Ultron: History-Based Query Optimization at Databricks

Supun Nakandala
Databricks
supun.nakandala@databricks.com

RK Korlapati
Databricks
rk.korlapati@databricks.com

Eric Liang
Databricks
ekl@databricks.com

Yunjia Zhang
Databricks
yunjia.zhang@databricks.com

Yuhao Zhang
Databricks
yuhao.zhang@databricks.com

Andrey Gubichev
Databricks
andrey.gubichev@databricks.com

Kelvin Jiang
Databricks
kelvin.jiang@databricks.com

Yannis Sismanis
Databricks
yannis.sismanis@databricks.com

Mostafa Mokhtar
Databricks
mostafa.mokhtar@databricks.com

Sid Taneja
Databricks
siddharth.taneja@databricks.com

Vuk Ercegovac
Databricks
vuk.ercegovac@databricks.com

Amir Hormati
Databricks
amir.hormati@databricks.com

Amit Shukla
Databricks
amit.shukla@databricks.com

Michalis Petropoulos
Databricks
michalis.petropoulos@databricks.com

Reynold Xin
Databricks
rxin@databricks.com

Ryan Marcus
University of Pennsylvania
rcmarcus@seas.upenn.edu

ABSTRACT

Databricks, a modern lakehouse data platform, operates at massive scale (over 50 million VMs, processing billions of queries per day) with limited data statistics, making traditional cost-based query optimization challenging. Databricks uses adaptive query execution (AQE) to mitigate many runtime inefficiencies, but certain decisions, such as join operator selection, could benefit significantly from accurate prior knowledge. We present Ultron, a production-grade history-based query optimization (HBO) framework recently deployed at Databricks. Since analytical workloads are often repetitive, Ultron can learn from prior executions of the same query and apply precise, subplan-level historical insights during optimization. To achieve this, Ultron introduces a three-tier architecture consisting of (1) a distributed cache for query history, (2) a low-latency local prediction layer with data-drift awareness, and (3) modular optimization applications that modify query plans without risky exploration. We demonstrate three applications of Ultron using data from the Databricks fleet and synthetic benchmarks, showing that Ultron improves the median latency of eligible joins by 25% in production, optimizes approximately 23% of all joins across the fleet, and achieves 96% post-hoc decision accuracy. Ultron operates with modest overhead (< 5% median additional optimization time) and scales elastically with load, demonstrating that precise history-based optimization can safely complement adaptive execution in large-scale cloud data platforms.

PVLDB Reference Format:
Supun Nakandala, RK Korlapati, Eric Liang, Yunjia Zhang, Yuhao Zhang, Andrey Gubichev, Kelvin Jiang, Yannis Sismanis, Mostafa Mokhtar, Sid Taneja, Vuk Ercegovac, Amir Hormati, Amit Shukla, Michalis Petropoulos, Reynold Xin, and Ryan Marcus. Ultron: History-Based Query Optimization at Databricks. PVLDB, 19(12): 4358 - 4371, 2026.
doi:10.14778/3827998.3828038

1 INTRODUCTION

Databricks is a lakehouse data platform used for a wide variety of applications. The lakehouse nature of Databricks means that query optimization and execution is often done with few data statistics, unlike traditional systems which might maintain robust statistics, like histograms, over each data column [65]. Even if similar statistics could be maintained at the scale of an entire lakehouse, these statistical techniques are imperfect, and often lead to suboptimal query plans [23]. While Databricks can compute and maintain such statistics, Databricks has developed an alternative route to perfect ahead-of-time query planning: instead of trying to guess the ideal query plan upfront, Databricks uses adaptive query processing (AQP) [61] to mitigate plan suboptimalities at query runtime.

While AQP significantly increases the robustness of query plans, there are certain elements of query execution that, to achieve maximum performance, must be known ahead of time. For example,

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828038

when partitioning an intermediate result for some distributed operator (like a sort), one must divide the data into a chosen number of partitions. If too many partitions are created, per-partition overhead becomes substantial, but if too few partitions are created, some nodes may be overloaded, or may experience out-of-memory errors. AQP can mitigate the latter case: AQP can select the largest partition and break it into smaller pieces, preventing the worst consequences of oversized partitions. But this re-partitioning has a cost; If the optimal partition count (which depends on data distribution, and is thus difficult to estimate) was known ahead of time, we could avoid both out-of-memory crashes and re-partitioning.

Fortunately, workloads on Databricks (like other OLAP engines [2, 51, 60, 66]) are repetitive. For the Databricks runtime, a tool for interactive data analysis, between 45% and 64% of queries had been previously executed verbatim in the prior week. For Databricks SQL (DBSQL), our platform for BI and analytics, the rate is between 67% and 80%. Furthermore, nearly 85% of joins seen across the platform are repeated (i.e., a join is repeated if the join predicate and subtress are semantically equivalent). These repetitive joins account for 73% of the total execution time on the platform.

This repetition is an opportunity. Here, we introduce Ultron, a history-based query optimizer (HBO). Like other history-based query optimizers [8, 17, 38], Ultron can look up statistics about a query's previous executions to get "perfect" information (modulo data changes). Thus, Ultron allows Databricks to *learn from its mistakes* and continuously improve query performance for important, high-volume queries. Ultron's design integrates with our current query optimizer, strategically inserting history-based decisions, rather than replacing the traditional query optimizer outright.

Designing a history-based query optimizer for Databricks required us to overcome several key challenges. First, query history needs to be stored both globally and efficiently. Since Databricks users can start, stop, grow, and relocate clusters, or use entirely serverless environments, there is no consistent "user cluster" on which to store data. However, simply storing history data in a separate distributed database would introduce unacceptable overheads on query optimization, since each call to the query optimizer would need to fetch data over the network. To resolve this, we use *Softstore*, a distributed key/value cache custom-built for Databricks. Instead of having clients poll Softstore for query history, Softstore actively pushes query history to clusters. Ultron is carefully designed to work with and without query history, so latency spikes or outages from Softstore will not impact query optimization availability.

Second, querying the log of executed queries is not trivial, since the log itself can be large. To make log queries both efficient and servable using compact data, we implement a framework called QuickPredict which can hash executed query plans and associate values with them, supporting constant-time local queries. QuickPredict can handle exact, literal insensitive matching, along with fuzzier, "near-match" style lookups. QuickPredict also efficiently tracks (by incorporating it into the hash function) the size of relevant tables when a historical query was executed, ensuring that Ultron will not use "ancient history" from when a particular database was much smaller (or larger).

Finally, Ultron applications, specific types of optimizations that make use of history data, must be carefully designed. Since every query Ultron optimizes will itself become history, Ultron's actions

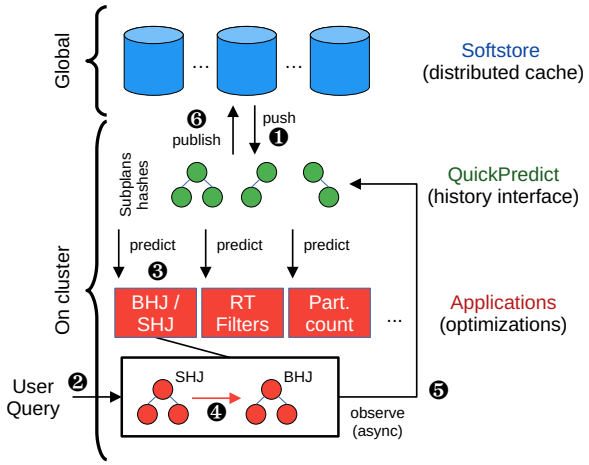

Figure 1: Ultron system model. A global distributed cache pushes query history data to a cluster-level cache. Various Ultron applications then use this history cache to optimize query plans. When new history values are observed, they are asynchronously saved back to the cluster cache, then published to the global cache.

impact the future data that Ultron sees. Traditional solutions to this problem, like reinforcement learning [46] or Bayesian optimization [37], are not acceptable here, since exploration would introduce variability into query plans. Ultron applications must be careful to ensure queries do not "flip-flop" between different plans, and that each decision is highly likely to improve query performance. Combining these two conditions, Ultron applications must, with rare exception, monotonically converge to a high quality plan.

Experimentally, we show that Ultron has a significant impact on Databricks' production workloads (improving the median eligible join's latency by 25%). We evaluate the parts of Ultron that are still under development using synthetic data, showing that Ultron can reduce the latency of TPC-H queries by up to 40%.

This paper makes the following contributions:

- We present the design of the Ultron framework for history-based query optimization, which serves Databricks' production fleet.
- We showcase three applications of Ultron to solve specific query optimization problems using history data.
- We evaluate Ultron on the Databricks' production fleet, highlighting lessons we learned along the way.

The rest of this paper is organized as follows. In Section 2, we describe Ultron's high-level architecture and design. In Section 3, we describe each component of Ultron and how they work together. We discuss applications of Ultron to various query optimization tasks in Section 4, and show experimental results in Section 5. In Section 6, we discuss key related work. Finally, we discuss remaining challenges in Section 7 and conclude in Section 8.

2 SYSTEM MODEL

Ultron's goal is to robustly apply insights from query history to query plans in a way that strictly (or at least nearly-always) improves performance. Intuitively, such history-based optimizations create a "closed loop" that allows a DBMS to learn from its mistakes, a longstanding goal in query optimization research [42].

Challenges Building a history-based query optimizer for Databricks brings about a number of key challenges. First, since Databricks clusters can be created or destroyed, query history needs to be stored outside of the user cluster. But, querying an external service in the hot path of the query optimizer would introduce unacceptable latency. Thus, history data must be preserved globally, but be accessible locally. Synchronizing global and local state must not interfere with query performance. Second, using query history to optimize queries requires (1) matching query subplans to history records, (2) ensuring that query plan changes do not cause large regressions, and (3) ensuring that query plans eventually converge to a high quality plan instead of "flip-flopping" between different plans. Preferably, Ultron should provide a unified framework for many different applications of history-based optimizations, such that each application does not have to "reinvent the wheel."

