AQD: Online Adaptive Query Dispatcher for HTAP Databases

Yang Wu
Department of Computer Science,
Tsinghua University
Beijing, China
wu-y22@mails.tsinghua.edu.cn

Tongliang Li
Alibaba Group
Hangzhou, China
litongliang.ltl@alibaba-inc.com

Xuanhe Zhou
Department of Computer Science,
Shanghai Jiao Tong University
Shanghai, China
zhouxh@cs.sjtu.edu.cn

Jianying Wang
Alibaba Group
Hangzhou, China
beilou.wjy@alibaba-inc.com

Xinjun Yang
Alibaba Group
Hangzhou, China
xinjun.y@alibaba-inc.com

Wenchao Zhou
Alibaba Group
Hangzhou, China
zwc231487@alibaba-inc.com

Chunxiao Xing
Beijing National Research Center for
Information Science and Technology,
Tsinghua University
Beijing, China
xingcx@tsinghua.edu.cn

Yong Zhang
Beijing National Research Center for
Information Science and Technology,
Tsinghua University
Beijing, China
zhangyong05@tsinghua.edu.cn

ABSTRACT

Hybrid Transactional-Analytical Processing (HTAP) has attracted growing attention from both academia and industry. Most HTAP systems adopt a dual-engine architecture, maintaining separate row and column engines to achieve workload isolation: row engines excel at transactional workloads, while column engines are optimized for analytical queries. For such systems, dispatching queries to the appropriate engine with ultra-low latency is highly desirable but remains challenging. Existing approaches often rely on traditional cost estimation, which is often inaccurate and fails to adapt to dynamic workload patterns. Moreover, they generally overlook resource balancing when dispatching workloads.

In this paper, we present AQD, an online Adaptive Query Dispatcher framework. AQD operates in two phases: (1) in the offline phase, it trains a LightGBM classifier using self-paced, Taylor-weighted boosting that emphasizes costly mispredictions; (2) in the online phase, it employs a LinTS-Delta bandit to adapt to workload drift via execution feedback, while a Mahalanobis-based regulator ensures balanced CPU and memory utilization across the two engines. We integrate AQD into PolarDB and evaluate it on standard benchmarks as well as real-world datasets. Experimental results show that AQD reduces average query latency by up to 42% compared to cost-threshold dispatching, achieving up to 87% of the optimal performance. Under concurrent query execution, it lowers latency by more than 40% while maintaining efficient and stable resource utilization with minimal dispatching overhead.

PVLDB Reference Format:
Yang Wu, Tongliang Li, Xuanhe Zhou, Jianying Wang, Xinjun Yang, Wenchao Zhou, Chunxiao Xing, and Yong Zhang. AQD: Online Adaptive Query Dispatcher for HTAP Databases. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, and the full version of the paper (including appendices) have been made available at https://github.com/earthwuyang/aqd.

1 INTRODUCTION

Hybrid Transactional–Analytical Processing (HTAP) has emerged as both a critical research area in academia and a widely adopted solution in industry [10, 12, 16, 24, 47, 51]. HTAP engines enable a single cluster to ingest high-volume OLTP updates while simultaneously answering OLAP queries over the same, freshly committed data, providing users with fast, efficient and transparent access to unified transactional and analytical capabilities [16, 51].

HTAP systems can be broadly classified into two categories: *single-engine* and *dual-engine* architectures. *Single-engine* systems like Oracle In-Memory [23], SQL Server Columnstore [24], SAP HANA [12], HyPer [20], and StarRocks [45] use one shared set of optimizer and execution engine for HTAP workloads, providing ultra-fresh data access. However, workload interference and resource contention can lead to performance degradation [44].

In contrast, *dual-engine* systems have gained widespread adoption in production environments, including TiDB paired with TiFlash [16], PolarDB [47], MySQL HeatWave [6], and PostgreSQL paired with DuckDB [7, 36]. These systems maintain separate row-oriented and column-oriented engines. The row engine excels at point queries and tuple-at-a-time processing using row storage, while the column engine optimizes for analytical scans through vectorized execution, columnar storage, and OLAP-specific optimizations. This architectural separation allows each engine to be highly optimized for its target workload, potentially achieving better performance than a one-size-fits-all approach.

Table 1: Query Characteristics and SQL Examples for Row-Store vs Column-Store Engines

	Row-Store (OLTP)	Column-Store (OLAP)
Optimal Query Characteristics	• Point queries • Full row retrieval • High selectivity (<5% rows) • Index range scans • Small result sets	• Aggregations • GROUP BY operations • Large table scans • Low selectivity (>20% rows) • Few column projections
Example SQL Queries	· `SELECT * FROM users WHERE user_id = 12345;` · `UPDATE orders SET status='completed' WHERE id = 789;` · `INSERT INTO logs (user_id, action) VALUES (123, 'login');`	· `SELECT region, SUM(sales) FROM orders GROUP BY region;` · `SELECT AVG(salary) FROM employees WHERE dept = 'Sales';` · `SELECT COUNT(*) FROM transactions WHERE date > '2024-01';`

The main challenge of dual-engine systems lies in intelligently dispatching queries to the appropriate engine to maximize overall system performance and make the query dispatch transparent to users. We term this the *query dispatch* problem of HTAP databases. As shown in Table 1, row-store engine and column-store engine have their own advantages and are suitable for different workloads [1, 43]. In production deployments, incorrect row/column dispatch decisions have become a primary operational pain point. Internal statistics from PolarDB, a major cloud HTAP deployment, reveal that approximately 20% of claims from customers stem from suboptimal dispatch decisions, making it the top issue affecting system performance and user experience.

Current approaches to query dispatch rely primarily on analytical cost models [10, 24, 47] or hand-tuned heuristics [12]. Unfortunately, cost estimations are often inaccurate and rule lists are partial, inevitably leading to suboptimal dispatch decisions. For example, PolarDB uses a simple cost-threshold rule, where *queries with estimated costs exceeding a fixed threshold execute on the column engine, while others run on the row engine*[1] [4]. However, this threshold-based approach frequently misclassifies queries, i.e., dispatching high-cost queries that would run faster on row storage to columnar, and vice versa. Moreover, real HTAP workloads exhibit dynamic patterns and the optimal engine choice depends on the evolving TP-to-AP load ratio, but current cost-based and heuristic methods fail to adapt to workload drift.

Inspired by recent advances in learned database optimization [30, 31, 48, 53], we extend this line of work to HTAP query dispatch. However, ML-based dispatch still faces several key challenges:

- **C1. Balancing CPU and memory across engines.** The dispatcher must balance processor load and memory consumption between engines, which is a dynamic property that static features cannot adequately capture.
- **C2. Microsecond-level latency constraints.** Dispatch decisions lie on the query compilation critical path, allowing microsecond-level inference latency.
- **C3. Different misprediction costs.** Misdispatching a long-running query wastes significantly more resources than misdispatching a short one, yet standard binary classification treats both errors equally.
- **C4. Noise in runtime measurement.** Execution times fluctuate due to system load and resource contention, requiring robustness against noise in training data.

To address these challenges, we propose AQD, an online Adaptive Query Dispatcher framework for HTAP query dispatch. The framework integrates three components: (1) a LightGBM classifier [19] trained on historical workloads for baseline predictions, (2) a LinTS-Delta bandit [3] that computes performance residuals against the baseline to adapt to workload distribution shifts, and (3) resource monitoring via Mahalanobis distance [32] scores to detect CPU/memory imbalances between engines. These scores are weighted based on system load estimated [17, 38]: under high load, latency minimization dominates due to queuing amplification[2], while resource balance takes precedence under low load. Queries are dispatched to the column engine when the combined score is positive. We validate our approach by implementing AQD in the PolarDB [47] kernel, demonstrating both feasibility and performance gains.

Contributions. We summarize our key contributions as follows:

(1) We propose and implement AQD, to the best of our knowledge the first learning-based HTAP query dispatch framework that integrates offline-trained LightGBM classifiers with online LinTS-Delta residual learning and adaptive Mahalanobis distance-based resource regulation.
(2) We propose a dispatch-specific feature engineering pipeline that can extract 142 raw features, and reduce the number to 32 via SHAP analysis [28, 29]. A novel self-paced Taylor-weighted boosting with six weight factors focuses learning on costly mispredictions while handling outliers (**C3**, **C4**).
(3) We formulate online query dispatch as contextual bandit problem and use LinTS-Delta for residual learning and Online Convex Optimization (OCO) for resource regulation. The framework dynamically combines performance and resource scores, achieving proved regret bounds for both latency regret and resource deviation (**C1**, **C2**).
(4) Experiments on PolarDB show individual query dispatch (no resource contention) reduces latency by up to **42%**. In end-to-end test, under concurrent query execution, latency drops over **40%** versus the baseline while maintaining better and stable resource utilization.

Outline. Section 2 introduces problem formulations of *Query-Level Dispatch* and *Workload-Level Dispatch*, and the framework of AQD (offline preparation and online dispatch). Section 3 details the techniques in our *Query-Level Dispatch* for individual query execution. Section 4 details the techniques in our *Workload-Level Dispatch* for concurrent query execution. Section 5 reports benchmark and production results. Section 6 analyses insights, limitations, and future work; Section 7 summarizes related work. Section 8 concludes the paper.

[1]Column engine queries can access row store data through row-column fusion operators.

[2]Even 1ms service-time errors amplify to 10ms tail latency at high utilization ($T \approx 1/(1-\rho)$ for M/M/1 queues).

2 AQD OVERVIEW

In this section, we first propose problem formulations for *Query-Level Dispatch* and *Workload-Level Dispatch*, and then overview the framework of AQD.

2.1 Problem Formulations

We formulate the *query dispatch problem* at two levels: *query-level dispatch* where queries are executed individually, and *workload-level dispatch* where queries are execute concurrently with resource constraints and system dynamics taken into account.

2.1.1 *Query-Level Dispatch.*

We begin with *query-level dispatch* without considering resource contention or system state, essentially a binary classification problem in the *offline preparation* phase.

Definition of Query-Level Dispatch Problem. Given a query q, represented by a feature vector $\mathbf{x} \in \mathbb{R}^d$ extracted directly from the query optimizer's internal statistics, our goal is to accurately predict which of the two engines (row or column) can execute the query more efficiently. Let $\ell_{\text{row}}(q)$ and $\ell_{\text{col}}(q)$ denote the execution latencies on the row and column engines respectively. The *query-level dispatch problem* is to learn a function: $f : \mathbf{x} \longrightarrow a^* \in \{0, 1\}$ where $a^* = \arg\min_{a \in \{0,1\}} \ell_a(q)$, with $a = 0$ selecting the row engine and $a = 1$ the column engine.

For each query q, we define the latency gap $w(q) = |\ell_{\text{row}}(q) - \ell_{\text{col}}(q)|$ measuring performance difference between engines. Let $z = \mathbf{1}[\ell_{\text{col}} < \ell_{\text{row}}]$ indicate the optimal engine (1 for column, 0 for row). Incorrect dispatches incur regret $r(q, a) = w(q) \cdot \mathbf{1}[a \neq z]$, making mistakes on queries with large performance gaps more costly.

