SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning

Immanuel Trummer, Samuel Moseley, Deepak Maram,
Saehan Jo, Joseph Antonakakis
Cornell University
{itrummer,sjm352,sm2686,sj683,jma353}@cornell.edu

ABSTRACT

Robust query optimization becomes illusory in the presence of correlated predicates or user-defined functions. Occasionally, the query optimizer will choose join orders whose execution time is by many orders of magnitude higher than necessary. We present SkinnerDB, a novel database management system that is designed from the ground up for reliable optimization and robust performance.

SkinnerDB implements several adaptive query processing strategies based on reinforcement learning. We divide the execution of a query into small time periods in which different join orders are executed. Thereby, we converge to optimal join orders with regret bounds, meaning that the expected difference between actual execution time and time for an optimal join order is bounded. To the best of our knowledge, our execution strategies are the first to provide comparable formal guarantees. SkinnerDB can be used as a layer on top of any existing database management system. We use optimizer hints to force existing systems to try out different join orders, carefully restricting execution time per join order and data batch via timeouts. We choose timeouts according to an iterative scheme that balances execution time over different timeouts to guarantee bounded regret. Alternatively, SkinnerDB can be used as a standalone, featuring an execution engine that is tailored to the requirements of join order learning. In particular, we use a specialized multi-way join algorithm and a concise tuple representation to facilitate fast switches between join orders. In our demonstration, we let participants experiment with different query types and databases. We visualize the learning process and compare against baselines.

PVLDB Reference Format:
Immanuel Trummer, Samuel Moseley, Deepak Maram, Saehan Jo, Joseph Antonakakis. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. *PVLDB*, 11 (12): 2074 - 2077, 2018.
DOI: https://doi.org/10.14778/3229863.3236263

Proceedings of the VLDB Endowment, Vol. 11, No. 12

DOI: https://doi.org/10.14778/3229863.3236263

1. INTRODUCTION

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

The goal of query optimization is to translate a declarative query, describing data to generate, into an optimal query plan. Execution cost of good and bad plans for a given query often differs by many orders of magnitude [21], making reliable query optimization a prerequisite for high-performance data processing. Perhaps surprisingly, the primary challenge in query optimization is often not the enumeration of an overly large search space. Instead, the primary challenge is to reliably identify the plan with minimal execution cost.

Execution cost estimates are based on cardinality estimates for intermediate results, generated during the execution of a query plan. Accurately estimating the size of intermediate results is however inherently difficult before a plan is executed [21]. For instance, data skew leads to correlations between different query predicates, making it hard to estimate the selectivity of predicate groups. Complex user-defined predicates (UDFs) have to be treated as black boxes from the optimizer perspective, at least when being encountered for the first time, and make optimization even harder. Traditional optimizers base their estimates on coarse-grained statistics about value distributions and many simplifying assumptions (e.g., independence between different query predicates) that are all too often violated. In that case, disastrous query plans may result whose execution time is higher than the optimum by many orders of magnitude.

We can gain information on the cardinality of intermediate results by executing plans (or plan fragments) on data samples. Such information is valuable as it might lead to better decisions with regards to query plan properties such as join order. On the other side, collecting information is computationally expensive. We cannot obtain reliable size estimates for all intermediate result candidates (as their number grows quickly as a function of query size). We need an appropriate strategy to balance the value of collected information against the cost of collecting it. The area of reinforcement learning [24] has produced a rich body of work providing strategies with formal guarantees for such scenarios. In this demonstration, we show how such strategies can be leveraged to learn optimal join orders.

The remainder of this paper is organized as follows. In Section 2, we give a high-level overview of the approach behind our system. Then, in Section 3, we describe differences to prior work in the area of robust query processing. Finally, we describe the specifics of our demonstration in Section 4.

2. APPROACH OVERVIEW

We discuss the main ideas behind our approach to join ordering in the following.