Designing a good application of history data to query plans can be surprisingly challenging. For example, a frequently-floated idea is to replace error-prone cardinality estimations [23] with observed values, falling back to the estimator when historical values are absent. This straightforward idea fails miserably in practice: because traditional cardinality estimators tend to underestimate (due to assuming independence), a join search procedure will "flee from knowledge," [30], preferring plans where most estimates come from the (biased-low) estimator instead of the true values from history. The result is query plans changing drastically between iterations, and converging to a suboptimal plan.

Ultron Ultron solves this problem with a three-tier architecture, illustrated in Figure 1. The top two tiers are used throughput Databricks for a wide range of applications, and we discuss them here in the context of Ultron. The topmost layer, Softstore, is a distributed key-value cache across the entire Databricks fleet. We do not directly query Softstore at query time (which would add too much latency to query optimization) – instead, we use Softstore to hold history data outside of a customer's cluster, so that history data is preserved when users shut down, resize, or clone their clusters. At the second tier, QuickPredict, Ultron stores a local database of query plans. When a cluster starts, Softstore pushes the user's query history to the local QuickPredict database. The final tier contains Ultron applications. Each application asks QuickPredict for estimates of a subplan's properties. If those properties are available, QuickPredict returns them, and the Ultron application can modify query plans accordingly. Finally, after a query executes, query history information is asynchronously saved to QuickPredict.

Example Figure 1 shows an example of how Ultron uses query history to promote a shuffle hash join to a broadcast hash join. When a cluster is created, and periodically thereafter, Softstore, a global distributed cache, pushes ❶ a copy of the user's history to the cluster's QuickPredict cache. When a ❷ user query arrives, various Ultron applications examine the query plan tree and potentially make modifications. The "BHJ / SHJ" application helps Databricks select the optimal join algorithm (broadcast hash join or shuffle hash join). To make these decisions, Ultron applications will ❸ ask the QuickPredict cache for an estimate of some quantity, such as the cardinality of the join. Based on QuickPredict's estimate (or lack thereof, if there is no match in the cache), the application may ❹ modify the query plan, such as by promoting a shuffle hash join to a broadcast hash join. After the query is executed, new history is observed, and asynchronously (so as not to delay query completion) ❺ saved in the local cluster cache. The local cluster cache periodically ❻ publishes changes to Softstore, so that future clusters or serverless instances can use the history cache.

Security & privacy Ultron was carefully designed with user security and privacy as a top priority. To ensure that one user's query history can never be viewed by another user (or even impact another user's workload), Softstore partitions its data storage by customer, so even an adversarially-written Softstore request cannot access another customer's data. Even Databricks engineers are unable to access production Softstore entries without explicit permission from the customer (e.g., via a support ticket). In addition to access controls, we also never store any query text or literals,[1] instead keying each history record with a hash of the query plan or subplan. This means that, even if Softstore were breached, query plans could not be recovered from the stored hash values alone. Softstore does record some history trace values, like selectivities, which are protected by Softstore's access controls.

Next, in Section 3, we explain each layer of Ultron's architecture and how they scale to meet Databricks' needs. We describe three specific history-based applications we have developed that use the Ultron framework in Section 4.

3 ULTRON

Ultron's architecture is separated into three tiers: Softstore is described in Section 3.1. The second tier, QuickPredict, is described in Section 3.2. Finally, Ultron applications are described in Section 3.3.

3.1 Softstore

Softstore is a distributed key-value store built for Databricks' needs, using a best-effort eventually consistency model that prioritizes availability and latency. Softstore shards data based on customer ID, which helps with tenant isolation (high usage clients get their own shards, low usage clients can be co-located) and security (it is impossible to access another customer's data from within a customer's scope). Softstore also supports state transfer, allowing the system to migrate one shard to another without downtime.

Softstore groups keys into namespaces, which can each have their own (1) capacity reservation, (2) time-to-live (TTL), and (3) eviction strategy. For Ultron, capacity is directly proportional to the number of distinct queries a customer runs. The TTL value for each record is optimal when a record lives long enough to be read again when a query repeats, so in theory the TTL value could be set based on the empirical query repeat frequency. For simplicity, we use a static, 1 week TTL value for all query history records, breaking ties by number of rows scanned. Ideally, a query's history trace

[1] This also helps reduce cache size.

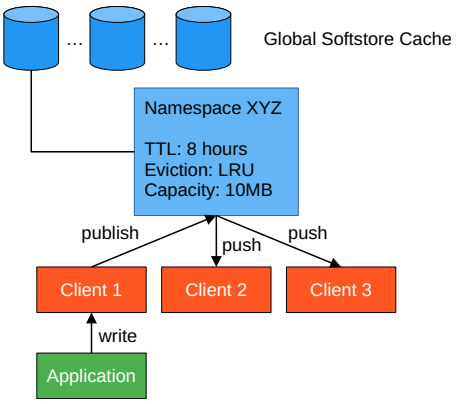

Figure 2: Softstore's namespaces and local caching. Instead of clients pulling data from the global cache, the global cache pushes data to clients that are subscribed to a namespace. Writes from clients are published to the global cache, potentially in batches.

should be kept in proportion to the acceleration that trace provides, not just how often the query runs. For example, if a history trace allows an hourly query that takes 10 minutes to take 2 minutes, that trace is arguably more valuable than a query run every 30 minutes that goes from 1 minute to 55 seconds. We leave adjusting the eviction policy to future work – for most customers, records are never evicted by the LRU policy, as the total number of distinct eligible queries rarely exceeds the namespace capacity.

A limitation of Softstore is that keys must not exceed 64 bytes. This means using a textual representation of a query or subplan as a key is impossible [52]. Ultron instead keys each history trace with a hash. This has some collision risk, but we found the impact of collisions to be negligible in practice. Storing only hash values instead of full query plans also has operational security benefits.

Local caching Since Softstore is widely used across Databricks for applications such as catalog access and statistics, Softstore also supports *local caching*, depicted in Figure 2. When a client requests that a namespace is cached locally, Softstore will push the namespace to the client when the client node comes online, and periodically update the client when the namespace changes. Unlike traditional tiered caching, if Ultron fails to find a key in the local cache, the lookup returns nothing – a request is *not* sent to the global cache.

The "push" model has many advantages compared to the traditional "pull" model in which the client calls the remote server on a cache miss and saves the result. First, the "push" model means that a client's very first lookup for a particular key could still hit the local cache, since that key may have been pushed to the client by Softstore. Second, the "push" model also allows Softstore responsively adjust its publication rate based on load.

Softstore's local caching and eventual consistency model means that stale data could be read from the local cache, and that data written to the local cache may not be flushed to the global database immediately. This creates complications for applications like catalog management, but Ultron is almost a perfect user of this system: stale history reads are OK, since query history rarely changes. If a key is in the global cache, but has not been pushed to the local cache yet, the optimizer will not see that piece of history information, but in this case we are no worse off than we were before Ultron.

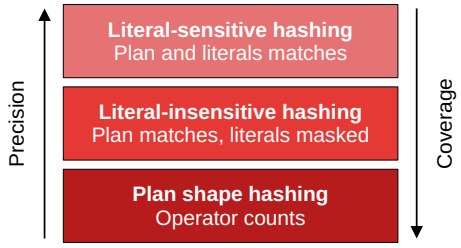

Figure 3: QuickPredict plan hashing modes. Different hashing modes trade increased coverage with decreased precision.

3.2 QuickPredict

QuickPredict is a general-purpose history-based prediction tool designed for low latency. QuickPredict is used widely across Databricks, including in Databricks' workload management system (scheduling, admission control, and elasticity). In Ultron's context, QuickPredict is responsible for marshaling history traces (observations) back to the local cache, and serving those traces (predictions) from the local cache to Ultron applications (the management of this cache, and its distribution, is discussed in Section 3.1). QuickPredict does this by providing an interface with two functions, `observe` and `predict`:

- Reads (`predict`): given a subplan and a requested statistic (e.g., cardinality), QuickPredict computes the subplan hash and performs a hash-value only check in the cache for a corresponding key (Softstore limits key length to 64 bytes, so the entire subplan cannot be stored – we find that collisions are negligible in practice, and hash-only checks are fast). If there is an entry for the key, and the requested statistic is present, QuickPredict returns that value. Otherwise, QuickPredict returns "not found," and the Ultron application does not modify the query plan.
- Writes (`observe`): after a query executes, observed values for various statistics are asynchronously written back to the local cache, and eventually synced with remote Softstore (and thus pushed to new clusters). This creates a decoupling from the statistics used by `predict` and the values sent to `observe`: if we think a value from a query execution might be useful for a future Ultron application, we can add an `observe` call for it so that it is available in future versions of Databricks.

Subplan hashing QuickPredict depends on subplan hashing to match subplans currently under optimization to entries in the QuickPredict cache. QuickPredict supports three levels of hashing that trade coverage (the likelihood of finding a match) with precision (how close the retrieved historical record is to the current query plan), as illustrated in Figure 3. An Ultron application can specify a hashing level when calling `predict`.

The most precise hashing type is *literal-sensitive hashing*; this is a hash of every plan node and the exact literal predicates (e.g., instantiated template values) used in the query. A literal-sensitive hash match means that the current query and the historical record came from semantically-identical SQL (e.g., the only difference in the SQL was something like the syntactic join or predicate order –

all literals match exactly). These matches have the highest precision. A *literal-insensitive hash* is the same as a literal-sensitive one, except the literal values (e.g., query parameters) are ignored. These hashes map instances of the same query template to the same history records, which can be helpful when the optimal query plan is not sensitive to the template parameter, which is the case for some templates [15, 18, 50]. Finally, a *plan shape hash* simply hashes the vector of operator type counts, a common featurization strategy for latency predictors [31, 60]. These matches have the highest coverage, but the least precision, and are useful for high-level and coarse-grained estimates about subplans.

Since OLAP workloads in the Databricks fleet are highly repetitive, most Ultron applications use literal-sensitive hashing to get the greatest possible precision.

