Demonstrating ADOPT: Adaptively Optimizing Attribute Orders for Worst-Case Optimal Joins via Reinforcement Learning

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

Mitchell Gray
Cornell University
Ithaca, NY, USA
meg346@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

Performance of worst-case optimal join algorithms depends on the order in which the join attributes are processed. It is challenging to identify suitable orders prior to query execution due to the huge search space of possible orders and unreliable execution cost estimates in case of data skew or data correlation.

We demonstrate ADOPT, a novel query engine that integrates adaptive query processing with a worst-case optimal join algorithm. ADOPT divides query execution into episodes, during which different attribute orders are invoked. With runtime feedback on performance of different attribute orders, ADOPT rapidly approaches near-optimal orders. Moreover, ADOPT uses a unique data structure which keeps track of the processed input data to prevent redundant work across different episodes. 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, ADOPT outperforms baselines, including commercial and open-source systems utilizing worst-case optimal join algorithms, particularly for complex queries that are difficult to optimize.

PVLDB Reference Format:
Junxiong Wang, Mitchell Gray, Immanuel Trummer, Ahmet Kara, and Dan Olteanu. Demonstrating ADOPT: Adaptively Optimizing Attribute Orders for Worst-Case Optimal Joins via Reinforcement Learning. PVLDB, 16(12): 4094 - 4097, 2023.
doi:10.14778/3611540.3611629

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

1 INTRODUCTION

Recently, worst-case optimal join algorithms have brought about a revolution in the field of join processing, in particular LeapFrog TrieJoin (LFTJ) [9]. Those algorithms provide formal guarantees of asymptotically worst-case optimal complexity, setting them apart

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611629

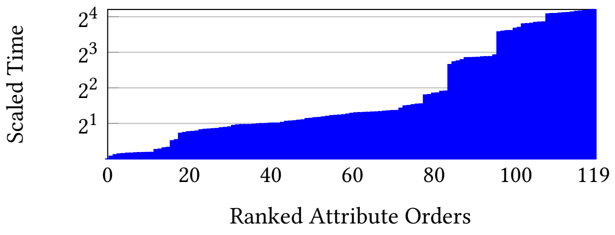

Figure 1: Execution times for different attribute orders of the five-clique query on the ego-Twitter graph.

from traditional join algorithms that are known to be sub-optimal. In practice, worst-case optimal join algorithms can lead to significant improvements in runtime performance of cyclic queries when compared to traditional approaches. As a result, they have been integrated in the commercial systems LogicBlox [2] and RelationalAI and also into various query engines such as those used for factorized databases, graph processing, general query processing, and in-database machine learning. However, the performance of any worst-case optimal join algorithm crucially depends on the order in which join attributes (i.e., groups of join columns that are linked by equality constraints) are processed. Even though all orders are equally good to achieve optimality in the worst case, their performance may differ significantly in practice. Figure 1 shows the performance between different attribute orders for the five-clique query on the ego-Twitter graph. Clearly, the performance gap between the best and worst attribute order is more than 16x.

We demonstrate ADOPT (ADaptive wOrst-case oPTimal joins), the first adaptive processing strategy for worst-case optimal join algorithms. ADOPT uses reinforcement learning (Monte Carlo Tree) to adaptively select best attribute orders. The goal of adaptive processing is to enable attribute order switches during query processing. Specifically, the processing time is divided into episodes, and for each episode, ADOPT selects an attribute order for executing 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.

In our demonstration, participants will be able to run their queries and gain insights into how ADOPT identifies optimal attribute orders by visualizing various aspects of adaptive processing via interactive videos.

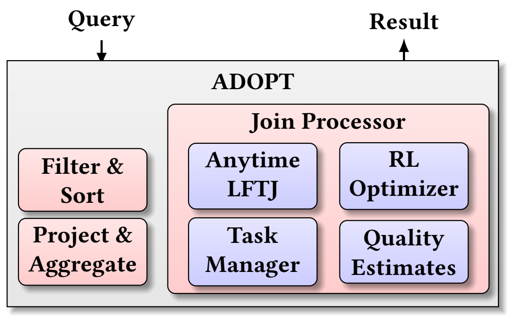

Figure 2: Overview of ADOPT system components.