2.1.2 *Workload-Level Dispatch.*

In production environments, dispatch decisions must adapt to dynamic workloads and maintain resource balance between engines. We formulate this as a constrained online optimization problem for the *online dispatch* phase.

Definition of Workload-Level Dispatch Problem. Index the incoming queries by $i = 1, \ldots, T$. For each query we choose a binary action $a_i \in \{0, 1\}$, where $a_i = 0$ dispatches it to the row engine and $a_i = 1$ to the column engine. Let $\ell_i(a_i)$ denote the latency observed for query i under that decision, and let $\mathbf{r}_i = (\rho_i^{\text{cpu}}, \rho_i^{\text{mem}}) \in [0, 1]^2$ be the CPU/memory share of the row engine sampled at the same instant. Given a slowly varying target $\boldsymbol{\gamma}_T = (\gamma^{\text{cpu}}, \gamma^{\text{mem}}) \in (0, 1)^2$, the *workload-level dispatch problem* is to minimize the total latency sum under the resource constraint at workload-level:

$$\begin{aligned} \min_{\{a_i\}_{i=1}^{T}} \quad & \sum_{i=1}^{T} \ell_i(a_i) \\ \text{s.t.} \quad & \frac{1}{T}\sum_{i=1}^{T} \mathbf{r}_i - \frac{1}{T}\sum_{i=1}^{T} \mathbb{1}_{\{a_i=0\}} \cdot \mathbf{1} = \boldsymbol{\gamma}_T \end{aligned} \tag{1}$$

where $\mathbf{1} = (1, 1)^\top$ is the all-ones vector. This constraint ensures that the average resource consumption bias of the row engine matches the target profile $\boldsymbol{\gamma}_T$.

Constraint Interpretation. To understand the resource balance constraint, define: $\bar{\mathbf{r}} = \frac{1}{T}\sum_{i=1}^{T} \mathbf{r}_i, \quad \bar{a} = \frac{1}{T}\sum_{i=1}^{T} \mathbb{1}_{\{a_i=0\}}$ where $\bar{\mathbf{r}}$ represents the average CPU/memory share consumed by the row engine, and $\bar{a}$ represents the fraction of queries routed to it. The difference $\bar{\mathbf{r}} - \bar{a} \cdot \mathbf{1}$ quantifies resource bias:

- Positive values indicate the row engine consumes more resources than its traffic share.
- Negative values indicate underutilization relative to query volume.
- Zero represents perfect proportionality between resource usage and query distribution.

Constraint (1) forces this bias to equal the target $\boldsymbol{\gamma}_T$. Common settings include:

- $\boldsymbol{\gamma}_T = (0, 0)$: balanced utilization between engines.
- $\boldsymbol{\gamma}_T$ slightly positive: reserve row engine capacity for OLTP bursts.
- $\boldsymbol{\gamma}_T$ slightly negative: bias toward column engine for AP-heavy workloads.

Complexity Analysis. The Workload-Level Dispatch Problem is NP-hard even in the offline setting with complete information, as we prove in Appendix A[3]. The online version makes this problem even harder because we must make decisions immediately without knowing what queries will come next, and we cannot change these decisions later.

2.2 Overall Framework of AQD

Next, we present the overall framework of AQD in Figure 1, which consists of two phases: *offline preparation* and *online dispatch.*

2.2.1 *Offline Preparation Phase.*

The offline phase builds a foundational model that predicts the optimal engine based on query features alone, without considering runtime resource contention. This phase comprises three stages:

(1) **Feature engineering (Section 3.1)**: We instrument the optimizer to expose 142 internal features from the `JOIN` structure, including join shape, cardinalities, and predicate types, etc. Then we apply SHAP analysis [28, 29] to reduce the feature number to 32.
(2) **Data collection (Section 3.2)**: We construct a comprehensive training dataset by executing queries from fifteen diverse workloads—including TPC-H, TPC-DS, HyBench, and production traces—on both row and column engines. Each query execution yields a labeled record: the 32-dimensional feature vector paired with ground-truth latencies from both engines.
(3) **Model training (Section 3.3)**: After evaluating decision trees, random forests, feed-forward neural networks, and LightGBM, we select LightGBM for its superior prediction accuracy, fast training and inference speed. We train a regret-weighted LightGBM model [19] using self-paced Taylor-weighted boosting.

2.2.2 *Online Dispatch Phase.*

While offline learning provides a strong baseline, production systems face dynamic workload patterns and resource constraints. The online phase addresses these challenges through three complementary stages:

(1) **LightGBM predicting (Section 4.1)**: (i) When a query arrives, the optimizer generates query plan. (ii) The system extracts 32 features from the optimizer's JOIN structure, including cost estimates, cardinality predictions, selectivity factors, and query shape characteristics. (iii) The LightGBM model processes these

[3]In the full version of this paper.

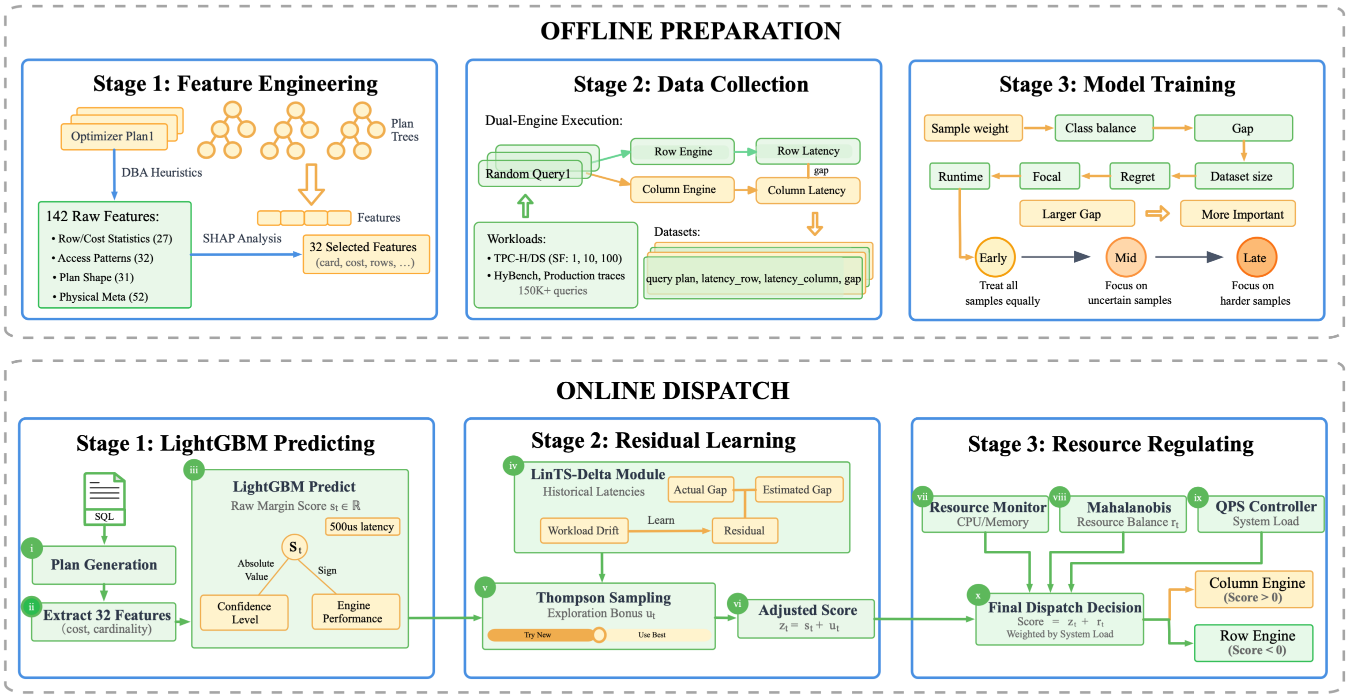

Figure 1: Framework of AQD

features and outputs a raw margin score indicating the column engine's expected benefit. This score serves as the base prediction, with its magnitude encoding confidence and sign suggesting the preferred engine.

(2) **Residual learning (Section 4.2)**: (iv) The LinTS-Delta module maintains running averages of historical latencies for both engines as counterfactual estimates. (v) Thompson sampling generates an exploration bonus based on the posterior distribution of prediction errors. (vi) The module combines the base LightGBM score with the exploration bonus to produce an adjusted dispatch score that balances exploitation of known patterns with exploration of uncertain cases.

(3) **Resource regulating (Section 4.3)**: (vii) The resource monitor tracks CPU and memory utilization of both engines. (viii) A Mahalanobis distance metric quantifies how far the current resource distribution deviates from the target balance. (ix) The QPS controller estimates system load and dynamically adjusts the relative importance of performance versus resource balance. (x) The final dispatch decision combines the performance-oriented score from Stage 2 with the resource balance score, weighted by current system load. Queries are dispatched to the column engine when the combined score is positive, otherwise to the row engine.

3 OFFLINE PREPARATION

In this section, we introduce the offline preparation phase of AQD, including *feature engineering*, *data collection*, and *model training*.

3.1 Feature Engineering

We instrument the PolarDB optimizer kernel to expose a comprehensive *142-dimensional* feature vector for every query plan. These raw features capture diverse aspects of query execution patterns organized into four semantic groups: (1) row and cost statistics including estimated cardinalities and I/O costs, (2) access-pattern counters tracking index usage and join methods, (3) plan-shape metrics measuring join depth and selectivity distributions, and (4) physical-meta signals such as buffer pool state and parallelism degree. Table 2 provides the complete categorization.

To identify the most predictive features while reducing computational overhead, we employ SHAP (SHapley Additive exPlanations)

Table 2: Feature groups extracted from the query optimizer.

Feature Group	Components and Description
Row & Cost Statistics (27 features)	Estimated rows, read/evaluation costs, total query cost, cardinality, cost ratios, and bytes scanned for quantifying data volume and computational complexity.
Access-Pattern Counters (32 features)	Access method distribution (range/ref/index/full scans), index usage, hash joins, temporary tables, and join method selections that profile query navigation patterns.
Plan-Shape Metrics (31 features)	Join tree depth, selectivity statistics, fan-out ratios, join imbalance, and subquery nesting that characterize the structural properties of execution plans.
Physical-Meta Signals (52 features)	Parallelism degree, buffer pool hits, compression flags, memory estimates, partition pruning, histogram coverage, and index quality metrics reflecting the physical execution environment.

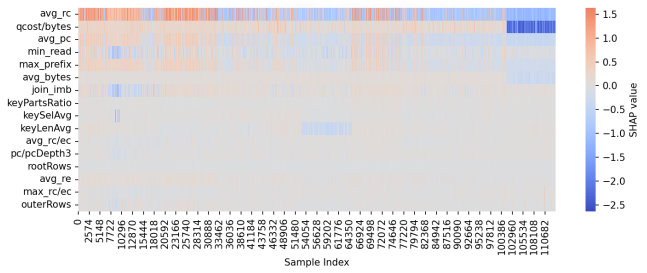

Figure 2: SHAP value heatmap: 16 of the selected 32 features. Darker colors indicate higher feature importance. Features are ordered by average importance.

analysis [28, 29] during offline training (Section 3.3). This game-theoretic approach quantifies each feature's contribution to the model's predictions across the entire training corpus. From the initial 142 features, we retain the top-32 that exhibit the highest SHAP values, achieving a 77% reduction in dimensionality with negligible no impact on prediction accuracy.