Data drift As data drifts over time, stored history records may become stale, possibly leading Ultron applications to make incorrect decisions. Like prior work, we find that even drastic data drift in established databases has only a small impact on optimal query plans [48, 63], but drastic size changes early in a database's lifecycle can have a large impact (e.g., as data goes from a single node to requiring distribution). Thus, QuickPredict additionally keys each hash to the size of the underlying database scans, which we believe is the dominant component of impactful data drift. To do this, QuickPredict computes, for a given subplan, a size factor S:

$$S = \left\lfloor \log_2 \left(\prod_{i \in \text{scans}} \text{scan size} \right) \right\rfloor = \left\lfloor \sum_{i \in \text{scans}} \log_2(\text{scan size}) \right\rfloor$$

Intuitively, since S increases with the log of scan sizes, S increases slowly for larger databases where a single insert is unlikely to substantially alter the data distribution, and S increases quickly for smaller databases where a single insert could shift the distribution. The size factor S is hashed into each key, regardless of the hashing type used. When the database grows to the next size factor, history records from $S-1$ are no longer accessed, and are eventually evicted by Softstore after their TTL expires.

The size factor also increases the precision of plan shape hashing. Without the size factor, a simple `SELECT * FROM X` query would hash to the same value regardless of `X` – with the added size factor, plan shapes only collide when they scan similar amounts of data.

Avoiding performance cliffs The precision of QuickPredict's hashes are good for ensuring that distinct queries do not erroneously share history, but it also introduces a risk of *performance cliffs*. If a table size grows into the next size factor $S + 1$, entirely dropping history data for the next query to access that table might result in a significant increase in query latency that is disproportionate to the amount of data added. Similarly, if a user modifies a query predicate slightly, the query hash will change. To avoid performance cliffs like these, QuickPredict probes first for the query's exact hash, but if one is not found, additionally probes for the same query with $S \pm 1$, and with literal-insensitive hashing. This ensures that queries with small data changes or small predicate changes will still get history data. If the predicate change or data size increase actually causes a change in a recorded value, this new change is recorded once the query finishes executing, and the next iteration of the query will find an exact hash match.

3.3 Usage

Ultron applications consume the QuickPredict API (Section 3.2) and modify query plans based on query history. Here, we describe the design constraints on Ultron applications. We discuss three Ultron applications developed at Databricks in Section 4.

When Ultron modifies a query plan, the recorded history for that query plan will also change. This "closes the loop," allowing the Databricks platform to learn from its mistakes autonomously, but it also introduces complexity for Ultron application authors. By modifying a query plan, an Ultron application essentially modifies its inputs in the next iteration. Ultron applications thus must be careful to ensure **convergence**. Customers do not appreciate their queries being fast one day but slow on another [63, 66]. It is acceptable for Ultron to incrementally change a plan across multiple iterations if each iteration improves upon the last, but eventually Ultron must ensure that a repeatedly executed query eventually reaches a steady state (a fixed point) with high probability.

Ensuring both convergence and that each step is an improvement is a challenge. Many reinforcement learning algorithms eventually converge to stable policies [46, 73], but they do so with frequent "exploration" steps to learn the policy space. Ultron seeks convergence with **no exploration**, since exploration carries a high risk of slowing down a query.[2] In general, it is not possible to find an optimal policy without some exploration or offline search [76] – this means that Ultron applications must be conservative, only adjusting query plans in ways that are likely to improve them and never exploring risky decisions. This tradeoff means that Ultron will almost always improve plans, and will always converge to a stable plan, but might not find the optimal plan.

4 APPLICATIONS

In this section, we showcase three Ultron applications we have developed at Databricks. More applications are under development. Each application uses the framework described in Section 3 to modify query plans based on prior executions of similar queries.

4.1 Join operator selection

Databricks supports two main types of distributed hash joins: shuffle-based hash joins (SHJ) and broadcast-based hash joins (BHJ). Broadcast hash join is used when the build side of the hash join is small enough to replicate across every node in a cluster. By broadcasting the small build side to every worker, BHJ allows any partitioning of the probe side to be processed independently by every worker. Shuffle hash join is used when the build side of a hash join is too large to replicate. SHJ works by co-partitioning the build and probe sides of the hash join on the join key, then distributing those partitions evenly across the cluster.

Picking between SHJ and BHJ is difficult because it requires accurate cardinality estimates. Since cardinality estimates are notoriously inaccurate [23], the Databricks optimizer errs on the side of caution, preferring to select SHJ unless the optimizer is highly confident that the build side is small. This is because selecting BHJ with a build side that is too large will heavily use network resources and likely cause out-of-memory errors or worker crashes, but using a SHJ with a build side that is small will still execute, albeit slowly.

[2] Ultron can make specific explorations if AQE is able to correct those errors.

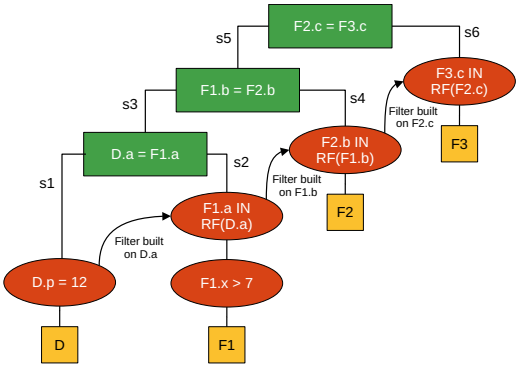

Figure 4: New runtime filters being progressively added to a query each iteration. In the first iteration, we build a runtime filter of `D.a` that is pushed to the scan of `F1`, but it is not clear that building a runtime filter over `F1.b` is worthwhile. After execution, we can observe that `F1.b = F2.b` is selective, and then choose to build the filter.

We developed an Ultron application that records the size of each build-side table in a query plan, and then uses this exact information to choose between SHJ and BHJ for future query iterations. If Databricks conservatively selects SHJ for a particular query where BHJ would have been optimal, QuickPredict will observe the true size of the build. Then, in the next query iteration, the application will call `predict` with literal-sensitive hashing to get the exact cardinality of the build side. This significantly derisks the decision to choose BHJ, since the build side is guaranteed to not have changed more than a size factor (see Section 3.2).

Analysis We can evaluate this Ultron application against the two criteria given in Section 3.3. In terms of convergence, this application converges in a single step, since the decision between BHJ and SHJ does not impact the size of the build side. In terms of avoiding regression, this application only promotes SHJ to BHJ when there is clear historical evidence for the promotion being safe.

4.2 Runtime filter injection

Queries with multiple joins can create large intermediate results, which can dominate execution [7]. A suite of techniques for dealing with this problem take advantage of the sideways information passing [20], in which a Bloom filter is built over the keys of a hash join and pushed down into the probe side, potentially past other joins, to leaf scan operators [40] (the Yannakakis [62] algorithm is a generalization of this trick [6, 43, 55, 70]).

Databricks uses *runtime filters* [61] to reduce intermediate joins sizes, which are adaptively selected during query execution. It is difficult to know ahead of time whether or not a runtime filter for a particular join should be built, as estimating join selectivity is one of the hardest parts of cardinality estimation [23].

Example Even estimating the power of a particular runtime filter from history data is difficult. Consider the example illustrated in Figure 4. The lowest join, between `D` and `F1`, is a join of two base tables, so the optimizer can estimate that the predicate on `D` (`D.p = 12`) is selective (s_1), and push a runtime filter to the scan of `F1` (note that this does not require estimating s_3). But, it is difficult to know if building a subsequent runtime filter from the intermediate join result (s_3) and pushing it into the scan of `F2` will be worthwhile, since the optimizer does not know if `F1.x > 7` or the initial pushed filter will be selective enough to make s_3 a viable filter candidate. So, the conservative defaults in the optimizer only create the initial runtime filter pushed to `F1`.

On the query's second iteration, the exact size of the first join's output is known (s_3). Assume this intermediate result is small. Then, the optimizer can safely use s_3 to generate a new runtime filter and push it to `F2`. To do this, Ultron stores the selectivity of each join both before and after applying a runtime filter. However, the optimizer still does not know the resulting selectivity of this new filter on `F2` (s_4), nor the ultimate size of the second join's output (s_5), so the conservative option is again not to build a filter for `F3`. Just as before, after the second iteration, s_5 is observed, and the correct decision about creating the final cascaded filter for `F3` can be computed for subsequent runs.

Analysis We can evaluate this Ultron application against the two criteria given in Section 3.3. While each additional runtime filter added is likely to be worthwhile (since the cardinality is directly observed), convergence is not automatically guaranteed. How do we know that Ultron will not add or remove filters in a "flip-flopping" way? The answer lies in the monotonic nature of runtime filtering: *once a runtime filter is added, it strictly reduces or maintains downstream cardinalities, meaning it will not invalidate previously added filters.* Since a cardinality is observed and a runtime filter is computed to be worthwhile, it remains effective unless the underlying data changes significantly. The most likely way such a shift could occur is by a large amount of new data being added, in which case the size factor (Section 3.2) would change, causing the optimization process to start over. Therefore, while we cannot guarantee that Ultron eventually finds the globally optimal set of runtime filters, we can guarantee that it safely converges on a local optimum (where all existing filters are worthwhile and no new filters meet the threshold). This approach is imperfect, but it successfully avoids the failure mode of highly variable query latency.

4.3 Partition count selection

Whenever a distributed operator partitions data (e.g., a sort or a shuffle hash join), the system must choose the number of partitions to create. Creating too many partitions can create unnecessary per-partition overhead, but creating too few partitions can lead to certain workers becoming overloaded [72]. The correct partition count is data-dependent, and thus difficult to estimate. Adaptive engines like Photon [5] can fix the case where partitions are too large by breaking large partitions into smaller ones at runtime, which comes at a cost, but is preferable to overloading a node.

Ultron can use history to avoid some of the cost of adaptive repartitioning. When the engine finds that partitions are too large and breaks them into smaller pieces, we create a history record. When the query is repeated, Ultron checks whether or not a previous execution of the query required repartitioning. If a previous iteration required re-partitioning, we use the final partitioning scheme of the previous iteration directly. This produces results identical to if adaptive repartitioning had run, but without the overhead.