2 OVERVIEW

Figure 2 shows the primary components of ADOPT, described in more detail in a recent paper [10], available as pre-print[1]. ADOPT performs in-memory data processing using a columnar data layout. It uses a reinforcement learning-based approach to select attribute orders and the LeapFrog TrieJoin algorithm to process joins.

Filter & Sort. ADOPT begins with a preprocessing step that uses unary predicates from the query to filter the tables. Following this, it constructs sort indices to facilitate the use of LeapFrog TrieJoin (LFTJ) during the join phase.

Anytime LFTJ. ADOPT employs a worst-case optimal join algorithm based on LeapFrog TrieJoin (LFTJ). This algorithm (LFTJ) considers join attributes in a fixed order to find value combinations that satisfy all join predicates. ADOPT implements an anytime variant of this algorithm, enabling it to frequently halt and resume execution. This feature facilitates an adaptive processing approach, which permits ADOPT to identify near-optimal attribute orders based on feedback obtained during run time.

RL optimizer. ADOPT utilizes a reinforcement learning-based optimizer to select attribute orders. The optimizer balances the exploration of new attribute orders with the exploitation of those that have proven good in the past. Each selected attribute order is executed for a limited number of steps (an "episode"), allowing ADOPT to test numerous attribute orders per second. To compare attribute orders, ADOPT defines a quality estimation metric that assesses the performance of an attribute order during a single invocation. Although performance may vary for the same attribute order across different invocations due to skewed data, ADOPT obtains increasingly accurate quality estimates over time by averaging the results of multiple invocations for a given attribute order.

ADOPT utilizes the UCT algorithm [4] to select attribute orders. The UCT algorithm represents the state space as a search tree, where each node denotes a state, and edges represent transitions. The tree nodes are equipped with statistics, which establish confidence bounds on the average reward linked with the sub-tree rooted at that node. Confidence bounds are updated as new reward samples become available. During each episode, the UCT algorithm selects a path from the root of the search tree to one of the leaf nodes. At each step, the algorithm chooses the child node with the highest upper confidence bound. Subsequently, the confidence bounds are updated for each node on the selected path after calculating the associated reward. This method ensures convergence to optimal policies, as has been shown in [4].

Task manager. When switching between attribute orders, it can be difficult to avoid redundant work. ADOPT addresses this issue by utilizing a task manager that keeps track of the portions of the join input that still need to be processed. Specifically, the task manager keeps track of (hyper)cubes in the Cartesian product space, created from the value ranges of all join attributes. Each cube represents a part of the input space that requires processing by some attribute order (i.e., the corresponding result tuples have not yet been added to final result set). The anytime LFTJ algorithm's execution is confined to cubes that have not been processed previously or concurrently by other threads. Data processing threads query the task manager for cubes (known as "target cubes") that do not overlap with any previously or concurrently processed cubes by other threads. Threads process the target cube until either completion or reaching the per-episode limit of computational steps. The task manager is informed of the processed portions of the target cube. The task manager then removes the processed cubes from the set of remaining cubes.

Quality Estimates. The reinforcement learning process in ADOPT relies on reward values to guide the selection of the best attribute order for a specific data part. These values represent the quality of an attribute order when processing a particular portion of the data. While quality estimates may differ across different runs of the same order, ADOPT gradually converges to the order with the highest average quality over time.

The volume of a cube is defined as the product of all value ranges of join attributes. We use $Volume(p)$ to denote the volume of a cube p. Using this notation, we can write $Volume(q)$ to indicate the volume of the cube that spans all join attributes of a query q.

To fully process a query, ADOPT needs to cover the entire space of attribute value combinations, which corresponds to the cube with the highest volume. Therefore, the more volume covered per unit time, the faster the query processing. This implies that the covered volume is a useful measure of progress. The reward function used by ADOPT takes this into account and uses the aggregate volume covered, scaled to the total volume to process, for a set of processed cubes P for query q. This scaling ensures that the reward values are between zero and one, as required by the UCT algorithm:

$$Reward(P, q) = (\sum_{p \in P} Volume(p))/Volume(q) \tag{1}$$

Project & Aggregate. ADOPT executes a post-processing stage for certain types of queries, involving group-by clauses and aggregates. For count queries, ADOPT integrates join processing with aggregation and thus does not require a separate post-processing stage.