2.1 Join ordering: learning-by-doing

Our focus differs from prior work on learning optimizers [38] or learning DBMS [33]. Those approaches generally learn from the execution of past queries in order to improve optimization of the following queries. This is however only possible if future queries are similar enough to past queries, if this similarity can be recognized by the optimizer, and if the underlying data remains stable. Even slight changes to a query can change execution time of plan candidates significantly. Instead of learning from past executions, our goal is to learn *during the execution of a query*. This ensures that all gained knowledge is directly applicable to the query at hand.

To enable learning during the execution of a single query, we divide query evaluation into many small time periods. We may try out a different join order in each period. Our selection of the next join order is generally based on our experiences gained from trying join orders in the preceding time periods. Here, we face a dilemma. In order to finish execution as quickly as possible, we are motivated to *exploit* previously gained knowledge and to use a join order that has worked well in the preceding periods (i.e., small execution time for a small data batch or, equivalently, fast progress for a fixed time budget). On the other side, we are motivated to try out join orders about which little is known so far as we have not tried them (often) yet. Such *exploration* could give us valuable information allowing us to speed up query evaluation via better choices in the future. All our processing strategies use the so called UCT algorithm [26] to strike an optimal balance between exploration (i.e., learning about new join orders) and exploitation (i.e., using join orders that have shown to work well for the current query).

2.2 Join order learning on top of existing DBMS execution engines

We propose three processing strategies, implemented in a novel DBMS called SkinnerDB. Our first processing strategy sits on top of an existing DBMS that is used as execution engine. In our demonstration, we will for instance use Postgres as underlying execution engine. We initially partition input data into small batches and use optimizer hints combined with timeouts to force the DBMS to explore different join orders. A crucial question in this context is the choice of an appropriate timeout. Choosing the timeout per batch too low prevents us from making progress. Choosing it too high allows bad join orders to increase execution time far beyond the time required by an optimal join order. Of course, we cannot know initially what timeout is optimal. We use an iterative scheme that selects timeouts from a geometric progression, carefully balancing execution time between different timeouts and slowly increasing the set of considered timeouts as execution time increases. The UCT algorithm, based on an appropriate search space and reward function, governs the choice of join orders for specific timeouts. The algorithm is iterative and ends once all data batches have been processed, thereby generating a complete result for the input query.

The latter strategy completely bypasses the original query optimizer. This is appropriate for difficult-to-optimize queries (e.g., due to UDF predicates or predicate correlations) where a traditional optimizer fails. However, it adds unnecessary learning overheads for standard queries that are easy to optimize by the original system. To combine good performance for difficult queries with reasonable performance for standard queries, we propose a hybrid evaluation strategy. This strategy divides execution time between time dedicated to plans proposed by the original optimizer and execution time dedicated to the learning based approach. We show in our demonstration that this strategy achieves robust performance in a variety of scenarios.

2.3 Join order learning with specialized execution engine

The previous two strategies are versatile and can in principle work with any DBMS offering an SQL interface. We treat the DBMS as a black box and make no assumptions on the internal processes by which queries are evaluated. For this flexibility, we pay a performance overhead compared to a customized execution engine. Our last evaluation strategy is based on a specialized execution engine, that is tailored towards the requirements of a learning based query evaluation strategy. Most importantly, the execution engine must enable us to switch quickly between different join orders, to repeatedly interrupt processing of a join order with little penalty, and to share evaluation progress as much as possible between different join orders. Among the key ideas to achieve those features, is a multi-way join strategy and a specialized tuple representation. Both aim at keeping intermediate results produced during execution as small as possible to speed up restoring and backing up evaluation progress when switching join orders.

2.4 Formal guarantees on near-optimal expected execution cost

Reinforcement learning methods typically come with formal guarantees bounding the expected regret (which is obtained by making optimal or near-optimal choices during learning). Our goal is to translate those regret bounds into bounds on execution cost regret in query evaluation. More precisely, we define as *regret* the time difference between actual query evaluation time for a given query and evaluation time required by an optimal join order. Our processing strategies offer bounds on expected regret of the form $(1 - 1/Poly(m)) \cdot n + O(\log(n)))$ where n is evaluation time and m is the number of joined tables in the current query ($Poly()$ designates a polynomial). We put those results into context. For traditional optimizers, the difference between execution time of an optimal and sub-optimal query plan can grow exponentially in the number of query tables. Hence, for worst-case queries where simplifying assumptions in cardinality estimation lead to sub-optimal plan choices, the regret of a traditional optimizer could grow as $(1 - 1/Exp(m)) \cdot n$ (where $Exp(m)$ designates an exponential function in the number of joined tables). Hence, our evaluation strategies are provably more robust than traditional query optimization.