Analysis Of the three applications discussed in this paper, partition count selection is the simplest to analyze. Since we simply compute the same partitioning scheme that would have been computed without Ultron in a faster way, it is clear that query latency will strictly improve and that plans will not "flip-flop". The only reason why the selected partitioning strategy for a query would become significantly suboptimal would be if the underlying data distribution changed. Since the most common way for this to happen is database growth, QuickPredict's size factors (Section 3.2) will cause adaptive partitioning to run again when the database grows.

4.4 Future applications

We plan to add more Ultron applications beyond the three listed here. For example, we plan to develop an Ultron application that helps choose which relation should be the build or probe side of a hash join based on saved cardinality statistics. Additional join operator types – like specialized range joins or skew joins – could also be selected based on history (i.e., if a specialized join operator was incorrectly applied, switch to a regular join operator, and vice-versa). In general, it may be possible to build an Ultron application to disable any misapplied optimization, as long as history data can reveal whether or not a particular optimization is beneficial for not. This substantially de-risks new optimizer rules, since if an optimizer rule misbehaves, it could be disabled by Ultron in all future iterations of the query.

5 EXPERIMENTS

Our experimental study primarily investigates the Ultron applications we outlined in Section 4. We also validate that QuickPredict and Softstore (see Section 2) are scalable and do not introduce significant overhead.

RQ1. How well does Ultron's join operator selection work? This is Ultron's most developed application, and we present evaluation data from Databricks' fleet. We present both a latency and accuracy analysis. See Section 5.1.

RQ2. Can Ultron be used to determine which runtime filters to build, iteratively improve those decisions over time, and converge to stable plans? We evaluate this capability on synthetic queries (e.g., TPC-H). See Section 5.2.

RQ3. Can Ultron help choose shuffle partition counts to avoid disk spilling and out-of-memory errors? We evaluate this capability on synthetic queries. See Section 5.3.

RQ4. Does Ultron's design – building upon QuickPredict and Softstore – scale to the size of Databricks' fleet? See Section 5.4.

Since each of Ultron's applications is in different stages of development (i.e., deployment, benchmarking, testing), we do not have a unified experimental setup. Instead, we explain the experimental setup in each section. Unless otherwise noted, all experiments were conducted on Databricks Runtime v18.

5.1 Join operator selection

Ultron's most developed application is choosing between shuffled hash join and broadcast hash join (see Section 4). Correctly selecting between these two operators is critical to avoid costly network usage (e.g., choosing a broadcast hash join when the relation to broadcast is large), avoiding disk spill (e.g., shuffling the wrong side of the hash join), and avoiding stragglers (e.g., shuffle hash join can lead to variance in how long each partition takes to complete). Getting this choice right ahead of time is difficult because cardinality estimation is notoriously difficult [23].

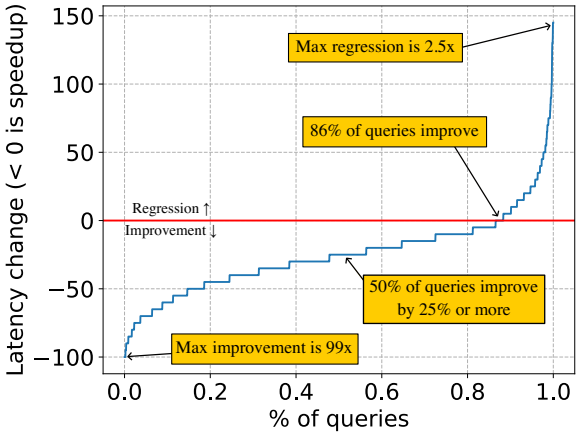

Figure 5: CDF of improvement based on observed pairs. Since query time is not measured in isolation, results may be skewed by other operations on a cluster when a query was executing. The line at 0 indicates the pre-Ultron performance.

5.1.1 Does Ultron's join operator selection improve query latency? To evaluate Ultron's learned join operator selection, we deployed Ultron in a pseudo-A/B test on the Databricks fleet. A customer query would randomly either be processed by Ultron, or processed normally. After two weeks, we searched query logs for pairs of queries with one Ultron execution and one non-Ultron execution, then computed the relative performance improvement (or regression) for Ultron. The CDF of these query pairs is plotted in Figure 5.

This experiment is far less controlled than we would have liked: while we find many qualifying query pairs, their measured execution times are on real customer clusters which may be running other queries, or may elastically grow or shrink.

That said, we find that **Ultron improves latency for 86% of eligible queries.** The median eligible query speeds up by 25%. Of the 14% of queries that regress, the maximum regression we observed was 2.5x. Ultron can lead to regressions in join operator selection when Ultron believes a highly-skewed table is large enough to be shuffled, since the skew can result in the shuffle sending the majority of data to a single node. In these cases, Photon's [5] adaptive query executor can cancel the operator and replan it using a broadcast [61], although this still has some performance cost. Other factors – like cost model errors – could also play a role.

Discussion The most common corrective action taken by Ultron's join operator selection application is promoting a shuffle hash join to a broadcast hash join in the middle of a large join tree. Since the Databricks optimizer cannot be certain of the size of intermediate query results (either because of a lack of statistics in the lakehouse environment or because of cardinality estimation errors), the optimizer is tuned to be conservative, and select shuffle hash join as the "safer" option, since broadcasting a large table could crash nodes

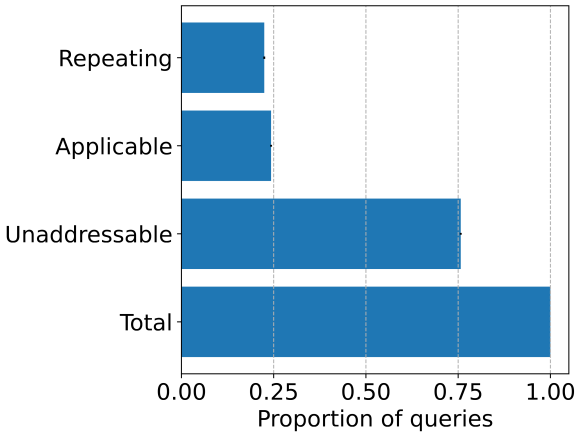

Figure 6: Proportion of queries eligible for Ultron, normalized to 1.0. About 75% of queries are excluded by the criteria in Section 5.1.2. Of the remaining 25%, nearly all (23%) repeat, making them candidates for optimization. Results are computed from samples, with low errors (see small error bars).

or cause network contention. Shuffle hash join must partition both sides of the hash join in order to distribute them, and when a particular partition is deemed "too large" by the AQE (skew mitigation), that partition is rewritten into smaller partitions (a costly operation). Thus, while using a shuffle hash join when a broadcast hash join would have sufficed is safe, the system incurs two performance penalties: one for rewriting partitions, and another for the shuffle itself. By promoting unnecessarily conservative shuffle hash joins to broadcast hash joins, both of these penalties are avoided.

5.1.2 How many joins can Ultron optimize? Not every join can be safely swapped from a broadcast hash join to a shuffle hash join: for example, if the join is part of a streaming operation, a particular join operator might be fixed by the layout of data output by the previous operators. Ultron further excludes from consideration joins that are not executed on Photon [5] (e.g., joins using the traditional Java Spark [64] executor), joins that are not in `SELECT` queries (i.e., joins in ETL jobs where past behavior might not predict future behavior), and joins that do not use Databricks' adaptive query executor [61] (this is an over-restriction, but for any join where it is safe to change the join operator mid-flight, it is also safe to change it pre-flight).

Figure 6 shows the distribution of eligible joins across the fleet, normalized to 1.0. An "Unaddressable" join, representing a little over 75% of the workload, is a join that is excluded from consideration for one of the reasons listed above. Of the remaining 25% of queries, labeled "Applicable," Ultron can potentially be used. In order for Ultron to change a join operator, a join must be repeated (either exactly or a fuzzy match, see Section 3). Almost all applicable queries do, in fact, repeat (23%), meaning that **Ultron currently optimizes about 23% of all joins processed by Databricks.** Combined with the result from Figure 5, this means that Ultron improves the latency of 20% of all join queries processed by Databricks. As we continue to develop Ultron, we hope to lower the percentage of unaddressable queries, but our conservative approach was driven by engineering necessity.

Table 1: Accuracy of Ultron's join operator selection algorithm evaluated on observed query pairs (first row) and on all applicable joins using a post-hoc cost model (second row).

Data	Ultron Accuracy	Always Shuffle	Proportion
Paired (queries)	0.86	n/a	0.002
Post-hoc (joins)	0.96 ± 0.007	0.56 ± 0.006	0.23

5.1.3 How accurate is Ultron's decision? While query latency is the most important metric for evaluating Ultron, we also tested Ultron's accuracy: when Ultron chooses between shuffle hash join and broadcast hash join, how often is Ultron's decision correct?

Unfortunately, it is difficult to evaluate the accuracy of the predictor on real data, since the counterfactual is unavailable. For example, if Ultron picks a shuffle hash join for a query, Ultron never learns how fast a broadcast hash join would have been. Thus, we adopt a *post-hoc* (after-the-fact) analysis. If Ultron selects a broadcast hash join, we approximate how effective a shuffle hash join would have been, and vice versa, using the Databricks cost model with known cardinalities (all cardinalities are known post-hoc). We can also measure Ultron's accuracy on the paired queries from Section 5.1.1, but for these queries we only know if the *entire* query improved or regressed (each query may contain multiple joins).

Table 1 shows the results: for the post-hoc analysis, which considers each *join* instead of each *query*, we find that **Ultron's join operator selection decisions are approximately 96% accurate** across all eligible queries. By comparison, a simpler strategy of always choosing a shuffle join – the safest choice to avoid OOMs – is only 56% accurate. For completeness, we also approximate Ultron's accuracy on the paired query data by making the strong assumption that each query has an equal number of joins, and that each join is equally important. Here, Ultron's accuracy is measured at 86%. We emphasize that this value relies on strong assumptions, and should only be considered alongside the post-hoc analysis.