3 EXTRACT OF EXPERIMENTAL RESULTS

We present a small extract of the full experimental results for ADOPT. More results, as well as more details on the experimental setup, are reported in the full paper [10].

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

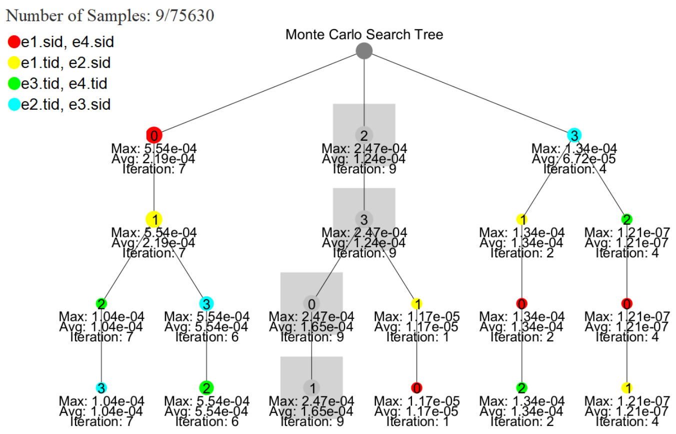
Number of Samples: 9/75630
Monte Carlo Search Tree
e1.sid, e4.sid
e1.tid, e2.sid
e3.tid, e4.tid
e2.tid, e3.sid
0 Max: 5.54e-04 Avg: 2.19e-04 Iteration: 7
2 Max: 2.47e-04 Avg: 1.24e-04 Iteration: 9
3 Max: 1.34e-04 Avg: 6.72e-05 Iteration: 4
1 Max: 5.54e-04 Avg: 2.19e-04 Iteration: 7
3 Max: 2.47e-04 Avg: 1.24e-04 Iteration: 9
1 Max: 1.34e-04 Avg: 1.34e-04 Iteration: 2
2 Max: 1.21e-07 Avg: 1.21e-07 Iteration: 4
2 Max: 1.04e-04 Avg: 1.04e-04 Iteration: 7
3 Max: 5.54e-04 Avg: 5.54e-04 Iteration: 6
0 Max: 2.47e-04 Avg: 1.65e-04 Iteration: 9
1 Max: 1.17e-05 Avg: 1.17e-05 Iteration: 1
0 Max: 1.34e-04 Avg: 1.34e-04 Iteration: 2
0 Max: 1.21e-07 Avg: 1.21e-07 Iteration: 4
3 Max: 1.04e-04 Avg: 1.04e-04 Iteration: 7
2 Max: 5.54e-04 Avg: 5.54e-04 Iteration: 6
1 Max: 2.47e-04 Avg: 1.65e-04 Iteration: 9
0 Max: 1.17e-05 Avg: 1.17e-05 Iteration: 1
2 Max: 1.34e-04 Avg: 1.34e-04 Iteration: 2
1 Max: 1.21e-07 Avg: 1.21e-07 Iteration: 4

Figure 3: Visualizing attribute order search tree in ADOPT.

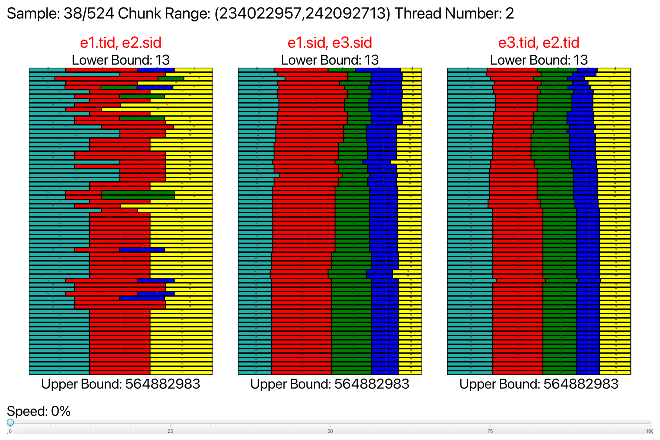

Figure 4: Visualizing processed volumes in the space of potential join results, projected onto different attributes.