Figure 2 visualizes the SHAP values of the top 16 of the 32 selected features as a heatmap, revealing that cost-related features (`avg_rc`, `qcost/bytes`) and cardinality estimates (`rootRows`, `avg_re`) rank among the most influential predictors. The complete list of selected features with their descriptions is provided in Appendix B[4].

3.2 Data Collection

To construct a robust and representative training dataset that captures both widely adopted industry-standard benchmarks and realistic production workloads, we executed each query on both row-store and column-store engines, ensuring comprehensive performance labeling across diverse query characteristics and data distributions. The public side of the workload comes from TPC-H and TPC-DS at scale factors 1, 10 and 100, and HYBENCH at scale factors of 1 and 10. To keep the mix realistic we added seven production traces [15]: AIRLINE, CREDIT, CARCINOGENESIS, EMPLOYEE, FINANCIAL, GENEEA, HEPATITIS.

We generate query workloads by emulating real-world query patterns [15]. The workloads include queries with random joins adhering to schema definitions, predicates sampled from underlying data distributions (with a maximum of 20 predicates per query and a maximum of 10 joins), and complex logical expressions involving AND and OR operators. Additionally, queries incorporate aggregations with up to 3 operations and a maximum of 2 columns per aggregation, optional GROUP BY clauses (with a 20% probability of including a HAVING condition), and ORDER BY clauses.

All queries were executed on an instrumented PolarDB instance, from which we recorded the MySQL optimizer's JSON plan, the actual runtimes of the row and column executions, and the relevant statistics. Each workload contributes at least 10K valid query samples, for a total of more than 150K labeled examples.

[4]In the full version of this paper.

3.3 Model Training

3.3.1 Model Selection. We evaluate four candidate models for query dispatch: CART (single decision tree) [27], Random Forest [9], feed-forward neural networks (FNN) [5, 39], and LightGBM [19]. Based on empirical results (Section 5.4.3), we select LightGBM for its superior prediction accuracy and training efficiency.

3.3.2 Training Objective. We transform the binary classification problem into regression by using the log-difference target: $\log(\ell^i_{\text{row}}) - \log(\ell^i_{\text{col}})$. This formulation naturally captures relative performance differences and improves gradient behavior. We use standard L2 regression loss with custom sample weighting (described below). Additionally, we compress gradients by 0.3× when the log-difference falls below 5%, preventing overfitting to queries with similar execution times on both engines.

3.3.3 Self-paced Taylor-weighted Boosting. Standard uniform weighting treats all training samples equally, but dispatch errors have asymmetric costs—mispredicting a query with large latency difference causes more harm than one with similar performance on both engines. We develop a novel sample weighting scheme motivated by Taylor expansion of the latency objective. For a query with row latency R and column latency C, the expected latency under probabilistic dispatch $p \in [0, 1]$ is: $T(p) = (1 - p)R + pC = R + p(C - R)$. Taylor expansion around the optimal choice $p^* = \mathbb{1}_{\{C<R\}}$ yields: $T(p) - T(p^*) \approx |C - R| \cdot |p - p^*| + \frac{1}{2}(C - R)^2 \cdot (p - p^*)^2$. The first-order term shows that misprediction cost scales linearly with the latency gap $|C - R|$, which motivates prioritizing queries with large performance differences.

Multi-Factor Weighting Scheme. Furthermore, we assign each training sample i a weight that combines six factors:

(1) **Weight A - Class balance**: Inversely proportional to class frequency to handle label imbalance.
(2) **Weight B - Gap amplification**: Proportional to $|\log(R_i/C_i)|$, emphasizing queries with large relative performance differences.
(3) **Weight C - Dataset size**: Inversely proportional to $\sqrt{n_d}$ where n_d is dataset size, preventing large datasets from dominating training.
(4) **Weight D - Regret**: Proportional to $|R_i - C_i|$, directly from Taylor analysis—larger absolute latency differences receive higher weight.
(5) **Weight E - Focal adjustment**: Proportional to $[1 - 2|p_i - 0.5|]^2$ where p_i is model prediction. This emphasizes uncertain queries ($p_i \approx 0.5$) while down-weighting confident cases.
(6) **Weight F - Runtime scale**: Proportional to $\min(R_i, C_i)^\alpha$ with $\alpha < 1$, giving more importance to slower queries.

The final weight is the product of these factors, with soft clipping applied to prevent extreme values. The clipping bounds relax gradually across epochs to allow the model to focus on increasingly difficult examples.

Self-Paced Learning Procedure. After each epoch, we update each sample's predicted probability based on the current model. This creates a natural curriculum: early epochs treat all samples equally (since all start with $p_i = 0.5$), while later epochs automatically focus on challenging boundary cases where the model remains uncertain. This progression from uniform to focused learning improves robustness to noisy runtime measurements.

3.3.4 *Training Configuration.*

Hyperparameters. Our LightGBM configuration balances model capacity with training efficiency:

- **Boosting**: GOSS (Gradient One-Side Sampling) for efficient large-scale training.
- **Trees**: 400 per epoch, early stopping after 100 rounds without improvement.
- **Tree structure**: max depth 18, up to 256 leaves, min 20 samples per leaf.
- **Learning**: rate 0.045, decayed by 0.75× per epoch.
- **Regularization**: L2 weight 1.0, no L1 regularization.

Cross-Validation and Ensemble. We employ 5-fold stratified cross-validation to ensure robust generalization. Each fold maintains the original class distribution and is trained independently with self-paced weighting. From the five resulting models, we select the top three by balanced accuracy on validation sets. During offline testing, these models form an ensemble with majority voting—queries dispatched to the column engine if more than half predict positive.

4 ONLINE DISPATCH

In this section, we describe the three stages of *online dispatch* phase: *LightGBM predicting*, *residual learning*, and *resource regulating*.

4.1 Stage 1: LightGBM Predicting

Each incoming query q_t is mapped to the d-dimensional ($d = 32$) feature vector $\mathbf{x}_t$ (see Section 3.1). LightGBM outputs a *margin* $s_t \in \mathbb{R}$. Using the logistic link $\pi_t = \sigma(s_t) = \frac{1}{1+e^{-s_t}}$, where π_t is the posterior probability that q_t should be executed on the column engine (AP).

Rather than thresholding π_t we pass the *raw margin* s_t to the next stage: its sign suggests the preferred engine, while the magnitude encodes confidence [14]. This continuous prior enables the residual learner (Section 4.2) to focus exploration on low-confidence or mispredicted queries.

4.2 Stage 2: Residual Learning

At time t when a query finishes, we observe only the latency of the *chosen* engine ℓ_{a_t}. This creates a fundamental challenge: how can we learn whether the *other* engine would have been faster without actually running the query on both engines. To address this partial feedback problem, We maintain exponentially-weighted moving averages (EWMAs) of observed latencies for each engine[5]: $\hat{\ell_{\text{row}}}[t]$ and $\hat{\ell_{\text{col}}}[t]$. These serve as *counterfactual estimates* for the unobserved engine's performance.

we cast online adaptation as a *linear contextual bandit* [2, 26] that learns to refine the base LightGBM predictions using streaming latency observations. We employ Thompson Sampling for exploration, which provides a principled probabilistic approach to the exploration-exploitation trade-off.

[5]$\text{EWMA}_t = \alpha\ell_t + (1-\alpha)\text{EWMA}_{t-1}$ with smoothing factor $\alpha = 0.03$.

The *signed residual* is constructed as below, where Δ_t is positive exactly when the column engine is (expected to be) faster. The logarithmic transformation ensures numerical stability and reduces the impact of latency outliers.

$$\Delta_t = \begin{cases} -\log(1+\ell_{col}[t]) + \log(1+\hat{\ell}_{row}[t]), & a_t = 1 \\ \log(1+\hat{\ell}_{col}[t]) - \log(1+\ell_{row}[t]), & a_t = 0 \end{cases} \tag{2}$$

4.2.1 *LinTS-Delta: Bayesian Exploration for Runtime Adaptation.*

We model the residual between predicted and actual engine performance as a linear function of query features: $\mathbb{E}[\Delta_t \mid \mathbf{x}_t] = \mathbf{x}_t^\top \boldsymbol{\theta}^\star$, where $\boldsymbol{\theta}^\star$ captures how query characteristics correlate with prediction errors. LinTS-Delta employs Thompson Sampling [40, 46] to balance exploration and exploitation. The algorithm maintains a Bayesian posterior over $\boldsymbol{\theta}$ with regularization parameter $\lambda = 0.5$: $V_t = \lambda I + \sum_{\tau=1}^{t} \mathbf{x}_\tau \mathbf{x}_\tau^\top, \mathbf{b}_t = \sum_{\tau=1}^{t} \mathbf{x}_\tau \Delta_\tau$. The posterior mean is $\hat{\boldsymbol{\theta}}_t = V_t^{-1}\mathbf{b}_t$ with covariance matrix V_t^{-1}. At each time step, we sample from this posterior distribution: $\tilde{\boldsymbol{\theta}}_t \sim \mathcal{N}(\hat{\boldsymbol{\theta}}_{t-1}, \sigma^2 V_{t-1}^{-1})$. The Thompson score for the current query is then computed as: $u_t = \mathbf{x}_t^\top \tilde{\boldsymbol{\theta}}_t$.

This approach follows the linear contextual bandit framework [3, 25], where the covariance matrix V_t^{-1} quantifies uncertainty about $\boldsymbol{\theta}$—high uncertainty (large eigenvalues of V_t^{-1}) induces exploration through diverse samples, while low uncertainty leads to exploitation.

We then combine the static LightGBM prediction s_t with the learned residual u_t through:

$$z_t = \tanh(s_t + u_t) \in (-1, 1) \tag{3}$$

This fusion allows Thompson Sampling to refine rather than replace the base model, providing runtime adaptation while leveraging the strong baseline performance. The tanh function ensures bounded outputs suitable for downstream resource-aware dispatch.

Performance Guarantee. LinTS-Delta provides theoretical guarantees on its learning efficiency. The algorithm is proven to be a *no-regret* learner, meaning it learns from its dispatch mistakes and improves over time. Specifically, after processing T queries, the average extra latency caused by suboptimal dispatch decisions decreases proportionally to $\sqrt{1/T}$ [25, 40]. This means the system gets twice as accurate when it sees four times more queries.

4.3 Stage 3: Resource Regulating

The first two stages favor the fastest engine, but production HTAP systems must also respect CPU and memory budgets. In stage 3, our resource regulator tweaks the dispatch probability just enough to keep the row/column engines "fairly level" over time. The steps are illustrated below:

Step 1 – Measure Utilization. After each query we record the row-engine share of resource utilization $\mathbf{r}_t = (\rho_t^{\text{cpu}}, \rho_t^{\text{mem}}) \in [0,1]^2$.