The gap in accuracy between the paired measurement and the post-hoc measurement can be explained by the noise factors in both measurements. Paired queries may have been issued at different times under different operating conditions, and post-hoc analysis via a cost model can only be as accurate as the cost model itself. In general, **we find that evaluating learned systems components on production systems is challenging**. Users will not accept high variance in query performance, so a query-level A/B test is not an option. A customer-level A/B test is possible, but lacks statistical power because of the large diversity among customers; if you have a few customers with outlier workloads, the A/B test results will be dominated by where those customers are placed.

5.2 Runtime filters

Runtime filters are filters that help transfer an induced join predicate from one table to another. A key problem is deciding when to build such a filter. If a join is between an unfiltered primary key and an unfiltered foreign key to that primary key, building and probing a filter would be a complete waste, since no rows are removed. However, when this is not the case, runtime filters can reduce intermediate result sizes, and thus query latency.

Databricks uses AQE to decide when to build runtime filters, but, as explained in Section 4, this approach is not fully optimal. Since AQE is tuned conservatively, AQE will often miss runtime filters that would have been beneficial. Ultron seeks to correct those cases using history. While we have not deployed Ultron's runtime filter approach, we have tested it on synthetic data.

5.2.1 Can Ultron improve runtime filter selection? Here, we evaluate Ultron's ability to select runtime filters using query history. We use the TPC-H [34] benchmark at scale factor 1000, and an 8-node cluster of `rd-fleet.xlarge` instances. Each query is run three times, once to warm the cache, once to populate the history data, and once to measure execution time. Queries are always run three times even if the technique being evaluated does not use history, in order to avoid measuring caching effects. Only the third query execution is measured and reported.

To assess whether or not Ultron picks good runtime filters, we compare against two baselines. In Figure 7, we compare Ultron's decisions against the standard Databricks optimizer when the optimizer is given perfect cardinality estimates. **Ultron's query latency is consistently within a few percentage points of the omniscient optimizer.** Since Ultron's history-based technique can record the actual efficacy of the filter, Ultron can even outperform the omniscient optimizer, albeit by a very small margin (e.g., q11).

In Figure 8, we compare Ultron against the Databricks optimizer with extended statistics (which are sometimes, but not always, available in lakehouse environments). In this case, **Ultron is able to learn significantly better (2x) runtime filter placements** for at least two queries (q3, q18). This is attributable to two factors: first, Ultron gets to see the real cardinality of the filter. Second, Ultron can be less conservative about creating a runtime filter because (a) the estimate is more accurate, and (b) if Ultron creates an ineffective filter, that filter will not be created the next time the query is executed (since the ineffectiveness is noted in the query log). This reveals a **surprising operational benefit of Ultron: since a mistake will not be made more than once, it is acceptable to take bigger risks** and potentially find better plans. In both figures, the "No filters" line represents disabling runtime filters entirely. Another possible baseline, enabling every possible runtime filter, significantly increases query latency, making measurement infeasible at this scale, so we do not include it here.

5.2.2 Does Ultron's runtime filter selection converge? A potential pitfall of a history-based learned query optimizer is repeatedly over-correcting a mistake, "flip-flopping" between different query plans. Each Ultron application is carefully designed to avoid exactly this problem (see Section 4), and doing so is trivial when each decision Ultron makes is mostly independent (as is the case for join operator selection). Runtime filters are not independent, however, since adding one runtime filter could make another runtime filter profitable. Here, we show, through a simple example, that Ultron is able to effectively explore the space of runtime filters without regressions, and with eventual convergence to a stable plan.

Figure 9 shows a SQL query that exemplifies the need for multiple runtime filters. Initially, it is not clear that the `c_acctbal` predicate is selective or not, so Databricks will conservatively omit runtime filters. However, this filter is actually selective, so a runtime filter would have been beneficial. Ultron will notice this, and on the second execution of the query, build a Bloom filter on `c_custkey` that is pushed into `orders`. The same situation repeats again at the next level of the query: initially, it does not seem like a filter built over `orders` will be effective, so the conservative choice is to omit it. On the third iteration of the query, Ultron will notice that the filter would actually be effective, and add it.

Figure 10 shows the latency of each iteration. Initially, no filters are constructed. During the second iteration, Ultron introduces a filter built on customer that is pushed into orders, dropping latency by 7%. During the third iteration, Ultron introduces a filter built on orders that is pushed into line item, saving an additional 28% from the second iteration. In the fourth and all future iterations, this plan will be stable, with no new filters added or removed.

5.3 Partition counts

The final Ultron application we evaluate is choosing partition counts for shuffle operators. The Databricks adaptive query executor (AQE) can change the partition size on the fly, splitting up partitions that are too large into smaller chunks. This helps Databricks avoid disk spills and out-of-memory errors. However, this re-partitioning is expensive, since data must be rewritten into smaller chunks. Ideally, we would know the optimal partition count ahead of time, but this requires accurate estimates of cardinality and skew.

Here, we demonstrate that Ultron can effectively remember the ideal partition count discovered by AQE, starting the next iteration of the query. We test our approach on TPC-H at a scale factor of 30,000 (30TB), with a cluster of 56 `rd-fleet.large` instances. We select a subset of the TPC-H queries that are sensitive to partition counts (queries that are not tested had little to no variation when partition count changed). Queries are run twice, to warm the cache and save query history. The second run is recorded.

The results are plotted in Figure 11. Query latencies are normalized to a fixed partition scheme that is optimal for the sum of query latencies (i.e., minimize the workload time). AQE finds the optimal partition count, but at a cost of having to reorganize partitions that are too large or too small, often reaching latencies that are slightly worse than the fixed strategy. Note that AQE is still preferable to a fixed strategy, since AQE avoids disk spills when possible and always avoids out-of-memory errors. Ultron uses the optimal partition count computed by AQE in the previous iteration, **getting the benefits of the perfect partition count without paying the penalty of reorganizing poorly sized partitions.** Ultron is never more than 10% slower than the optimal fixed partition count, and is often significantly faster than AQE (e.g., on q10 and q12).

5.4 Overhead

Ultron adds overhead to the query optimization and execution process; Ultron's calls to QuickPredict, along with hashing parts of the query plan, come at a cost. Overall, **we find that Ultron adds less than 5% overhead to optimization time for the median query**, with a P95 overhead of 12%. Since Ultron accelerates the median eligible query's *runtime* by nearly 25%, this is a worthwhile tradeoff (runtime is generally much longer than optimization time).

We also investigated the "worst-case scenario" of Ultron's overhead by synthetically engineering a workload with no effective

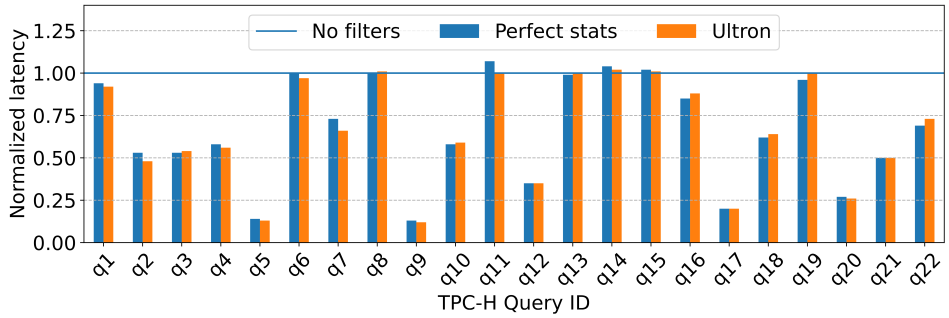

Figure 7: TPC-H query latency, normalized to the latency of a plan with no runtime filters, with runtime filters selected with perfect cardinality estimates and by Ultron. Ultron's selection matches or nearly matches the decisions made by the optimizer with perfect cardinality estimates, indicating that history is a good substitute for perfection, in this case.

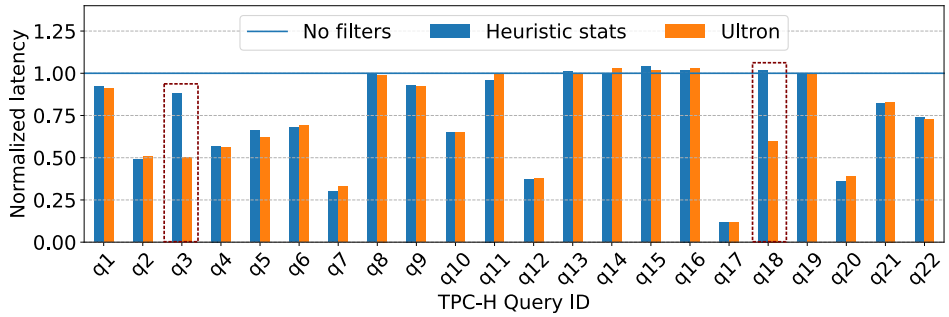

Figure 8: TPC-H query latency, normalized to the latency of a plan with no runtime filters, with filters selected with optimizer estimates and by Ultron. In some cases (dashed maroon boxes), Ultron's use of history leads to a significantly better plan.

```
SELECT
     l_returnflag,
     SUM(l_extendedprice * (1 - l_discount)) AS total_revenue
 FROM (
     -- LEVEL 2: Join Orders result (Build) to Lineitem (Probe)
     -- This creates Bloom Filter for lineitem
     SELECT
         l.l_extendedprice, l.l_discount, l.l_returnflag
     FROM (
         -- LEVEL 1: Join Customer result (Build) to Orders (Probe)
         -- This creates Bloom Filter for orders
         SELECT o.o_orderkey
         FROM (
             SELECT c.c_custkey
             FROM customer c
             WHERE c.c_acctbal > 7000.0
         ) f1_customer
         JOIN orders o ON f1_customer.c_custkey = o.o_custkey
     ) f2_orders
     JOIN lineitem l ON f2_orders.o_orderkey = l.l_orderkey
 ) f3_lineitem
GROUP BY l_returnflag ORDER BY total_revenue DESC;
```

Figure 9: Ultron will progressively add each level of runtime filter to this SQL query.