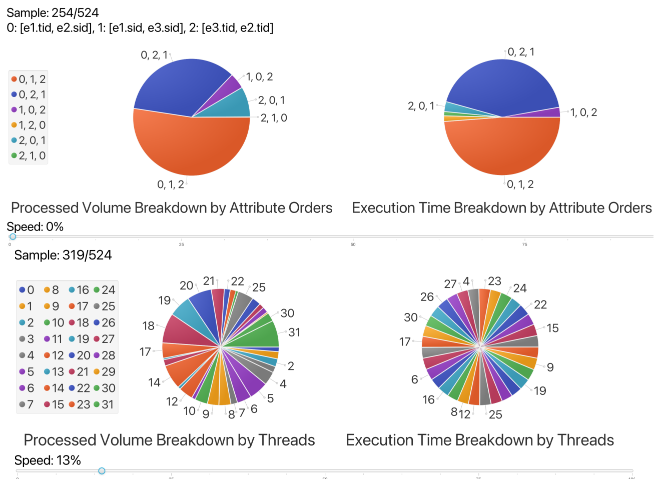

Figure 5: Breakdowns of execution time and processed volume by attribute order and by thread.

Benchmarks. We adopt the approach from previous studies that compared worst-case optimal join algorithms with traditional join plans. Prior work evaluates the performance of these algorithms on clique and cycle queries using the binary edge relations of four graph datasets from the SNAP network collection [6]. Whereas ADOPT is primarily targeted at benchmarks with cyclic queries, we evaluate it on the join order benchmark (JOB) [5] as well.

Systems. ADOPT is implemented in JAVA (jdk 1.8) and uses 10,000 steps per episode and a UCT exploration ratio of 1E-6. The competitors include the open-source engines MonetDB [3] (Database Server Toolkit v11.39.7, Oct2020-SP1) and PostgreSQL 10.21 [7], which employ traditional join plans, as well as a commercial engine System-X (implemented in C++) that uses the worst-case optimal LFTJ algorithm [9], the open-source engine EmptyHeaded that uses a worst-case optimal join algorithm [1], and SkinnerDB [8] (implemented in Java jdk 1.8), which 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 in Table 1. We used a server with 2 Intel Xeon Gold 5218 CPUs with a clock speed of 2.3 GHz (32 physical cores), 384GB of RAM, and a 512GB hard disk. ADOPT, EmptyHeaded, MonetDB, SkinnerDB, and System-X were set to run in memory. By default, all engines use 64 threads; for ADOPT, we also investigate its runtime as a function of the number of threads.

Results. Table 1 shows the total execution time in seconds for each benchmark. In the case of the JOB benchmark, the ">" symbol denotes that the reported time is only for a subset of the queries, while for the four graph datasets, the ">" symbol indicates that the time taken for some cyclic queries exceeded the six-hour timeout. The speedup factors in parentheses after the system runtimes indicate how much faster ADOPT is compared to those systems. ADOPT performs better than its competitors on all graph benchmarks, achieving speedups of 2-30x. In the case of the JOB benchmark, ADOPT is almost as good as MonetDB, with a speedup of 0.91x, and outperforms the other systems by a factor of 1.4-6.3x.

4 DEMONSTRATION SETUP

The demonstration setup consists of one table with several laptops, enabling visitors to execute queries via ADOPT and to visualize various aspects of the query execution process as videos. Visitors will be able to use ADOPT locally, installed on the laptops. To run queries on larger data sets, visitors will have access to a remote server with a large number of cores and main memory. We will load all datasets mentioned in the last section into ADOPT prior to the demonstration. This enables visitors to run queries on a variety of graphs and relational databases. We also provide example queries for each of the benchmarks that users can run with a single click.

Running queries via the ADOPT demo results in log files that contain traces, describing the query evaluation process. For instance, those log files capture the attribute orders, selected in different episodes, reward values obtained, and the internal counters of the reinforcement learning algorithm used for order selection. Also, the logs store information on execution time breakdowns, data skew, as well as per-thread progress metrics analyzing the efficiency of

Table 1: Total time in seconds to compute all queries for each benchmark for different baselines.