Step 2 – Compute an imbalance score. Let the instantaneous deviation of the row engine's utilisation from its target be

$$\mathbf{e}_t = \mathbf{r}_t - \boldsymbol{\gamma}_t = \left(e_t^{\text{cpu}}, e_t^{\text{mem}}\right)^\top \in \mathbb{R}^2 \tag{4}$$

Periodically we recompute the empirical covariance matrix from the K most recent deviations,

$$\Sigma_t = \frac{1}{K-1}\sum_{k=1}^{K}(\mathbf{e}_{t-k} - \bar{\mathbf{e}})(\mathbf{e}_{t-k} - \bar{\mathbf{e}})^\top, \qquad \bar{\mathbf{e}} = \frac{1}{K}\sum_{k=1}^{K}\mathbf{e}_{t-k}, \tag{5}$$

which, in two dimensions, expands to

$$\Sigma_t = \begin{pmatrix} s^2_{\text{cpu}} & s_{\text{cpu,mem}} \\ s_{\text{cpu,mem}} & s^2_{\text{mem}} \end{pmatrix}, \tag{6}$$

where

$$s^2_{\text{cpu}} = \text{Var}\big[e^{\text{cpu}}\big], \quad s^2_{\text{mem}} = \text{Var}\big[e^{\text{mem}}\big], \quad s_{\text{cpu,mem}} = \text{Cov}\big[e^{\text{cpu}}, e^{\text{mem}}\big]. \tag{7}$$

We measure the imbalance magnitude with the Mahalanobis distance [32] and attach a sign based on the CPU deviation:

$$\begin{aligned} d_t &= \text{sgn}\big(e^{\text{cpu}}_t\big)\sqrt{\mathbf{e}_t^\top \Sigma_t^{-1}\mathbf{e}_t} \\ &= \text{sgn}\big(e^{\text{cpu}}_t\big)\sqrt{\frac{(e^{\text{cpu}}_t)^2 s^2_{\text{mem}} - 2e^{\text{cpu}}_t e^{\text{mem}}_t s_{\text{cpu,mem}} + (e^{\text{mem}}_t)^2 s^2_{\text{cpu}}}{s^2_{\text{cpu}} s^2_{\text{mem}} - s^2_{\text{cpu,mem}}}}. \end{aligned} \tag{8}$$

Finally we squash the signed distance to $[-1, 1]$:

$$r_t = \tanh(d_t),$$

so that positive values indicate the row engine is *over-utilizing* CPU/MEM relative to its target ($e^{\text{cpu}}_t > 0$), while negative values indicate *under-utilization.*

Step 3 – Fuse speed and resource signals. Recall $z_t \in (-1, 1)$ from Phase 2 (positive $\Rightarrow$ column engine predicted faster). We mix the two scores:

$$s^{\text{final}}_t = \omega_t z_t + (1-\omega_t)\, r_t, \quad \omega_t = 0.4 + 0.3 \cdot \sigma\big(\hat{\lambda}_t/5 - 1\big),$$

where $\hat{\lambda}_t$ is the current QPS estimate and $\sigma(x) = 1/(1+e^{-x})$.

- Low load (QPS ≈ 0) $\Rightarrow \omega_t \approx 0.481 \longrightarrow$ prioritize resource parity.
- High load (QPS $\longrightarrow +\infty$) $\Rightarrow \omega_t \approx 0.7 \longrightarrow$ prioritize latency.

Dispatch to the column engine iff $s^{\text{final}}_t > 0$.

Step 4 – Update the target using a simple OCO move. Periodically we look at the current resource error $g_t := \mathbf{r}_t - \boldsymbol{\gamma}_t$ and run an Online-Convex-Optimization (OCO) [54] step on the target share:

$$\boldsymbol{\gamma}_{t+1} = \Pi_{[0,1]^2}\big(\boldsymbol{\gamma}_t - \beta g_t\big), \qquad \beta = \frac{c}{\sqrt{t}} \ (c > 0). \tag{9}$$

Here $\Pi_{[0,1]^2}$ is the projection operator: it simply clips each coordinate so the target stays in the box $[0,1]^2$ (that is, never below 0 or above 1).

- **Column repeatedly wins on speed.** Suppose over the last 30s the column engine is almost always faster than the row engine. In that case the row engine's CPU/MEM share $\mathbf{r}_t$ is "too high for its performance," so $g_t = \mathbf{r}_t - \boldsymbol{\gamma}_t$ is *positive.* The update $\boldsymbol{\gamma}_{t+1} = \boldsymbol{\gamma}_t - \beta g_t$ therefore *reduces* the row-engine target and gives a larger share to the column engine for the next window.
- **Row engine is short on resources.** If monitoring shows the row engine now has *less* CPU/MEM than its target, then $\mathbf{r}_t$ drops below $\boldsymbol{\gamma}_t$ and g_t becomes *negative.* The same update rule $\boldsymbol{\gamma}_{t+1} = \boldsymbol{\gamma}_t - \beta g_t$ now *increases* the row-engine target, asking the dispatcher to send more queries its way and restore balance.

4.4 Online Query Dispatch Algorithm

Algorithm 1 integrates three stages of online query dispatch with continuous adaptation. After initialization (Lines 2-6), each query q_t is processed as follows: Stage 1 extracts features $\mathbf{x}_t$ and computes base score s_t via LightGBM (Lines 8-9). Stage 2 applies Thompson Sampling to generate residual adjustment u_t, yielding latency score $z_t = \tanh(s_t + u_t)$ (Lines 10-12). Stage 3 tracks resource deviation $\mathbf{e}_t$ and computes normalized Mahalanobis distance r_t for resource balancing (Lines 13-23). The final score $s^{\text{final}}_t = \omega_t z_t + (1-\omega_t) r_t$ uses load-adaptive weighting ω_t, with positive values selecting the column engine (Lines 24-26). Post-execution, the algorithm updates EWMA latencies, LinTS posterior using log-residual rewards, and resource targets via online convex optimization (Lines 28-33).

4.5 Performance Guarantee

AQD provides theoretical guarantees on both latency optimization and resource balance:

Theorem 4.1. *Let a horizon of T queries be served by the three-phase pipeline described in Sections 4.1–4.3. Assume:*

(1) ***bounded latency:*** $\max_t \ell_{a_t} \le L_{\max}$*;*
(2) ***bounded resources:*** $\|\mathbf{r}_t\|_2 \le 1\ \forall t$*;*
(3) ***weight constraint:*** *the scalar weight satisfies* $\alpha_t \ge 0.48\ \forall t$*.*

Then there exists constants $C_1 > 0$ and $C_2 > 0$ such that

$$\frac{1}{T}\sum_{t=1}^{T}\big(\ell_t(a_t) - \min_a \ell_t(a)\big) \le \frac{C_1\sqrt{\log T}}{\sqrt{T}}, \tag{10}$$

$$\frac{1}{T}\sum_{t=1}^{T}\big\|\mathbf{r}_t - \boldsymbol{\gamma}_t\big\|_2^2 \le \frac{C_2}{\sqrt{T}}. \tag{11}$$

Equation (10) *bounds the* average extra latency *with respect to the optimal;* (11) *bounds the* average squared resource drift *relative to the adaptive target* $\boldsymbol{\gamma}_t$.

The proof sketch of Theorem 4.1 is shown in Appendix C[6].

5 EXPERIMENTAL EVALUATION

We evaluate AQD through comprehensive experiments covering both offline model training and online query dispatch under varying concurrency levels.

5.1 Experimental Setup

Hardware. All experiments run on a dual-socket server equipped with two Intel® Xeon Platinum 8269CY processors (26 cores, 52 threads each, base 2.5 GHz, turbo 3.8 GHz) and 768 GB DDR4 RAM.

Implementation. AQD is fully integrated into PolarDB at the kernel level. The dispatcher is implemented in modern C++, linking against the native LightGBM C API for inference [19]. The trained model (approximately 10 MB) is loaded by the engine. Feature extraction and inference combined add about 500 μs overhead per query.

[6]In the full version of this paper.

Algorithm 1 AQD: Unified online adaptive dispatch with Thompson Sampling & Mahalanobis balancing

```
Require: LightGBM model $f$ (offline–trained); LinTS hyper-parameters $(\lambda, \sigma^2)$; OCO step-size constant $c$; update period $S$ (seconds); buffer length $K$
Ensure: Engine choice $a_t \in \{0, 1\}$ for every incoming query $q_t$
1: Initialise:
2:     $V \leftarrow \lambda I_d$, $\mathbf{b} \leftarrow \mathbf{0}$                          ▷ LinTS posterior stats
3:     $\hat{\ell}^{\text{row}} \leftarrow 0$, $\hat{\ell}^{\text{col}} \leftarrow 0$                              ▷ EWMA latencies
4:     $\boldsymbol{\gamma}_1 \leftarrow (0.5, 0.5)$                      ▷ row-engine CPU/MEM target
5:     $\Sigma_1 \leftarrow I_2$, $\mathcal{B} \leftarrow \varnothing$                          ▷ covariance, buffer
6:     $lastCov$ ←Now, $lastOCO$ ←Now
7: for each query $q_t$ arriving at time $t$ do
8:     /* Stage 1 – LightGBM Predicting */
9:     $\mathbf{x}_t \leftarrow$ExtractFeatures$(q_t)$; $s_t \leftarrow f(\mathbf{x}_t)$
10:    /* Stage 2 – Residual Learning */
11:    sample $\tilde{\boldsymbol{\theta}}_t \sim \mathcal{N}(V^{-1}\mathbf{b}, \sigma^2 V^{-1})$
12:    $u_t \leftarrow \mathbf{x}_t^\top \tilde{\boldsymbol{\theta}}_t$; $z_t \leftarrow \tanh(s_t + u_t)$                 ▷ $z_t \in (-1, 1)$
13:    /* Stage 3 – Resource Regulating */
14:    $\mathbf{r}_t \leftarrow (\rho_t^{\text{cpu}}, \rho_t^{\text{mem}})$
15:    $\mathbf{e}_t \leftarrow \mathbf{r}_t - \boldsymbol{\gamma}_t$
16:    append $\mathbf{e}_t$ to $\mathcal{B}$ and drop oldest if $|\mathcal{B}| > K$
17:    if Now − $lastCov \geq S$ then
18:        $\Sigma_t \leftarrow$ COV$(\mathcal{B})$
19:        $lastCov$←Now
20:    end if
21:    $d_t \leftarrow \operatorname{sgn}(e_t^{\text{cpu}})\sqrt{\mathbf{e}_t^\top \Sigma_t^{-1} \mathbf{e}_t}$
22:    $r_t \leftarrow \tanh(d_t)$
23:    $\omega_t \leftarrow 0.4 + 0.3 \cdot \sigma(\hat{\lambda}_t/5 - 1)$
24:    $s_t^{\text{final}} \leftarrow \omega_t z_t + (1 - \omega_t) r_t$
25:    $a_t \leftarrow \mathbf{1}[s_t^{\text{final}} > 0]$                              ▷ 1: column, 0: row
26:    $\ell_{a_t} \leftarrow$Execute$(q_t, a_t)$                              ▷ actual latency
27:    $\hat{\ell}^{(a_t)} \leftarrow \alpha\,\ell_{a_t} + (1 - \alpha)\,\hat{\ell}^{(a_t)}$            ▷ update EWMA latency
28:    $$\Delta_t = \begin{cases} -\log(1 + \ell_{a_t}) + \log(1 + \hat{\ell}^{\text{row}}), & a_t = 1 \\ -\log(1 + \hat{\ell}^{\text{col}}) + \log(1 + \ell_{a_t}), & a_t = 0 \end{cases}$$
                                                           ▷ residual reward
29:    $V \leftarrow V + \mathbf{x}_t \mathbf{x}_t^\top$; $\mathbf{b} \leftarrow \mathbf{b} + \mathbf{x}_t\,\Delta_t$
30:    if Now − $lastOCO \geq S$ then
31:        $\beta_t \leftarrow c/\sqrt{t}$
32:        $\boldsymbol{\gamma}_{t+1} \leftarrow \Pi_{[0,1]^2}[\boldsymbol{\gamma}_t - \beta_t(\mathbf{r}_t - \boldsymbol{\gamma}_t)]$
33:        $lastOCO$←Now
34:    end if
35: end for
```

5.2 Baselines

We compare two variants of AQD (LightGBM static and LightGBM dynamic) against four representative approaches:

(1) **Row-only**: Dispatches all queries to the row engine.
(2) **Column-only**: Dispatches all queries to the column engine.
(3) **Cost-threshold**: PolarDB's default strategy—dispatches to column engine when optimizer cost exceeds 5×10^4, otherwise to row engine.
(4) **Hybrid Optimizer**: Uses linear regression to map optimizer costs to actual runtimes, then selects the engine with lower predicted latency [8].