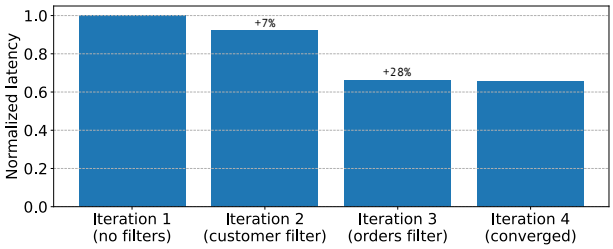

Figure 10: Each iteration of the query in Figure 9 allows Ultron to add another effective runtime filter. On the second iteration, a new filter is built on `customer`, saving 7%. On the third iteration, a new filter is built on `orders`, saving 28%. In the fourth and in all future iterations, the plan is stable.

repetition. We replaced Ultron's hash functions with a simple incrementing value, ensuring that every subplan of every query appears unique. Then, we executed TPC-H for 48 hours, simulating two days of fully distinct queries. We compared the final iteration to a single TPC-H run with all Ultron features (and thus overhead) disabled. Comparing the two runs, we observed that Ultron's overhead led to an 8% slowdown in query optimization time, and a 0.5% slowdown in overall latency – in this case, execution is significantly more expensive than optimization. For very short running queries, the workload overhead could approach the optimization overhead (8%). Additionally, the QuickPredict data stored in Softstore reached 3MB (capped at 50MB in production).

It is critical to ensure that Ultron is able to scale under load. If Ultron relied on a central service, for example, that central service might slow down under load, resulting in a fleet-wide slowdown.

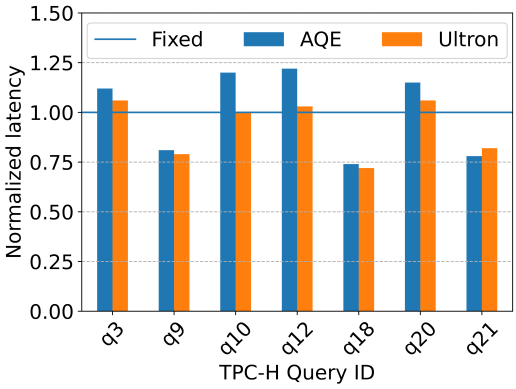

Figure 11: TPC-H queries with different partition count selection methods. Latencies are normalized to the "Fixed" strategy, which uses the fixed partition count that minimizes total latency. "AQE" uses adaptive query processing to split up partitions that are too large (the default strategy prior to Ultron). Ultron sets the partition count equal to the previously observed optimal, gaining the benefits of AQE without the cost.

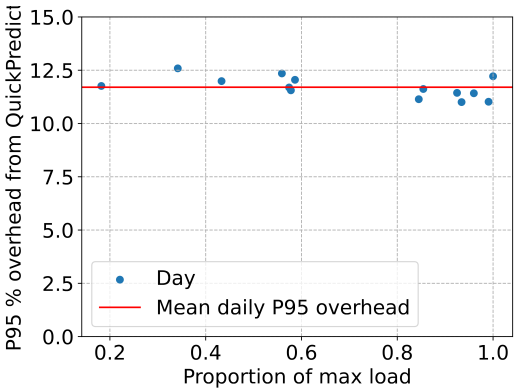

Figure 12: Relationship between load (x-axis) and query compilation overhead due to QuickPredict (y-axis). Each point represents one day. QuickPredict's overhead remaining constant despite changes in load indicates that QuickPredict's architecture scales well to Databricks' requirements.

Fortunately, Ultron's architecture rarely needs to contact a central service. To validate Ultron's scalability, we measured how Ultron's optimization overhead changed as system load changed. Figure 12 shows how system load impacted P95 optimization overhead over a two-week-long period (14 points on the plot). Even at peak load, the P95 optimization overhead added by Ultron never exceeds 12.6%. More importantly, **the overhead seems uncorrelated with load, implying that Ultron is far from being bottlenecked.**

Other operational concerns – like log rotation, history retention, and metadata size from the collected history are also of concern. Due to space constraints, we do not discuss these elements here.

6 RELATED WORK

Query optimization is one of the longest-standing problems in data management [25, 36]. Recent changes to the query optimization "contract" [10] have sought to work around some of the thorniest problems in query optimization, such as cardinality estimation. History-based query optimization – in which the query optimizer examines query logs to gain insight into future queries – is popular in analytics (OLAP) workloads due to their repetitive nature [17, 38]. Query logs have also been used to improve parameterized query optimization [50], robust optimization [22], re-optimization [4, 13, 33], predicate caching [35], latency prediction models [16, 28, 31, 53, 59, 60, 71], and reinforcement learning powered query optimizers [12, 24, 27, 44, 57, 58, 74, 75].

History-based optimization (HBO) is distinct from these other applications because of two features. First, HBO seeks to *explicitly* use history information in the optimization process. For example, an HBO technique might explicitly check query logs for scans over particular tables. In contrast, reinforcement learning powered optimizers only implicitly use query logs (as training data). Second, HBO uses *precise* history data. While latency predictors (for example) use pairs of query plans and query latencies, HBO techniques might store specific facts about a query, like how many rows were read by a particular subplan or if a specific join had to spill to disk. An advantage of precise history usage is that HBO techniques are generally easier to reason about than (for example) fully RL-powered optimizers, since decisions have a specific provenance through query logs. This provenance is also an operational benefit, since it can be used to debug misbehaving systems [38].

Meta's Presto deployment also implements history-based optimization [38]. Differences in our and Meta's use cases drove four key differences in our implementations. First, Meta's system uses a pull-based cache for history data, blocking the planner on a cache miss, but always taking advantage of available history data. Ultron uses a push-based cache, which can avoid first-use cache misses and never blocks the planner, but may plan a query without using history data. Second, while Meta's system has workers periodically send statistics to the coordinator node, Ultron uses a fully asynchronous (and post-execution) history capture technique. This keeps history recording entirely out of the query path, but since the asynchronous write can be delayed, it is possible that history from a query will not be available when optimizing an immediate subsequent query. Third, Meta's system hashes query plans in a way that is invariant to some rewrites, such as join reordering. Ultron, on the other hand, keys history entries to the exact execution plan. Fourth, Meta's implementation is primarily focused on correcting optimizer estimates. Thus, Meta's approach exposes history as a general stats source consumed by existing optimizer rules. On the other hand, Ultron's applications can store and query specific purpose-built statistics for a particular application, creating feedback loops that can improve over multiple iterations.

LLM-powered query optimizers Recent research on query optimization has used large language models (LLMs) to choose between alternative plans [1], directly construct plans [24, 47], and navigating physical configurations [11, 41, 45, 54, 67, 68]. Notably, Zhang et al. [67] and Liu et al. [24] both combine LLM inference with historical query logs, allowing an LLM-powered agent to observe recent

history and act accordingly, creating a stochastic mirror of our deterministic approach. This also reveals a natural synergy between HBO and LLM-powered approaches: an intelligent agent can use data collected by HBO to ground decisions. We leave exploration of this direction to future work.

Offline optimization Offline optimization, or "superoptimization" [26], is another "contract breaker" [10] that strives for good operational characteristics by doing risky exploration offline. For example, a system might use offline execution time to compute specific cardinalities [9], execute alternative query plans [2, 48, 63, 66], or explore the relationship between query parameters and optimal plans [15]. While offline optimization can explore riskier plan changes than HBO (HBO's chosen query plans will be executed in the hot path), offline optimization requires significantly more infrastructure (cloning data, ensuring offline work does not interfere with important queries, and billing questions).

Adaptive query processing Like Ultron, adaptive query processing (AQP) (e.g., [3, 14, 19, 21, 32, 39, 49, 56, 69]) also seeks to improve query plans using information unavailable during optimization. AQP is especially important in lakehouse environments, where data statistics are often sparse [65]. Databricks makes extensive use of adaptive query processing [61], which works in tandem with Ultron. Generally, AQP techniques can improve a query plan after observing some data *at a cost* – for example, Databricks' AQP techniques can swap the build and probe side of a hash join at runtime, but only after partially observing data from both sides. One way to view Ultron is simply as *remembering* what corrections an adaptive query processor applied, so that next time the system can perform the repair without the cost. While not all AQP techniques can be made into Ultron applications (e.g., a optimizing UDFs), we are planning to pursue broader integrations with AQP in the future.

7 REMAINING CHALLENGES

While Ultron has been successfully deployed at Databricks, operating a history-based query optimizer in a large lakehouse environment presents several ongoing challenges. We highlight three key areas for future work: improving debuggability, handling schema evolution, and generalizing historical insights to new workloads.

Debugging customer queries Diagnosing performance regressions in a cloud-scale distributed SQL engine is inherently complex; introducing an asynchronous, history-based optimizer adds another "thing that can go wrong." Currently, when a customer experiences a query regression related to Ultron, the primary mitigation strategy is coarse-grained: disabling Ultron entirely for that query or cluster. While effective as a fallback, more advanced, fine-grained debugging tools are needed. Future tooling could include "provenance tracking" for plans, allowing support engineers to trace exactly which trace triggered an incorrect change in the optimizer's behavior. Additionally, it would be beneficial to support "time-travel" debugging, where the optimizer can simulate planning using the exact state of the history cache at a specific point in the past.

Schema evolution Ultron is designed to be resilient to table size changes (via the size factor, see Section 3.2), but Ultron currently treats schema evolution (e.g., adding or dropping a column) as a hard invalidation event. When a schema change occurs, the historical execution traces for the affected tables are discarded to prevent the optimizer from applying stale or incompatible statistics. While schema evolution is relatively rare compared to data ingestion, discarding the entire history is overly conservative. For example, adding a new column to a table rarely affects the cardinalities or optimal join strategies of queries that do not reference that new column. A significant remaining challenge is developing a mechanism for partial or localized history invalidation.