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

multi-threading. Rather than reading logs, visitors will be able to transform those traces into multiple, interactive visualizations. These visualizations will enable visitors to explore various aspects of ADOPT's query evaluation process. To enable users to explore visualizations without spending the time to run queries first, we will also provide a set of pre-recorded log files, capturing a diverse mix of queries and benchmarks.

As ADOPT uses an adaptive evaluation strategy, visualizations are not static but dynamic (i.e., videos), showing the evolution of attribute order choices and other variables over the course of a query evaluation. Visitors will have various possibilities to interact with those visualizations, adapting the speed of the evolution (e.g., to see key moments in slow motion), stopping and resuming visualizations, as well as gathering additional information by hovering over or clicking on certain visualization features. In doing so, visitors gain a deeper understanding into the internals of ADOPT and how adaptive processing converges to optimal attribute order choices.

Figures 3, 4, and 5 show example visualizations that visitors will be able to access during the demo. Those visualizations are described in more detail next.

Figure 3 shows a visualization that focuses on ADOPT's learning-based optimizer. ADOPT uses reinforcement learning to select attribute orders to try in each time slice. Figure 3 visualizes the internal state of that learning algorithm (which changes over time). The space of possible attribute orders is visualized as a search tree. Tree edges represent the selection of one single attribute. Paths from the root node to a leaf node represent an entire attribute order. Initially, the search tree is built only partially (to avoid disproportional startup overheads). Visitors will be able to watch the tree gradually expand over time, as query evaluation proceeds. Currently selected attribute orders are marked up with gray rectangles. The node size represents the average reward that the learning algorithm currently associated with specific nodes. Nodes that are visited less frequently, gradually fade over time.

Figure 4 shows a visualization that captures the way in which ADOPT processes the space of potential join results. ADOPT gradually explores the space of potential join results (i.e., attribute value combinations), i.e., it searches this space for corresponding result tuples. ADOPT keeps track of parts of that space already explored, thereby avoiding redundant work across different episodes. Multiple threads explore non-overlapping parts of the search space in parallel (and, possibly, using different attribute orders). Figure 4 offers visitors insights into how the aforementioned search space is divided across different threads. The space of possible join result is, in general, high-dimensional (it has one dimension for each query attribute) and therefore difficult to visualize directly. Instead, the visualization in Figure 4 displays multiple projections, showing the percentage of value combinations explored by different threads, given a fixed value range in one of the three attributes.

Figure 5 shows dynamic breakdowns, considering different metrics and breakdown criteria. The upper part shows breakdowns of execution time and processed volume in the space of join results by attribute order. As ADOPT selects different attribute orders in different episodes, execution time (and volume explored) is split across multiple attribute orders. Over time, ADOPT converges to high-quality attribute orders. Therefore, visitor will observe how the breakdown, starting from a rather uniform distribution, is more and more dominated by a single attribute order as query evaluation proceeds. The lower part of Figure 5 shows breakdowns by thread. Here, visitors will be able to judge the efficiency of ADOPT's parallelization mechanism, showing that the execution time breakdown per thread is almost uniform over the course of query evaluation. On the other hand, the volume explored per thread is typically not uniform, showing that exploration overheads per volume unit are typically non-uniform (due to heterogeneous data).

REFERENCES

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: a relational engine for graph processing. In *SIGMOD*. 431–446. https://doi.org/10.1145/2882903.2915213 arXiv:1503.02368

[2] Molham Aref, 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

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

[4] 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

[5] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. In *VLDB Journal*, Vol. 27. Springer Berlin Heidelberg, 643–668. https://doi.org/10.1007/s00778-017-0480-7

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

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

[8] 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

[9] 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*. OpenProceedings.org, 96–106. https://doi.org/10.5441/002/icdt.2014.13

[10] Junxiong Wang, Immanuel Trummer, Ahmet Kara, and Dan Olteanu. 2023. ADOPT: Adaptively Optimizing Attribute Orders for Worst-Case Optimal Join Algorithms via Reinforcement Learning. *arXiv preprint arXiv:2307.16540* (2023).