LightGBM static only uses the prediction from offline trained LightGBM model to dispatch queries, while LightGBM dynamic includes residual learning and resource regulating.

Additionally, we evaluate three ML-based alternatives (Decision Tree, Random Forest, Feed-forward Neural Network) in Section 5.4.3 to validate our choice of LightGBM.

5.3 Evaluation Metrics

We employ different metrics for offline and online evaluation phases:

Metrics for Query-Level Dispatch. We evaluate query-level dispatch using several metrics: (1) **Prediction Quality** measures standard classification metrics including accuracy, macro-precision, macro-recall, and macro-F$_1$ score, with separate F$_1$ scores reported for "easy" queries (where cost-threshold is correct) and "hard" queries (where cost-threshold is wrong); (2) **Average Runtime** measures the average runtime across all queries in a workload in seconds; (3) **Overall Improvement** captures the relative latency reduction compared to the cost-threshold baseline, calculated as improvement = $\frac{\text{avg-rt}_{\text{cost-thr}} - \text{avg-rt}_{\text{method}}}{\text{avg-rt}_{\text{cost-thr}}}$; and (4) **Improvement → Optimal** represents the fraction of theoretically achievable improvement captured, computed as $\frac{\text{avg-rt}_{\text{cost-thr}} - \text{avg-rt}_{\text{method}}}{\text{avg-rt}_{\text{cost-thr}} - \text{avg-rt}_{\text{optimal}}}$.

Metrics for Workload-Level Dispatch. For workload-level dispatch evaluation, we track: (1) **Makespan**, the total wall-clock time from first query arrival to last query completion, which captures system throughput under load; (2) **Average Latency**, measuring the mean per-query execution time under concurrent load; (3) **P95 Latency**, the 95th percentile query latency that indicates tail behavior; and (4) **Resource Utilization**, tracking CPU percentage and memory consumption separately for row and column engines to measure resource balance and efficiency.

5.4 Offline Preparation

We first evaluate the offline LightGBM model's ability to predict optimal engine selection without runtime adaptation.

5.4.1 Single-Dataset Results. Across the 15 workloads in Table 3, **LightGBM delivers the lowest latency on 14 of them**. Relative to the cost-threshold rule it trims runtime by a *median 17 %*, reaching 54 % on `tpch100`, 44 % on `airline`, 65 % on `employee`. It improves over cost-threshold method up to 99% towards the optimal. Accuracy remains strong ($\geq$ 0.85 on 11 workloads). The only outlier is the highly imbalanced `credit` trace, where the column engine alone is 5 % faster than LightGBM, yet the model still beats the cost rule by 17 %. F$_1$ is mostly above 0.9 for "easy" queries (the cost-threshold method predicts them correctly), and has improvement over the cost-threshold method on "hard" queries (the cost threshold method misclassifies them).

5.4.2 Cross-Dataset Test. To test the model's generalization across datasets, we conduct a *leave-three-datasets-out* study. The fifteen datasets are partitioned into five disjoint test groups (each group

Table 3: Prediction quality and average runtime (s) on individual datasets.
↑ = higher is better; ↓ = lower is better.

	Dataset														
Metric ↑/↓	`tpch1`	`tpch10`	`tpch100`	`tpcds1`	`tpcds10`	`tpcds100`	`hybench_sf1`	`hybench_sf10`	`airline`	`credit`	`carcinogenesis`	`employee`	`financial`	`geneea`	`hepatitis`
row avg-rt ↓	5.4970	21.0160	8.4372	4.4619	10.4921	18.3224	2.5789	8.3067	0.1371	3.8453	0.0364	0.1367	0.5373	0.3610	**0.0023**
col avg-rt ↓	0.9219	3.9870	6.8619	0.5928	2.0995	3.0967	0.6385	2.0188	0.0802	**0.6044**	0.0195	0.0368	0.0911	0.1351	0.0378
cost-thr. avg-rt ↓	0.9248	4.1090	6.9390	0.5944	2.3815	3.2861	0.5955	1.8997	0.0628	0.7663	0.0149	0.0636	0.1001	0.1183	**0.0023**
hybrid avg-rt ↓	0.9616	4.4140	7.6116	0.6028	2.3927	3.5906	0.8604	2.9427	0.0621	0.6347	0.0237	0.0335	0.2100	0.1278	0.0217
LightGBM avg-rt ↓	**0.9206**	**4.036**	**3.1683**	**0.5872**	**2.1460**	**3.1795**	**0.5609**	**1.7424**	**0.0350**	0.6361	**0.0109**	**0.0223**	**0.0826**	**0.1139**	**0.0023**
optimal avg-rt ↓	0.9008	3.8300	2.7906	0.5637	1.6681	2.2593	0.5176	1.5222	0.0307	0.5857	0.0108	0.0218	0.0810	0.1066	**0.0023**
accuracy ↑	0.8677	0.8584	0.9309	0.7840	0.7917	0.8106	0.8965	0.8921	0.9323	0.7275	0.9668	0.9892	0.9376	0.9335	0.9997
macro precision ↑	0.8393	0.8276	0.8817	0.7296	0.6965	0.7666	0.8967	0.8873	0.8138	0.5298	0.6598	0.9803	0.7303	0.7320	0.4998
macro recall ↑	0.9879	0.9547	0.8367	0.9662	0.9444	0.9110	0.8849	0.8728	0.8701	0.9585	0.8889	0.9956	0.9420	0.8466	0.5000
macro F_1 ↑	0.9076	0.8867	0.8586	0.8314	0.8017	0.8326	0.8899	0.8792	0.8410	0.6824	0.7574	0.9879	0.8228	0.7851	0.4999
F_1 (easy) ↑	0.9863	0.9804	0.9222	0.9569	0.9520	0.9347	0.9406	0.9239	0.8731	0.6741	0.1778	0.9619	0.8264	0.7948	0.5000
F_1 (hard) ↑	0.6609	0.5244	0.2286	0.4719	0.3289	0.4478	0.6992	0.7424	0.8000	0.6918	0.9677	0.9957	0.8108	0.7829	0.0000
overall impr. ↑	0.0045	0.0177	0.5434	0.0122	0.0989	0.0324	0.0581	0.0828	0.4431	0.1699	0.2659	0.6491	0.1748	0.0376	0.0000
impr. → optimal ↑	0.1722	0.2609	0.9089	0.2360	0.3302	0.1038	0.4439	0.4168	0.8677	0.7209	0.9700	0.9867	0.9153	0.3797	0.0000

Note: Bold values indicate the best performance among row-only, column-only, cost threshold, hybrid optimizer, and LightGBM methods.

contains three datasets and appears exactly once as the *unseen* test set; the remaining twelve are used for training).

Table 4 summarizes the results. Against MySQL's *cost-threshold* rule (our baseline for *improvement*), LightGBM reduces mean latency by **42 %**, **14 %**, **13 %**, **43 %**, and **22 %** on the five splits, reclaiming **87 %**, **75 %**, **65 %**, **81 %**, and **71 %** of the oracle's headroom while keeping accuracy mostly around 90%. LightGBM achieves f1 score over 0.79 on all test sets.

5.4.3 *Model Comparison.*

To evaluate the impact of the learning algorithm itself, we train four classifiers—LightGBM, Random Forest (RF), a single Decision-Tree (DT), and feed forward neural network (FNN) —on the *same* features and training data. Table 5 compares their performance.

LightGBM delivers the lowest average runtime (**2.41 ms**), exceeding the cost-threshold rule by **41.5 %** and recovering **96.8 %** of the optimal's headroom, while maintaining the highest macro-F_1 and hard-query F_1. RF provides a moderate gain but is 18 % slower than LightGBM. Both DT and FNN fail to improve much over the heuristic baseline.

5.4.4 *Ablation Study.*

We ablate two implementation knobs—*hard-example replay* and the six sample-weight terms *A*–*F* introduced in Section 3.3.3—by disabling them one at a time. Turning off any single component hurts both runtime and quality, showing that each contributes additively. Hard-example replay is the most influential: removing it raises mean latency by +3.2% and lowers macro-F_1 by 0.032, while the hard-query F_1 drops from 0.646 to 0.586. The detailed results are shown in Appendix D[7].

5.5 Online Dispatch

We now evaluate AQD's performance under concurrent query execution, where runtime adaptation becomes critical for handling dynamic workloads and resource contention in production environments. The generation of the queries is described in Section 3.2.

[7]In the full version of this paper.

5.5.1 *Comparison between Four Dispatch Methods.*

To isolate the benefits of online adaptation, we conduct a comprehensive evaluation comparing four dispatch strategies under increasing concurrency levels from 200 to 1000 concurrent queries. The `Cost Threshold` and `Hybrid Optimizer` represent state-of-the-art rule-based approaches currently deployed in production systems [47], while `LightGBM Static` employs our offline-trained gradient boosting model without runtime adaptation. The `LightGBM Dynamic` variant incorporates our novel LinTS-Delta algorithm, combining Bayesian exploration with resource-aware regulation to address the exploration exploitation tradeoff in online settings.

Performance Analysis. Table 6 shows learning-based approaches outperform traditional methods with 17.9-84.7% makespan reduction across all concurrency levels. Dynamic dispatch benefits vary by workload: at moderate loads (200-600 queries), it improves makespan by 1.9-12.8% over static dispatch, peaking at 600 queries where runtime adaptation best handles resource contention. At extreme concurrency (1000 queries), dynamic dispatch excels with 21.1% better average latency (1.365s vs 1.730s), 28.0% better P95 latency versus static dispatch, and 63.5% latency reduction versus traditional approaches. This validates that LinTS-delta's Thompson Sampling provides robust exploration-exploitation balance for superior adaptability across varying workloads.

