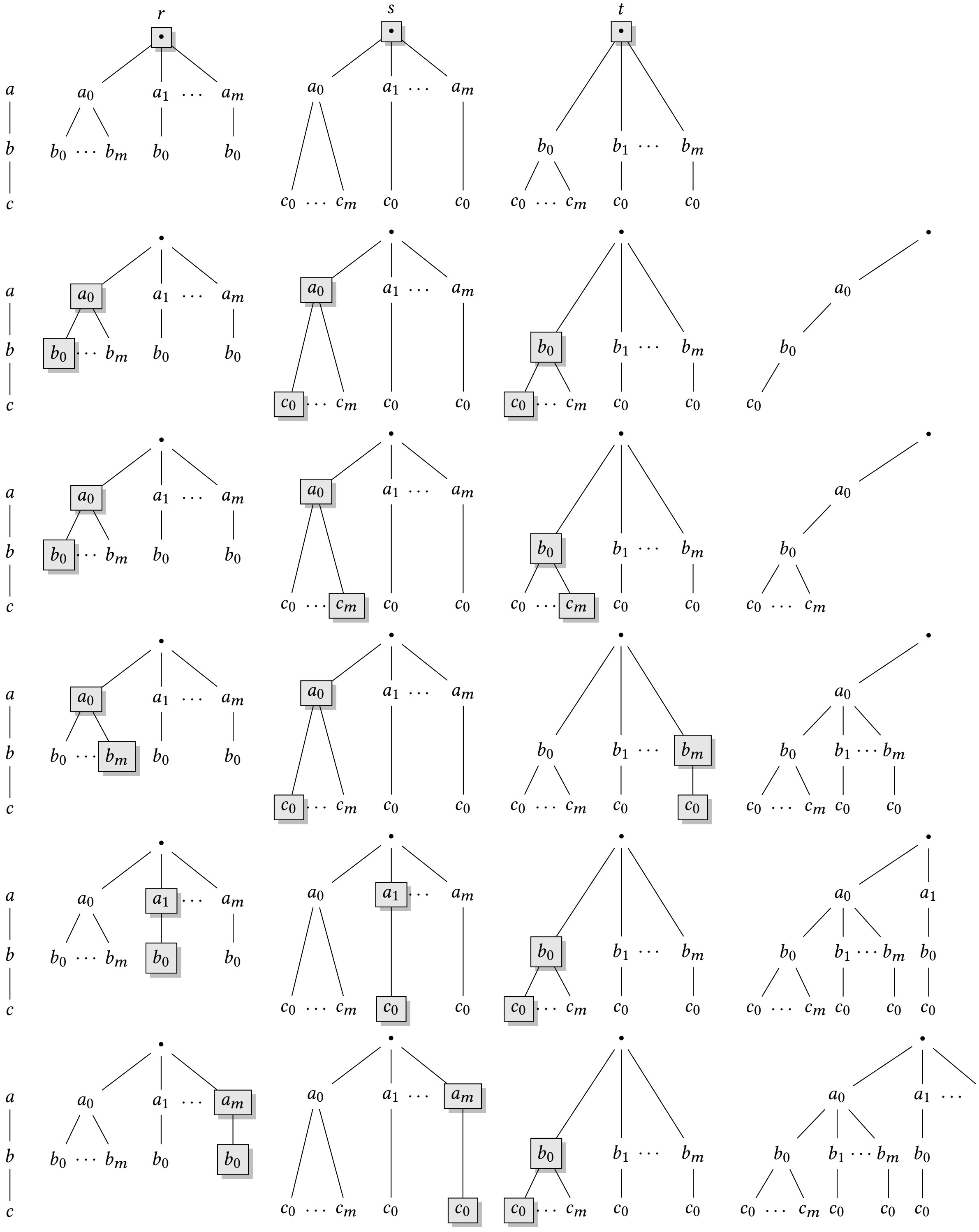


Figure 21: LFTJ execution for relations r, s, and t in Figure 18 following the attribute order $a - b - c$ (depicted to the left). The last column shows how the join output is produced one tuple at a time.