Generalizing to new queries and customers Currently, Ultron delivers its most significant impact on repetitive queries. While optimizing repetitive queries is valuable to customers (most queries are repetitive), focusing exclusively on repetitive queries risks "overfitting" the system architecture to those queries [29]. Today, Ultron provides limited benefit during the "cold start" phase of a new workload (e.g., on-boarding). To optimize these zero-history scenarios, one must generalize from historical traces to structurally similar, but strictly novel, queries, which introduces significant risks. Generalization error could cause severe performance regressions. Furthermore, in a multi-tenant cloud environment, generalizing across different customers requires stringent security and privacy constraints. Any cross-tenant learning mechanism must guarantee that query plan shapes or statistical insights from one customer's workload cannot inadvertently leak sensitive data to another tenant. Overcoming these hurdles remains an active area of research.

More opportunities for history This work presented three Ultron applications, but many more are in development at Databricks. Potential future applications might look at how often a particular subplan occurs within a query plan or workload, or might consider which rules have already fired on a particular subplan. Rules that remember properties of deterministic UDFs may also be considered.

Scalability & overhead Ultron's benefits depends on the improvements to query runtime outweighing the added optimization cost. While our current applications achieve this for our customers, we are closely monitoring the overhead induced by Ultron, as adding additional applications might increase overhead non-linearly. Currently, maintaining a good tradeoff between optimization overhead and performance improvement is the responsibility of the engineer implementing an Ultron application. We leave studying how to automatically manage this tradeoff – possibly by disabling certain applications for particular workloads or customers – to future work. One specific difficulty to automatically managing Ultron applications is that Ultron applications can use history in significantly different ways: some Ultron applications may only activate when certain query patterns appear, while others might look at every subplan generated by the optimizer.

8 CONCLUSION

This work presented Ultron, a history-based query optimization framework designed for Databricks. Ultron handles user clusters that frequently grow, stop, or start, while still keeping optimization decisions fast and local. We showcase three applications of Ultron – selecting join operators, selecting runtime filters, and selecting partition counts – that ensure convergence and rarely cause regressions. In future work, we hope to develop more Ultron applications, as well as considering offline execution to test riskier optimizations.

REFERENCES

[1] Peter Akioyamen, Zixuan Yi, and Ryan Marcus. 2024. The Unreasonable Effectiveness of LLMs for Query Optimization *(MLForSystems @ NeurIPS '24)*. Vancouver, BC, Canada. https://doi.org/10.48550/arXiv.2411.02862 arXiv:2411.02862 [cs].

[2] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithvi Pandian, Nikolay Leptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *PVLDB* 14, 1 (Aug. 2023). https://doi.org/10.14778/3611540.3611544

[3] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, New York, NY, USA, 261–272. https://doi.org/10.1145/342009.335420

[4] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization with Rio. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 936–938. https://doi.org/10.1145/1066157.1066294

[5] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom Van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. ACM, Philadelphia, PA, USA, 2326–2339. https://doi.org/10.1145/3514221.3526054

[6] Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. 2025. Instance-Optimal Acyclic Join Processing Without Regret: Engineering the Yannakakis Algorithm in Column Stores. https://doi.org/10.48550/arXiv.2411.04042 arXiv:2411.04042 [cs].

[7] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (July 2024), 3215–3228. https://doi.org/10.14778/3681954.3681995

[8] Tracy Boggiano and Grant Fritchey. 2019. What Is Query Store? In *Query Store for SQL Server 2019: Identify and Fix Poorly Performing Queries*, Tracy Boggiano and Grant Fritchey (Eds.). Apress, Berkeley, CA, 1–29. https://doi.org/10.1007/978-1-4842-5004-4_1

[9] Sunil Chakkappen, Suratna Budalakoti, Ramarajan Krishnamachari, Satyanarayana R Valluri, Alan Wood, and Mohamed Zait. 2017. Adaptive statistics in Oracle 12c. *Proceedings of the VLDB Endowment* 10, 12 (Aug. 2017), 1813–1824. https://doi.org/10.14778/3137765.3137785

[10] Surajit Chaudhuri. 2009. Query optimizers: time to rethink the contract?. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 961–968. https://doi.org/10.1145/1559845.1559955

[11] Sibei Chen, Ju Fan, Bin Wu, Nan Tang, Chao Deng, Pengyi Wang, Ye Li, Jian Tan, Feifei Li, Jingren Zhou, and Xiaoyong Du. 2025. Automatic Database Configuration Debugging using Retrieval-Augmented Language Models. *Proc. ACM Manag. Data* 3, 1 (Feb. 2025), 13:1–13:27. https://doi.org/10.1145/3709663

[12] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer Towards Generating Efficient and Robust Query Execution Plans. *Proceedings of the VLDB Endowment* 16, 7 (March 2023), 1777–1789. https://doi.org/10.14778/3587136.3587150

[13] Guilherme Damasio, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Alex Mihaylov, Jaroslaw Szlichta, and Calisto Zuzarte. 2019. Guided automated learning for query workload re-optimization. *Proceedings of the VLDB Endowment* 12, 12 (Aug. 2019), 2010–2021. https://doi.org/10.14778/3352063.3352120 Offline workload extraction phase.

[14] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. Adaptive query processing. *Found. Trends databases* 1, 1 (Jan. 2007), 1–140. https://doi.org/10.5555/1331939.1331940

[15] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbuken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: Robust Learning for Faster Parametric Query Optimization. *Proceedings of the 2023 ACM SIGMOD Conference* 1, 1 (May 2023), 109. https://doi.org/10.1145/3588963

[16] Jennie Duggan, Olga Papaemmanouil, Ugur Cetintemel, and Eli Upfal. 2014. Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT '14)*. 109–120.

[17] Google. 2026. Use history-based optimizations | BigQuery. https://docs.cloud.google.com/bigquery/docs/history-based-optimizations

[18] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. 1997. Parametric query optimization. *The VLDB Journal* 6, 2 (May 1997), 132–151. https://doi.org/10.1007/s007780050037

[19] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. 2004. Adapting to source properties in processing data integration queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD '04)*. ACM, Paris France, 395–406. https://doi.org/10.1145/1007568.1007613

[20] Zachary G. Ives and Nicholas E. Taylor. 2008. Sideways Information Passing for Push-Style Query Processing. In *2008 IEEE 24th International Conference on Data Engineering (ICDE '08)*. 774–783. https://doi.org/10.1109/ICDE.2008.4497486

[21] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. 2018. Cuttlefish: A Lightweight Primitive for Adaptive Query Processing. *arXiv preprint* (Feb. 2018). https://arxiv.org/abs/1802.09180

[22] Amin Kamali, Verena Kantere, Calisto Zuzarte, and Vincent Corvinelli. 2024. RobOpt: A Tool for Robust Workload Optimization Based on Uncertainty-Aware Machine Learning. In *Companion of the 2024 International Conference on Management of Data (SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 468–471. https://doi.org/10.1145/3626246.3654755

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

[24] Hanwen Liu, Qihan Zhang, Ryan Marcus, and Ibrahim Sabek. 2026. SEFRQO: A Self-Evolving Fine-Tuned RAG-Based Query Optimizer. *Proceedings of the ACM on Management of Data* 3, 6 (May 2026). https://doi.org/10.1145/3769826

[25] Guy Lohman. 2014. Is Query Optimization a "Solved" Problem?. In *ACM SIGMOD Blog (ACM Blog '14)*. https://wp.sigmod.org/?p=1075

[26] Ryan Marcus. 2023. Learned Query Superoptimization. In *Joint Workshops at 49th International Conference on Very Large Data Bases (AIDB@VLDB '23)*. CEUR Workshop Proceedings, Vancouver, BC, Canada.

[27] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. China. https://doi.org/10.1145/3448016.3452838 Award: 'best paper award'.

[28] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *PVLDB* 12, 11 (2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[29] Ryan Marcus, Jeffrey Tao, Peizhi Wu, and Zijie Zhao. 2026. Survivorship Bias in Industrial Database Workloads *(CIDR '26)*. California, USA. Award: 'best paper award'.

[30] V. Markl, P. J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T. M. Tran. 2007. Consistent selectivity estimation via maximum entropy. *The VLDB Journal* 16, 1 (Jan. 2007), 55–76. https://doi.org/10.1007/s00778-006-0030-1

[31] Mert Akdere and Ugur Cetintemel. 2012. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering (ICDE '12)*. IEEE, 390–401.

[32] Chris Olston, Jing Jiang, and Jennifer Widom. 2003. Adaptive filters for continuous queries over distributed data streams. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 563–574. https://doi.org/10.1145/872757.872825

[33] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 2019. How I Learned to Stop Worrying and Love Re-optimization. *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (April 2019), 1758–1761. https://doi.org/10.1109/ICDE.2019.00191 Conference Name: 2019 IEEE 35th International Conference on Data Engineering (ICDE) ISBN: 9781538674741 Place: Macao, Macao.

[34] Meikel Poess and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *SIGMOD Records* 29, 4 (Dec. 2000), 64–71. https://doi.org/10.1145/369275.369291

[35] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2024. Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses. In *Companion of the 2024 International Conference on Management of Data (SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 347–359. https://doi.org/10.1145/3626246.3653395

[36] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD '79 (SIGMOD '79)*, John Mylopolous and Michael Brodie (Eds.). Morgan Kaufmann, San Francisco (CA), 511–522. https://doi.org/10.1016/B978-0-934613-53-8.50038-8

[37] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104, 1 (Jan. 2016), 148–175. https://doi.org/10.1109/JPROC.2015.2494218

[38] Pranjal Shankhdhar, Feilong Liu, Jay Narale, James Sun, Rebecca Schlussel, and Lyublena Antova. 2024. Presto's History-Based Query Optimizer. *Proceedings of the VLDB Endowment* 17, 12 (Aug. 2024), 4077–4089. https://doi.org/10.14778/3685800.3685828

[39] Kyumars Sheykh Esmaili, Tahmineh Sanamrad, Peter M. Fischer, and Nesime Tatbul. 2011. Changing flights in mid-air: a model for safely modifying continuous queries. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 613–624. https://doi.org/10.1145/1989323.1989388

[40] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. 2013. Materialization strategies in the Vertica analytic database: Lessons learned. In *2013 IEEE 29th International Conference on Data Engineering (ICDE) (ICDE '13)*. 1196–1207. https://doi.org/10.1109/ICDE.2013.6544909

[41] Vikramank Singh, Kapil Eknath Vaidya, Vinayshekhar Bannihatti, Sopan Khosla, Balakrishnan Narayanaswamy, Rashmi Gangadharaiah, and Tim Kraska. 2024. Panda: Performance Debugging for Databases using LLM Agents. In *Conference on Innovative Data Systems Research (CIDR '24)*.