Resource Efficiency and Stability. Figure 3 reveals striking differences in resource utilization during 800 concurrent query execution. Traditional strategies exhibit severe inefficiency with volatile CPU patterns—the hybrid optimizer peaks at 36% total CPU utilization with the column engine alone consuming 29%, while learned methods stabilize at 2-3% after a 40-second warm-up period, achieving a 12× reduction. The column engine shows the most dramatic improvement, dropping from sustained 10-20% usage in traditional methods to under 2% with ML-based routing. Memory consumption further validates our approach: traditional strategies grow from 115GB to over 130GB with high volatility, while LightGBM variants maintain stable footprints around 111GB. This rapid convergence to efficient resource utilization demonstrates that intelligent query dispatch prevents engine thrashing and enables predictable performance characteristics essential for production deployments.

Table 4: Cross-dataset evaluation. Each row trains on twelve datasets and tests on the three unseen datasets shown in Test Set. ↑/↓ indicate the desired direction.

Test Set	row avg-rt↓	col avg-rt↓	cost-thr. avg-rt↓	hybrid avg-rt↓	LightGBM avg-rt↓	oracle avg-rt↓	acc↑	macro F_1↑	overall impr.↑	impr.→oracle↑
tpcds_sf10 + hybench_sf10 + tpch_sf100	6.4601	2.3459	2.2719	2.7930	1.3194	1.1772	0.8939	0.8933	0.4193	0.8701
carcinogenesis + employee + hybench_sf1	0.9782	0.2566	0.2484	0.3353	0.2150	0.2036	0.9177	0.9155	0.1347	0.7460
tpch_sf1 + airline + hepatitis	1.2654	0.1896	0.1848	0.2403	0.1612	0.1483	0.9184	0.9009	0.1280	0.6476
tpch_sf10 + credit + tpcds_sf100	2.1314	0.6332	0.6678	0.7735	0.3804	0.3118	0.8021	0.7920	0.4303	0.8072
tpcds_sf1 + geneea + financial	0.6593	0.1142	0.1184	0.1239	0.0920	0.0813	0.9022	0.8727	0.2233	0.7122

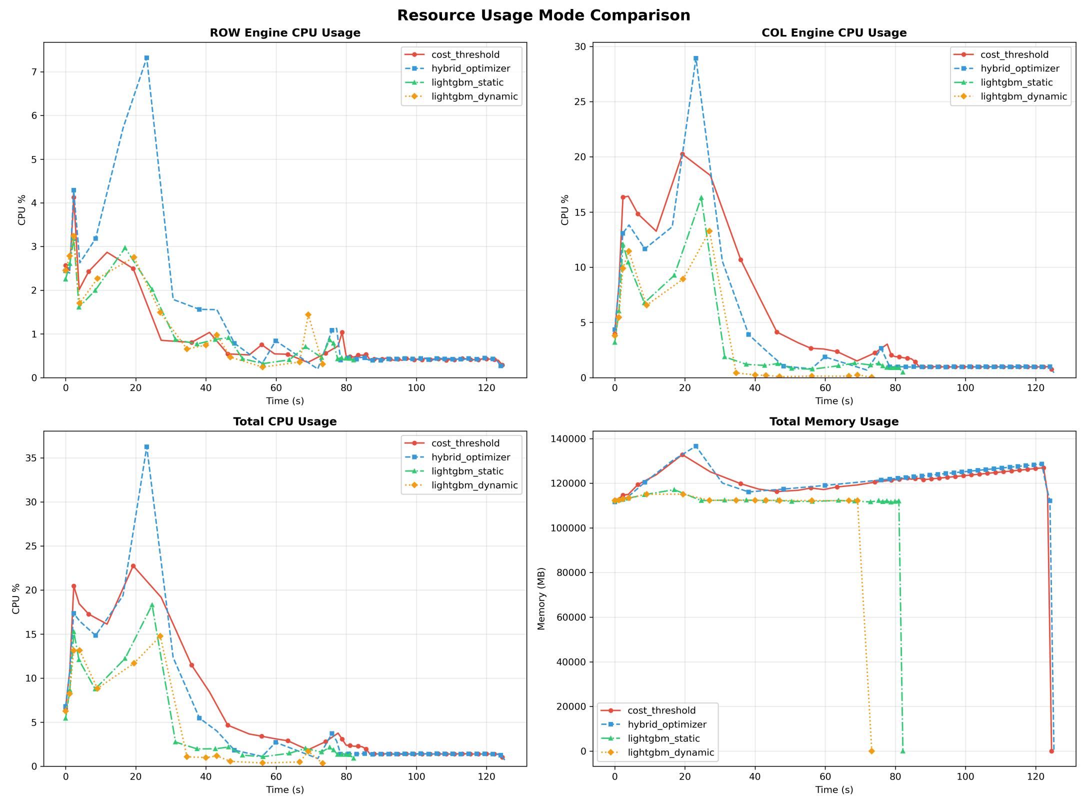

Figure 3: Resource utilization profiles during 800-query concurrent execution across four dispatch strategies. Learned methods achieve stable resource consumption after initial warm-up.

Table 5: Comparison among Different AI Models

Metric	LightGBM	Random Forest	Decision Tree	FNN
Avg. runtime (ms) ↓	**2.41**	2.86	4.10	4.21
Accuracy ↑	**0.912**	0.784	0.424	0.491
Macro F_1 ↑	**0.911**	0.784	0.298	0.431
F_1 (hard) ↑	**0.724**	0.456	0.124	0.270
Overall impr. ↑	**41.5%**	30.5%	0.5%	−2.1%
Impr. → optimal ↑	**96.8%**	71.1%	1.3%	−4.8%

5.6 Concurrent Execution on Standard Benchmarks

We evaluate AQD on industry-standard benchmarks under concurrent execution of their official template queries. Table 7 compares dispatch strategies across TPC-DS, TPC-H, and ClickBench workloads.

Learning-based approaches outperform traditional methods on all benchmarks. Dynamic dispatch achieves the best results: 4.3% latency reduction on TPC-DS, 29.1% on TPC-H, and 11.7% on ClickBench compared to cost threshold. The performance gap between static and dynamic variants is 3.2% on TPC-DS (3.2%). ClickBench shows the largest makespan improvement (8.3%), demonstrating effectiveness on real-world workloads. These results validate that AQD's adaptive approach successfully handles diverse query patterns while maintaining stable performance across different benchmark characteristics.

Table 8 compares four dispatch strategies on HTAP benchmark hybench [50]. Our LightGBM-based methods (`LightGBM Static` and `LightGBM Dynamic`) achieve higher overall HTAP scores than

Table 6: Performance Comparison of Dispatch Strategies under Varying Concurrency Levels.

Dispatch Strategy	Makespan (s)↓	Avg Latency (s)↓	P95 Latency (s)↓
	200 Concurrent Queries		
Cost Threshold	127.11	1.649 0	6.939 0
Hybrid Optimizer	126.45	1.624 0	6.569 0
LightGBM Static	20.42	0.485 0	1.623 0
LightGBM Dynamic	**19.49**	**0.423** 0	**1.356** 0
Improvement	84.7%	74.4%	80.5%
	400 Concurrent Queries		
Cost Threshold	126.61	1.218 0	4.367 0
Hybrid Optimizer	125.80	1.334 0	5.558 0
LightGBM Static	22.88	0.576 0	2.701 0
LightGBM Dynamic	**22.45**	**0.524** 0	**1.613** 0
Improvement	82.3%	57.0%	63.1%
	600 Concurrent Queries		
Cost Threshold	126.44	1.191 0	5.138 0
Hybrid Optimizer	126.92	1.271 0	4.967 0
LightGBM Static	44.36	0.766 0	3.110 0
LightGBM Dynamic	**38.67**	**0.668** 0	**2.641** 0
Improvement	69.4%	43.9%	48.6%
	800 Concurrent Queries		
Cost Threshold	125.83	2.104 0	9.193 0
Hybrid Optimizer	126.49	1.773 0	8.167 0
LightGBM Static	83.43	1.297 0	4.941 0
LightGBM Dynamic	**74.50**	**1.141** 0	**4.562** 0
Improvement	40.8%	45.8%	50.4%
	1000 Concurrent Queries		
Cost Threshold	137.36	3.738 0	16.074 0
Hybrid Optimizer	125.86	2.910 0	11.842 0
LightGBM Static	113.59	1.730 0	6.881 0
LightGBM Dynamic	**112.79**	**1.365** 0	**4.955** 0
Improvement	17.9%	63.5%	69.2%

Note: Bold values indicate best performance for each metric.

Table 7: Performance on Standard Benchmarks

Benchmark	Strategy	Makespan (s)	Avg Latency (s)
TPC-DS	Cost Threshold	171.84	104.98
	Hybrid Optimizer	171.84	105.31
	LightGBM Static	172.18	103.90
	LightGBM Dynamic	**171.27**	**100.50**
TPC-H	Cost Threshold	145.95	105.63
	Hybrid Optimizer	145.90	106.18
	LightGBM Static	**145.73**	75.18
	LightGBM Dynamic	145.76	**74.87**
ClickBench	Cost Threshold	407.77	48.47
	Hybrid Optimizer	384.27	43.70
	LightGBM Static	374.88	42.99
	LightGBM Dynamic	**373.95**	**42.78**

the baselines. Among them, `LightGBM Dynamic` delivers the best performance (2.10), with strong gains in both transactional and mixed workloads.

Table 8: Hybench Benchmark Results for Different Dispatch Strategies.

Method	AP-QPS ↑	TP-TPS ↑	XP-QPS ↑	XP-TPS ↑	HTAP-Score ↑
Cost Threshold	**0.21**	29.78	**0.68**	0.47	1.93
Hybrid Optimizer	0.20	20.25	0.62	0.65	1.73
LightGBM Static	**0.21**	32.77	0.48	0.68	2.00
LightGBM Dynamic	**0.21**	**33.97**	0.52	**0.78**	**2.10**

Note: QPS/TPS denote throughput for AP/TP workloads, XP-QPS/XP-TPS denote HTAP workload performance, and the HTAP-Score is the geometric mean across components.

5.7 RL Learning Process

Figure 4 demonstrates the online learning behavior of our LinTS-Delta bandit. We can see the bandit is learning the residual between actual latency and estimated latency and updating adaptively, especially at the beginning and during the 1500 and 2200 queries.

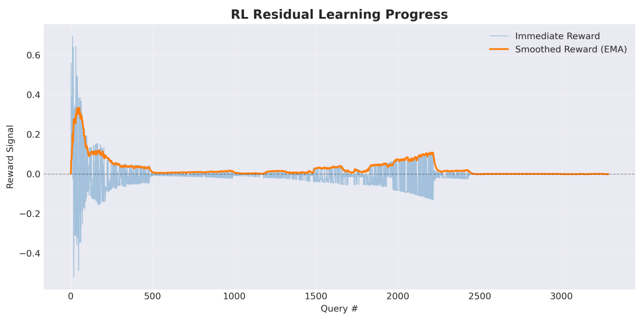

Figure 4: RL learning process

6 DISCUSSION

We analyze AQD's design decisions, examine the factors driving its performance gains, and discuss limitations and future directions.

Offline-Online Synergy. AQD's two-phase architecture reflects a fundamental tradeoff in learned database systems: the need for both predictive accuracy and runtime adaptability.