3. RELATED WORK

Our approach connects to prior work that collects information on predicate selectivity by evaluating them on data samples [9, 10, 22, 23, 25, 28, 31, 42]. We compare in our experiments against a recently proposed representative [42].

Most prior approaches rely on a traditional optimizer to select interesting intermediate results to sample. They suffer if the original optimizer generates very bad plans. The same applies to approaches for interleaved query execution and optimization [1, 5, 7] that repair initial plans at run time if cardinality estimates turn out to be wrong. Robust query optimization methods [3, 4, 6, 11] assume that predicate selectivity is known within narrow intervals which is often not the case [17]. Prior work [15, 16] on query optimization without selectivity estimation is still based on simplifying assumptions (e.g., independent predicates) that are often violated.

Machine learning has been used in the context of databases to estimate cost for query plans whose cardinality values are known [2, 27], to predict query [19] or workflow [34] execution times, result cardinality [29, 30], or interference between query executions [14]. LEO [1, 38], IBM's learning optimizer, leverages past query executions to improve cardinality estimates for similar queries. Ewen et al. [18] use a similar approach for federated database systems. This works only if past queries are similar and if this similarity can be recognized (slight changes in the query can lead to huge differences in execution time for a given join order). Instead, we focus on learning during the execution of a given query.

Adaptive processing strategies have been explored in prior work [5, 12, 13, 36, 37, 40, 41]. Our work uses reinforcement learning and is therefore most related to prior work using reinforcement learning in the context of Eddies [40]. We compare against this approach in our experiments. Among the differences are for instance the following. First, we use a more recent reinforcement learning algorithm [20] as base. Second, we cleanly separate learning-related data structures from the execution engine, allowing us to perform join order learning on top of existing DBMSs in a non-intrusive manner (Eddies associate single tuples or tuple batches with meta-data). Third, Eddies do not deliberately discard intermediate result tuples but complete them instead. We take great care not to spend a disproportional amount of time on bad join orders and may discard intermediate results if they seem to be part of bad join orders. We could not achieve our formal regret bounds without that feature. In contrast to worst-case optimal join algorithms [32], our guarantees are probabilistic and refer to optimal execution time (not worst-case result sizes). Also, they are valid for arbitrary predicates (no restriction to equality predicates).

4. DEMONSTRATION

We describe in the following the benchmarks and baselines used during the demonstration. We describe the metrics applied to compare baselines as well as visualizations that enable attendants to understand how different approaches work. Finally, we describe the demo organization.

4.1 Benchmarks

We use a mix of traditional benchmarks (typically easy from the query optimization perspective), as well as corner cases that are extremely difficult for traditional query optimizers. Our corner cases focus on two weaknesses of traditional query optimizers: data skew and user-defined predicates. Both factors make it hard to estimate the cardinality of intermediate results. Data skew leads to query predicates that are correlated (i.e., a tuple satisfying one predicate is more or less likely to satisfy another predicate). This violates the predicate independence assumption made by traditional optimizers, thereby leading to erroneous estimates. User-defined predicates, on the other side, correspond to black boxes from the optimizer perspective. Their selectivity must be guessed, leading potentially to large errors. More precisely, we consider the following benchmarks.

TPC-H Variants. TPC-H [39] is one of the most popular benchmarks. It has been designed to challenge the execution engine, rather than the query optimizer. Hence, the original TPC-H benchmark is relatively easy to solve by traditional optimizers. In our demo, we consider the original TPC-H queries as well as several variants. For those variants, we reformulate queries into semantically equivalent versions that are nevertheless much harder to optimize. For instance, we replace all unary predicates in TPC-H queries by user-defined functions with equivalent semantics.