[42] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *VLDB (VLDB '01)*. 19–28. http://dl.acm.org/citation.cfm?id=645927.672349

[43] Mihail Stoian, Andreas Zimmerer, Skander Krid, Amadou Latyr Ngom, Jialin Ding, Tim Kraska, and Andreas Kipf. 2025. Parachute: Single-Pass Bi-Directional Information Passing. https://doi.org/10.48550/arXiv.2506.13670 arXiv:2506.13670 [cs].

[44] Pavel Sulimov, Claude Lehmann, and Kurt Stockinger. 2024. GenJoin: Conditional Generative Plan-to-Plan Query Optimizer that Learns from Subplan Hints. https://doi.org/10.48550/arXiv.2411.04525 arXiv:2411.04525 [cs].

[45] Zhaoyan Sun, Xuanhe Zhou, Jianming Wu, Wei Zhou, and Guoliang Li. 2025. D-Bot: An LLM-Powered DBA Copilot. In *Companion of the 2025 International Conference on Management of Data (SIGMOD '25)*. Association for Computing Machinery, New York, NY, USA, 235–238. https://doi.org/10.1145/3722212.3725091

[46] Richard S. Sutton and Andrew G. Barto. 1998. *Introduction to Reinforcement Learning* (1st ed.). MIT Press, Cambridge, MA, USA.

[47] Jie Tan, Kangfei Zhao, Rui Li, Jeffrey Xu Yu, Chengzhi Piao, Hong Cheng, Helen Meng, Deli Zhao, and Yu Rong. 2025. Can Large Language Models Be Query Optimizer for Relational Databases? *Proc. ACM Manag. Data* 3, 6 (Dec. 2025), 306:1–306:28. https://doi.org/10.1145/3769771

[48] Jeffrey Tao, Natalie Maus, Haydn Jones, Yimeng Zeng, Jacob R. Gardner, and Ryan Marcus. 2025. Learned Offline Query Planning via Bayesian Optimization. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 179. https://doi.org/10.1145/3725316

[49] Immanuel Trummer, Samuel Moseley, Deepak Maram, Saehan Jo, and Joseph Antonakakis. 2018. SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning. *PVLDB* 11, 12 (2018), 2074–2077. https://doi.org/10.14778/3229863.3236263

[50] Kapil Vaidya, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. 2022. Leveraging query logs and machine learning for parametric query optimization. *Proceedings of the VLDB Endowment* 15, 3 (Feb. 2022), 401–413. https://doi.org/10.14778/3494124.3494126

[51] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. *Proceedings of the VLDB Endowment* 17, 11 (July 2024), 3694–3706. https://doi.org/10.14778/3681954.3682031

[52] Alexander Van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proceedings of the VLDB Endowment* 16, 6 (Feb. 2023), 1413–1425. https://doi.org/10.14778/3583140.3583156

[53] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*. 363–378.

[54] Pengyi Wang, Sibei Chen, Ju Fan, Bin Wu, Nan Tang, and Jian Tan. 2025. Andromeda: Debugging Database Performance Issues with Retrieval-Augmented Large Language Models. In *Companion of the 2025 International Conference on Management of Data (SIGMOD '25)*. Association for Computing Machinery, New York, NY, USA, 243–246. https://doi.org/10.1145/3722212.3725080

[55] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees. *Proc. ACM Manag. Data* 3, 3 (June 2025), 235:1–235:28. https://doi.org/10.1145/3725423

[56] Ziyun Wei and Immanuel Trummer. 2024. ROME: Robust Query Optimization via Parallel Multi-Plan Execution. *Proceedings of the ACM on Management of Data* 2, 3 (May 2024), 170:1–170:25. https://doi.org/10.1145/3654973

[57] Lianggui Weng, Rong Zhu, Di Wu, Bolin Ding, Bolong Zheng, and Jingren Zhou. 2024. Eraser: Eliminating Performance Regression on Learned Query Optimizer. *PVLDB* 17, 5 (2024), 926–938. https://doi.org/10.14778/3641204.3641205

[58] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2023. FASTgres: Making Learned Query Optimizer Hinting Effective. *Proceedings of the VLDB Endowment* 16, 11 (Aug. 2023), 3310–3322. https://doi.org/10.14778/3611479.3611528

[59] Wentao Wu, Hakan Hacigumus, Yun Chi, Shenghuo Zhu, Junichi Tatemura, and Jeffrey F. Naughton. 2013. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable?. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13)*. IEEE Computer Society, Washington, DC, USA, 1081–1092. https://doi.org/10.1109/ICDE.2013.6544899

[60] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *Proceedings of the 2024 International Conference on Management of Data (SIGMOD '24) (SIGMOD '24)*. Santiago, Chile. https://doi.org/10.48550/arXiv.2403.02286

[61] Maryann Xue, Yingyi Bu, Abhishek Somani, Wenchen Fan, Ziqi Liu, Steven Chen, Herman Van Hovell, Bart Samwel, Mostafa Mokhtar, Rk Korlapati, Andy Lam, Yunxiao Ma, Vuk Ercegovac, Jiexing Li, Alexander Behm, Yuanjian Li, Xiao Li, Sriram Krishnamurthy, Amit Shukla, Michalis Petropoulos, Sameer Paranjpye, Reynold Xin, and Matei Zaharia. 2024. Adaptive and Robust Query Execution for Lakehouses at Scale. *Proceedings of the VLDB Endowment* 17, 12 (Aug. 2024), 3947–3959. https://doi.org/10.14778/3685800.3685818

[62] M. Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases Conference (VLDB '81)*. 82–94. https://doi.org/10.5555/1286831.1286840

[63] Zixuan Yi, Yao Tian, Zachary G. Ives, and Ryan Marcus. 2025. Low Rank Learning for Offline Query Optimization. *Proceedings of the ACM on Management of Data* 3, 3 (2025), 183. https://doi.org/10.1145/3725412

[64] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing (HotCloud'10)*. USENIX Association, USA, 10.

[65] M. Zaharia, A. Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics *(CIDR '21)*. Virtual Event. https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[66] Wangda Zhang, Matteo Interlandi, Paul Mineiro, Shi Qiao, Nasim Ghazanfari, Karlen Lie, Marc Friedman, Rafah Hosn, Hiren Patel, and Alekh Jindal. 2022. Deploying a Steered Query Optimizer in Production at Microsoft. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. ACM, Philadelphia PA USA, 2299–2311. https://doi.org/10.1145/3514221.3526052

[67] William Zhang, Wan Shen Lim, and Andrew Pavlo. 2026. This is Going to Sound Crazy, But What If We Used Large Language Models to Boost Automatic Database Tuning Algorithms By Leveraging Prior History? We Will Find Better Configurations More Quickly Than Retraining From Scratch! *Proc. ACM Manag. Data* 4, 1 (April 2026), 90:1–90:29. https://doi.org/10.1145/3786704

[68] Xinyi Zhang, Tiantian Chen, Zhentao Han, Zhaoyan Hong, Wei Lu, Sheng Wang, Mo Sha, Anni Wang, Shuang Liu, Yakun Zhang, Feifei Li, and Xiaoyong Du. 2026. Why Database Manuals are Not Enough: Efficient and Reliable Configuration Tuning for DBMSs via Code-Driven LLM Agents. *Proc. VLDB Endow.* 19, 6 (May 2026), 1358–1371. https://doi.org/10.14778/3797919.3797940

[69] Yunjia Zhang, Yannis Chronis, Jignesh M. Patel, and Theodoros Rekatsinas. 2023. Simple Adaptive Query Processing vs. Learned Query Optimizers: Observations and Analysis. *Proceedings of the VLDB Endowment* 16, 11 (July 2023), 2962–2975. https://doi.org/10.14778/3611479.3611501

[70] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. *Proc. ACM Manag. Data* 3, 3 (June 2025), 146:1–146:28. https://doi.org/10.1145/3725283

[71] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: a tree transformer model for query plan representation. *Proceedings of the VLDB Endowment* 15, 8 (April 2022), 1658–1670. https://doi.org/10.14778/3529337.3529349

[72] Jingren Zhou, Nicolas Bruno, and Wei Lin. 2012. Advanced partitioning techniques for massively distributed computation. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, Scottsdale Arizona USA, 13–24. https://doi.org/10.1145/2213836.2213839

[73] Li Zhou. 2016. A Survey on Contextual Multi-armed Bandits. http://arxiv.org/abs/1508.03326

[74] Rong Zhu, Lianggui Weng, Wenqing Wei, Di Wu, Jiazhen Peng, Yifan Wang, Bolin Ding, Defu Lian, Bolong Zheng, and Jingren Zhou. 2024. PilotScope: Steering Databases with Machine Learning Drivers. *PVLDB* 17, 5 (2024), 980–993. https://doi.org/10.14778/3641204.3641209

[75] Sergey Zinchenko and Sergey Iazov. 2024. HERO: Hint-Based Efficient and Reliable Query Optimizer. https://doi.org/10.48550/arXiv.2412.02372 arXiv:2412.02372 [cs].

[76] Joshua P Zitovsky. 2023. Revisiting Bellman Errors for Offline Model Selection. In *Proceedings of the 40th International Conference on Machine Learning (ICML '23)*. Hawaii, USA.