Offline Learning. The offline phase distills 142 raw features into 32 SHAP-selected signals and trains a 10MB LightGBM ensemble using regret-weighted boosting. This phase captures stable query-to-engine mappings from historical workloads, encoding complex decision boundaries that no fixed threshold can express. The model achieves up to 87% of optimal performance on isolated queries, providing a strong foundation for runtime decisions.

Online Adaptation. Runtime dynamics—resource contention, workload shifts, and queueing effects—necessitate online correction. Our LinTS-Delta bandit learns residual errors between predicted and actual performance, while the OCO-based resource controller maintains CPU/memory balance through Mahalanobis distance scoring. This dual adaptation mechanism reduces makespan by up to 52% compared to static dispatch at high concurrency (Table 6), demonstrating that even accurate offline models benefit from online refinement.

The key insight is that offline and online components address complementary aspects: the former captures query-intrinsic properties (selectivity, join patterns), while the latter handles system-extrinsic factors (load, resource availability). This separation of concerns enables sub-millisecond decisions while maintaining theoretical guarantees on both regret and resource deviation.

Limitations and Future Directions. We summarize the limitations of our method as below and point out future research directions:

(1) **Plan-Level Granularity.** AQD currently makes binary engine choices per query. Fine-grained operator-level dispatch could further improve performance, particularly for queries with mixed OLTP/OLAP characteristics. In the future, we plan to train machine learning model to dispatch each operator to row engine or column engine to achieve better optimization.

(2) **Model Maintenance.** While our system includes mechanisms for online adaptation, long-term model drift remains a challenge. We plan to address this through two complementary approaches:
   (a) *Periodic retraining*: A background process monitors prediction accuracy and triggers retraining when validation regret exceeds thresholds.
   (b) *Incremental learning*: During low-utilization periods, the system selectively re-executes queries on both engines to gather fresh labels, updating the model through mini-batch gradient boosting without service interruption.

(3) **Generalization across Systems.** AQD's core principles — feature engineering, regret-weighted training, and bandit-based learning—directly apply to other dual-engine HTAP systems. The approach can be readily transferred to TiDB, MySQL HeatWave, and Postgres paired with DuckDB. Only the feature extraction layer requires system-specific adaptation, while the learning pipeline remains unchanged.

(4) **Interpretability.** Future work could integrate learned decision rules with natural language generation to produce human-readable explanations for dispatch decisions (e.g., "Dispatched to column engine due to high join fanout (100×) on `lineitem` table with aggregation operations").

7 RELATED WORK

We list research on HTAP architectures, query dispatch techniques, and ML-based database optimization approaches.

HTAP Architectures. HTAP systems [44, 51] can be classified into: (i) Single-Engine Systems. These maintain both row and column formats within one engine. SAP HANA [13] pioneered unified storage with per-operator format selection. Oracle Database In-Memory [23] and SQL Server Columnstore [11, 24] add columnar capabilities to row engines. HyPer [20] uses fork-based snapshots for isolation. StarRocks [45] is a modern MPP analytical database that combines a vectorized execution engine, columnar storage, and real-time ingestion to provide high-performance HTAP capabilities. (ii) Dual-Engine Systems. They separate OLTP and OLAP engines with data synchronization. TiDB + TiFlash [16] uses Raft-based replication, ByteHTAP [10] achieves sub-second freshness, and PolarDB [47] introduces row-column fusion operators. SingleStore [35] and Snowflake Unistore [18] follow similar patterns.

Table 9: Design space for row/column dispatch strategies.

Granularity	Pros	Cons
Static layout [37]	(1) No run-time choice; (2) simple	(1) Cannot combine row and column strengths
Plan-level [6]	(1) Smaller search space; (2) No intra-query engine switches	(1) Cannot mix layouts when only part of a plan benefits
Operator-level [12, 16, 23, 24, 47]	(1) Highest potential performance; (2) Fine-grained optimisation	(1) Large search space; (2) More complex implementation

Query Dispatch Strategies. Query dispatch varies along two dimensions: granularity and methodology. Table 9 summarizes dispatch granularities. Static-layout engines like IBM DB2 BLU [37] eliminate runtime decisions entirely. Plan-level dispatch, exemplified by MySQL HeatWave [6], dispatches entire queries to a single engine, balancing flexibility with implementation simplicity. Operator-level systems—including SQL Server [24], Oracle In-Memory [23], SAP HANA [22], TiDB [16], and PolarDB [47] — enable fine-grained per-operator engine selection, maximizing performance potential at the cost of increased optimization complexity.

Dispatch methodologies include cost-based, rule-based, and learning-based. Most systems employ cost models [24, 47] or manually tuned heuristics [12]. Recent work explores machine learning: ByteHTAP [10] mentions that it applies tree-CNNs for plan-level dispatch decisions but does not provide experimental results. However, existing dispatch approaches remain static after deployment, unable to adapt to workload shifts or system dynamics.

Learning-Based Database Optimization. Recent work has explored machine learning for database optimization. Neo and Bao enhance traditional cost models using gradient-boosted trees [30, 31], while Wu et al. apply graph neural networks for query memory prediction [48]. Online learning approaches include deep RL for join ordering [21] and multi-armed bandits for index selection [34, 49]. Resource-aware schedulers such as Auto-WLM [42], LSched [41], and buffer-sensitive RL [52] incorporate system constraints but do not address query-level row/column dispatch under dynamic HTAP workloads. Our work fills this gap by combining offline learning with online adaptation for dispatch decisions that balance performance and resource utilization.

8 CONCLUSION

AQD combines supervised learning, contextual bandits, and resource-aware control to improve query dispatch in HTAP systems. Our prototype implementation in PolarDB achieves up to 42% latency reduction across diverse workloads, reaching 87% of optimal performance. Under high concurrency, the system reduces latency by over 40% while maintaining stable resource utilization, with approximately 500 μs dispatch overhead. The key finding is that integrating offline prediction with online adaptation outperforms single-model approaches. This work demonstrates the feasibility of our online adaptive query dispatcher framework in HTAP database systems.

Table 10: Top-32 features selected by SHAP importance

Index	Name	Description
`f02`	`avg_rc`	Average read cost per table (I/O cost estimate)[1]
`f96`	`qcost/bytes`	Query cost normalized by total bytes read
`f03`	`avg_pc`	Average prefix cost (cumulative cost up to each table)[2]
`f23`	`min_read`	Minimum read cost among all tables in the plan
`f22`	`max_prefix`	Maximum prefix cost (identifies dominant branches)
`f05`	`avg_bytes`	Average data volume per table (row_length × rows)
`f81`	`join_imb`	Maximum join branch imbalance ratio[3]
`f124`	`keyPartsRatio`	Fraction of index columns actually used
`f126`	`keySelAvg`	Average leading-column selectivity of chosen indexes[4]
`f125`	`keyLenAvg`	Average key length in bytes (normalized by 256)
`f29`	`avg_rc/ec`	Average ratio of read cost to evaluation cost
`f50`	`pc/pcDepth3`	Cost amplification ratio beyond join depth 3[5]
`f32`	`rootRows`	Final query output cardinality (log-transformed)
`f00`	`avg_re`	Average rows examined per table
`f30`	`max_rc/ec`	Maximum read-to-evaluation cost ratio
`f31`	`outerRows`	Cardinality of first non-trivial table access[6]
`f94`	`log_re`	Total examined rows (log-scaled with 10^8 cap)
`f34`	`pcDepth3`	Cumulative cost at join depth 3
`f16`	`avg_sel`	Average selectivity across all tables[7]
`f18`	`join_depth`	Maximum join tree depth
`f01`	`avg_rp`	Average prefix rowcount (rows flowing through)
`f09`	`eq_ref_ratio`	Fraction of equality lookups (eq_ref)
`f27`	`rc/re`	I/O cost per examined row
`f26`	`pc/rc`	Total-to-read cost ratio (evaluation overhead)
`f96`	`log_qcost`	Total query cost (log-scaled with 10^{10} cap)
`f97`	`qcost/rows`	Cost per output row
`f15`	`max_sel`	Maximum selectivity (identifies full scans)
`f93`	`log_rootRows2`	Alternative scaling of output cardinality
`f28`	`ec/re`	Evaluation cost per examined row
`f33`	`fanout`	Fan-out amplification from outer to final rows
`f54`	`late_fanout`	Maximum selectivity at depth ≥ 4
`f08`	`ref_ratio`	Fraction of non-unique index lookups (`ref`)

[1]Read cost estimates disk I/O operations. All costs use MySQL's abstract cost units where 1.0 typically represents a random page read.
[2]Prefix cost accumulates both read and evaluation costs from the first table through the current position in the join order.
[3]Computed as $\log(1+\max(l_i, r_i)/\min(l_i, r_i))$ for adjacent join branches, where l_i, r_i are the prefix rowcounts. High values indicate skewed hash joins.
[4]Derived from MySQL's `rec_per_key` statistics. Selectivity = `rec_per_key[0]` / `table_rows`. Lower values indicate more selective (better) indexes.
[5]Identifies queries where costs explode after the third join. High ratios suggest poor join ordering or missing indexes in deep branches.
[6]The first table accessed with a non-trivial access method (not `JT_ALL`). Critical for understanding join order quality.
[7]Selectivity = output_rows / examined_rows per table. Values near 0 indicate highly selective filters; near 1 suggests table scans.

9 APPENDIX

9.1 Appendix A. NP-hardness of Workload-Level Query Dispatch

We establish NP-hardness via a reduction from the well–known Partition problem.

Definition 9.1 (Partition). Given $2m$ positive integers $W = \{w_1, \dots, w_{2m}\}$, decide whether there exists a subset $S \subseteq W$ of size m such that

$$\sum_{w \in S} w \;=\; \sum_{w \in W \setminus S} w \;=\; \frac{1}{2}\sum_{j=1}^{2m} w_j. \tag{12}$$

Theorem 9.2 (AQD is NP-hard). *The decision version of the* Workload-Level Dispatch *problem* (1) *is NP-hard even when*

(i) *only a* single *resource dimension is present,*
(ii) *exactly one query arrives in each epoch, and*
(iii) *dispatch to the column engine incurs zero latency.*

Proof. Let $W = \{w_1, \dots, w_{2m}\}$ be an arbitrary instance of the partition problem. We build an AQD instance whose feasible dispatch plans correspond *one-to-one* with balanced partitions of W.

Step 1: **Time horizon and queries.** Set the horizon length to $T = 2m$ and release exactly one query in each epoch $t \in \{1, \dots, T\}$.

Step 2: **Latency parameters.** Dispatching query t to

- the *row* engine ($a_t = 0$) incurs latency $\ell_t(0) = w_t$;
- the *column* engine ($a_t = 1$) incurs latency $\ell_t(1) = 0$.

Step 3: **Resource model.** Use a single resource (e.g. CPU). Each epoch consumes the constant amount

$$r_t = \tfrac{1}{2}, \qquad t = 1, \dots, T, \tag{13}$$

and set the target utilisation to $\gamma_T = 0$. The AQD constraint therefore becomes

$$\frac{1}{T}\sum_{t=1}^{T} r_t \;-\; \frac{1}{T}\sum_{t=1}^{T} \mathbb{1}_{\{a_t=0\}} \;=\; 0, \tag{14}$$

which simplifies to

$$\sum_{t=1}^{T} \mathbb{1}_{\{a_t=0\}} \;=\; m. \tag{15}$$

Hence any feasible schedule must dispatch *exactly m* queries to the row engine.