Join Order Benchmark. The join order benchmark is based on real data from the international movie database [21]. Its data is skewed, making cardinality estimation harder than for synthetically generated benchmark data sets. We consider the original benchmark as well as several variants with semantically equivalent queries.

Micro-Benchmarks. We use several micro-benchmarks that challenge the query optimizer due to user-defined predicates or data skew. On the other side, we created micro-benchmarks that make optimization easy as all possible join orders are equivalent.

4.2 Baselines

We demonstrate multiple approaches implemented in SkinnerDB. First, we demonstrate SkinnerDB running on top of existing database systems. Here, we use either a pure learning approach or a hybrid approach (which periodically switches between learned join orders and plans proposed by the original optimizer). We use Postgres [35] as underlying database system for our demonstration (while the approach can be applied to other systems as well). Second, we demonstrate SkinnerDB as stand-alone system, its learning based optimizer running on top of a tailored execution engine. This approach optimizes performance but requires to move data between systems.

We compare our approach against several baselines. We compare against two other database systems: Postgres [35] and MonetDB [8]. Postgres features a mature, cost-based query optimizer. It is therefore representative for strengths and weaknesses of traditional optimizers. MonetDB is optimized for high-performance analytical data processing. We use it to show that even a highly optimized execution engine cannot make up for the effects of badly chosen join orders.

Furthermore, we compare against several query optimization baselines. To make the comparison as fair as possible, we implement those baselines on top of the SkinnerDB execution engine. As a first baseline, we consider Eddies [40]. This is an adaptive processing strategy that uses reinforcement learning as well (but in a different way compared to our system). Next, we consider re-optimization [42], an approach to fix initial query plans based on sampling. Finally, we consider a traditional cost-based optimizer implemented on top of the SkinnerDB execution engine.

4.3 Organization

Attendants can access at least one laptop, running SkinnerDB and the various baselines. The demo machine has all

of the aforementioned benchmarks installed. Participants may use existing queries or modifications thereof. We measure run time for all compared approaches. Also, for all approaches that use our own execution engine, we measure additional metrics such as the number of evaluated predicates. This allows to compare different approaches in different scenarios, illustrating their strengths and weaknesses.

We produce several types of visualizations. Those visualizations allow participants to gain insights with regards to internal processes of SkinnerDB. In particular, we visualize developments in the UCT search tree, capturing the learning process of SkinnerDB. We color tree nodes based on the number of times join orders were tried and based on received average rewards. We visualize the search tree at different stages of the query evaluation process. At the start of evaluation, little is known about what join order is optimal. Hence, the number of tries is distributed over tree parts quite uniformly. As evaluation progresses, preferences for specific join orders begin to form which is captured in the search tree.

5. CONCLUSION

We present SkinnerDB, a novel database system that trades peak performance in standard cases for robustness in corner cases. SkinnerDB learns optimal join orders during query evaluation, offering formal guarantees on expected execution cost. Our system works either as an optimizer layer on top of existing database systems or as a standalone system. In our demo, we allow participants to compare our system against various baselines on different benchmarks.

6. REFERENCES

[1] A. Aboulnaga, P. Haas, M. Kandil, S. Lightstone, G. Lohman, V. Markl, I. Popivanov, and V. Raman. Automated statistics collection in DB2 UDB. In *PVLDB*, pages 1169–1180, 2004.

[2] M. Akdere and U. Cetintemel. Learning-based query performance modeling and prediction. In *ICDE*, pages 390–401, 2011.

[3] K. H. Alyoubi. *Database query optimisation based on measures of regret*. PhD thesis, 2016.

[4] K. H. Alyoubi, S. Helmer, and P. T. Wood. Ordering selection operators under partial ignorance. In *CIKM*, pages 1521–1530, 2015.

[5] R. Avnur and J. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.

[6] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *SIGMOD*, pages 119–130, 2005.

[7] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *SIGMOD*, pages 107–118, 2005.

[8] P. Boncz, K. M.L., and S. Mangegold. Breaking the memory wall in MonetDB. *CACM*, 51(12):77–85, 2008.

[9] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD*, pages 263–274, 2002.

[10] S. Chaudhuri and V. Narasayya. Automating statistics management for query optimizers. In *ICDE*, pages 7–20, 2001.

[11] H. D., P. N. Darera, and J. R. Haritsa. Identifying robust plans through plan diagram reduction. *PVLDB*, 1(1):1124–1140, 2008.

[12] A. Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, 33(1):44–49, 2004.

[13] A. Deshpande, Z. Ives, and V. Raman. Adaptive Query Processing. *Foundations and Trends® in*, 1(1):1–140, 2006.

[14] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, pages 337–348, 2011.

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

[16] A. Dutt and J. Haritsa. Plan bouquets: query processing without selectivity estimation. In *SIGMOD*, pages 1039–1050, 2014.

[17] A. El-Helw, I. F. Ilyas, and C. Zuzarte. StatAdvisor: recommending statistical views. *PVLDB*, 2(2):1306–1317, 2009.

[18] S. Ewen, M. Ortega-Binderberger, and V. Markl. A learning optimizer for a federated database management system. *Informatik - Forschung und Entwicklung*, 20(3):138–151, 2005.

[19] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries – better decisions enabled by machine learning. In *ICDE*, pages 592–603, 2009.

[20] S. Gelly, L. Kocsis, and M. Schoenauer. The grand challenge of computer go: monte carlo tree search and extensions. *Communications of the ACM*, 3:106–113, 2012.

[21] A. Gubichev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[22] P. Haas and A. Swami. Sampling-based selectivity estimation for joins using augmented frequent value statistics. In *ICDE*, pages 522–531, 2011.

[23] P. J. Haas and A. N. Swami. Sequential sampling procedures for query size estimation. *SIGMOD Rec.*, 21(2):341–350, 1992.

[24] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: a survey. *JAIR*, 4:237–285, 1996.

[25] K. Karanasos, A. Balmin, M. Kutsch, F. Ozcan, V. Ercegovac, C. Xia, and J. Jackson. Dynamically optimizing queries over large scale data platforms. In *SIGMOD*, pages 943–954, 2014.

[26] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European Conf. on Machine Learning*, pages 282–293, 2006.

[27] J. Li, A. C. König, V. R. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for SQL queries using statistical techniques. *PVLDB*, 5(11):1555–1566, 2012.

[28] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *SIGMOD*, pages 1–11, 1990.

[29] T. Malik and R. Burns. A black-box approach to query cardinality estimation. In *CIDR*, pages 56–67, 2007.

[30] T. M. T. Malik, R. B. R. Burns, N. V. C. N. V. Chawla, and A. S. A. Szalay. Estimating query result sizes for proxy caching in scientific database federations. In *ACM/IEEE SC 2006 Conference (SC'06)*, pages 102–115, 2006.

[31] T. Neumann and C. Galindo-Legaria. Taking the edge off cardinality estimation errors using incremental execution. In *BTW*, pages 73–92, 2013.

[32] H. Q. Ngo, E. Porat, and C. Ré. Worst-case optimal join algorithms. In *PODS*, pages 37–48, 2012.

[33] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR*, 2017.

[34] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki. PREDIcT: towards predicting the runtime of large scale iterative analytics. *PVLDB*, 6(14):1678–1689, 2013.

[35] PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org/, 2017.

[36] L. Quanzhong, S. Minglong, V. Markl, K. Beyer, L. Colby, and G. Lohman. Adaptively reordering joins during query execution. In *ICDE*, pages 26–35, 2007.

[37] V. R. V. Raman, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, pages 353–364, 2003.

[38] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *PVLDB*, pages 19–28, 2001.

[39] TPC. TPC-H Benchmark, 2013.

[40] K. Tzoumas, T. Sellis, and C. S. Jensen. A reinforcement learning approach for adaptive query processing. Technical report, 2008.

[41] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *PVLDB*, pages 285–296, 2003.

[42] W. Wu, J. F. Naughton, and H. Singh. Sampling-based query re-optimization. In *SIGMOD*, pages 1721–1736, 2016.