Step 4: **Latency budget.** Define

$$B \;=\; \frac{1}{2}\sum_{j=1}^{2m} w_j. \tag{16}$$

The decision question is:

Does there exist a dispatch plan that satisfies (15) and achieve total latency $\sum_{t=1}^{T} \ell_t(a_t) = B$?

($\Rightarrow$) Assume the Partition Problem has a solution $S \subseteq \{1, \dots, 2m\}$ with $|S| = m$ and $\sum_{i \in S} w_i = B$. Route the queries indexed by S to the row engine ($a_i = 0$) and the others to the column engine ($a_i = 1$). Equation (15) holds, and the total latency is precisely B, so the AQD instance is feasible.

($\Leftarrow$) Conversely, suppose the AQD instance admits a feasible schedule. Let $S = \{\, i \mid a_i = 0 \,\}$ be the indices sent to the row engine. By (15) we have $|S| = m$, and the latency budget forces $\sum_{i \in S} w_i = B$. Hence S is a balanced partition of W.

Complexity. The reduction runs in $O(T) = O(m)$ time and space, proving that AQD is NP-hard under the stated restrictions. Because a candidate dispatch plan can be verified in polynomial time, the decision version of AQD is in NP and therefore NP-complete. Therefore, the Online Query Dispatch Optimization Problem is NP-hard. □

9.2 Appendix B. Top 32 Features Selected by SHAP Analysis

Table 10 details the top 32 features selected by SHAP analysis.

9.3 Appendix C. Proof Sketch for Theorem 4.1

Sketch. Let $a_t^\star = argmin_a \ell_t(a_t)$ and define $\Delta_t = \ell_t(a_t) - \ell_t(a_t^\star)$.

Table 11: Ablation Study of LightGBM static (↑ = higher is better; ↓ = lower is better).

Variant	avg-rt↓	accuracy↑	macro F_1↑	F_1(hard)↑	overall impr.↑	impr.→optimal↑
full model (baseline)	**2.4298**	**0.8733**	**0.8724**	**0.6460**	**0.4105**	**0.9566**
− no self-paced learning	2.5069	0.8411	0.8407	0.5860	0.3918	0.9130
− no sample weights	2.5258	0.8366	0.8362	0.5761	0.3872	0.9023
− weight A disabled	2.5166	0.8430	0.8425	0.5848	0.3894	0.9075
− weight B disabled	2.5161	0.8429	0.8424	0.5847	0.3895	0.9078
− weight C disabled	2.5166	0.8425	0.8425	0.5889	0.3894	0.9075
− weight D disabled	2.5141	0.8441	0.8437	0.5890	0.3900	0.9089
− weight E disabled	2.4650	0.8502	0.8496	0.6032	0.4019	0.9366
− weight F disabled	2.5258	0.8366	0.8362	0.5761	0.3872	0.9023

Latency regret. Stage 2 uses Linear Thompson Sampling (LinTS). With bounded features ($\|\mathbf{x}_t\|_2 \leq 1$) and rewards ($|\Delta_t| \leq L_{\max}$), LinTS yields $\mathbb{E}\left[\sum_{t=1}^{T} \Delta_t\right] \leq C_1^{'}\sqrt{T \log T}$ for fixed dimension d (Theorem 2 [33]). Stage 3 scales this regret by at most $1/\alpha_t \leq 1/0.48$, giving (10) after division by T.

Resource drift. Stage 3 updates $\boldsymbol{\gamma}_{t+1} = \Pi_{[0,1]^2}(\boldsymbol{\gamma}_t - \eta_t(\mathbf{r}_t - \boldsymbol{\gamma}_t))$ with steps $\eta_t = c/\sqrt{t}$. This is projected OGD on the convex loss $f_t(\boldsymbol{\gamma}) = \frac{1}{2}\|\mathbf{r}_t - \boldsymbol{\gamma}\|_2^2$. The cumulative bound $\sum_t f_t(\boldsymbol{\gamma}_t) \leq O(\sqrt{T})$ (Theorem 1 [54]), which implies (11).

$\square$

Interpretation. The latency regret decreases at $O\left(\sqrt{\log T/T}\right)$, while the resource imbalance decreases at $O(1/\sqrt{T})$. Hence the system becomes progressively faster and more balanced as the workload grows.

9.4 Appendix D. Table of Offline LightGBM Training Ablation Study

Table 11 shows the ablation study results of Offline LightGBM Training using self-paced Taylor-weighted boosting.

REFERENCES

[1] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 967–980.

[2] Yasin Abbasi-Yadkori, Dávid Pál, and Csaba Szepesvári. 2011. Improved algorithms for linear stochastic bandits. *Advances in neural information processing systems* 24 (2011).

[3] Shipra Agrawal and Navin Goyal. 2013. Thompson sampling for contextual bandits with linear payoffs. In *International conference on machine learning*. PMLR, 127–135.

[4] Alibaba Cloud Database Kernel Team. 2023. *Accelerate HTAP Long-Tail Requests and Analyze PolarDB-IMCI Row/Column Fusion.* https://www.alibabacloud.com/blog/about-database-kernel-%7C-accelerate-htap-long-tail-requests-and-analyze-polardb-imci-row-column-fusion_600258 Alibaba Cloud Blog, 4 Aug. 2023.

[5] FANN authors. 2003. FANN: Fast Artificial Neural Network Library—C++ API Reference. https://libfann.github.io/fann/docs/files/fann_cpp-h.html. Version 2.2.0, accessed 2025-07-13.

[6] MySQL HeatWave authors. 2025. *MySQL HeatWave Technical Brief.* Business / Technical Brief. Oracle Corporation. https://www.oracle.com/a/ocom/docs/mysql-heatwave-technical-brief.pdf

[7] Ilaria Battiston, Kriti Kathuria, and Peter Boncz. 2024. OpenIVM: a SQL-to-SQL Compiler for Incremental Computations. In *Companion of the 2024 International Conference on Management of Data*. 516–519.

[8] Beilou. 2022. *400x Faster HTAP Real-time Data Analysis with PolarDB.* Alibaba Cloud ApsaraDB. https://www.alibabacloud.com/blog/598985

[9] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[10] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, et al. 2022. ByteHTAP: bytedance's HTAP system with high data freshness and strong data consistency. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3411–3424.

[11] Adam Dziedzic, Jingjing Wang, Sudipto Das, Bolin Ding, Vivek R Narasayya, and Manoj Syamala. 2018. Columnstore and B+ tree-Are Hybrid Physical Designs Important?. In *Proceedings of the 2018 International Conference on Management of Data*. 177–190.

[12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. *ACM Sigmod Record* 40, 4 (2012), 45–51.

[13] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[14] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. 2017. On calibration of modern neural networks. In *International conference on machine learning*. PMLR, 1321–1330.

[15] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-shot cost models for out-of-the-box learned cost prediction. *arXiv preprint arXiv:2201.00561* (2022).

[16] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[17] J Stuart Hunter. 1986. The exponentially weighted moving average. *Journal of quality technology* 18, 4 (1986), 203–210.

[18] Snowflake Inc. 2022. Introducing Snowflake Unistore. https://www.snowflake.com/blog/introducing-unistore/. Blog post.

[19] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).

[20] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.

[21] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[22] Jens Krueger, Martin Grund, Christian Tinnefeld, Hasso Plattner, Alexander Zeier, and Franz Faerber. 2010. Optimizing write performance for read optimized databases. In *Database Systems for Advanced Applications: 15th International Conference, DASFAA 2010, Tsukuba, Japan, April 1-4, 2010, Proceedings, Part II 15*. Springer, 291–305.

[23] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1253–1258.

[24] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1740–1751.

[25] Tor Lattimore and Csaba Szepesvári. 2020. *Bandit algorithms.* Cambridge University Press.

[26] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. 2010. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*. 661–670.

[27] Wei-Yin Loh. 2011. Classification and regression trees. *Wiley interdisciplinary reviews: data mining and knowledge discovery* 1, 1 (2011), 14–23.

[28] Scott M Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2019. Explainable AI for trees: From local explanations to global understanding. *arXiv preprint arXiv:1905.04610* (2019).

[29] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).

[30] 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*. 1275–1288.

[31] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).

[32] Goeffrey J McLachlan. 1999. Mahalanobis distance. *Resonance* 4, 6 (1999), 20–26.

[33] Gergely Neu, Iuliia Olkhovskaia, Matteo Papini, and Ludovic Schwartz. 2022. Lifting the information ratio: An information-theoretic analysis of thompson sampling for contextual bandits. *Advances in Neural Information Processing Systems* 35 (2022), 9486–9498.

[34] R Malinga Perera, Bastian Oetomo, Benjamin IP Rubinstein, and Renata Borovica-Gajic. 2021. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 600–611.

[35] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, et al. 2022. Cloud-native transactions and analytics in singlestore. In *Proceedings of the 2022 International Conference on Management of Data*. 2340–2352.

[36] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.

[37] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.

[38] Stuart W Roberts. 2000. Control chart tests based on geometric moving averages. *Technometrics* 42, 1 (2000), 97–101.

[39] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.

[40] Daniel J Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, Zheng Wen, et al. 2018. A tutorial on thompson sampling. *Foundations and Trends® in Machine Learning* 11, 1 (2018), 1–96.

[41] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. Lsched: A workload-aware learned query scheduler for analytical database systems. In *Proceedings of the 2022 International Conference on Management of Data*. 1228–1242.

[42] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan Narayanaswamy. 2023. Auto-WLM: Machine learning enhanced workload management in Amazon Redshift. In *Companion of the 2023 International Conference on Management of Data*. 225–237.

[43] Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. 2024. Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3290–3303.

[44] Haoze Song, Wenchao Zhou, Feifei Li, Xiang Peng, and Heming Cui. 2023. Rethink query optimization in htap databases. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[45] StarRocks Contributors. 2025. StarRocks: An Open-Source, High-Performance Analytical Database. https://starrocks.io/. Accessed 2025-08-28.

[46] William R Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3/4 (1933), 285–294.

[47] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, et al. 2023. Polardb-imci: A cloud-native htap database system at alibaba. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[48] Yang Wu, Xuanhe Zhou, Xiaoguang Li, Jinhuai Kang, Chunxiao Xing, Tongliang Li, Xinjun Yang, Wenchao Zhou, Feifei Li, and Yong Zhang. 2025. MemQ: A Graph-Based Query Memory Prediction Framework for Effective Workload Scheduling. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 3876–3889.

[49] Yang Wu, Xuanhe Zhou, Yong Zhang, and Guoliang Li. 2024. Automatic index tuning: A survey. *IEEE Transactions on Knowledge and Data Engineering* 36, 12 (2024), 7657–7676.

[50] Chao Zhang, Guoliang Li, and Tao Lv. 2024. HyBench: A new benchmark for HTAP databases. *Proceedings of the VLDB Endowment* 17, 5 (2024), 939–951.

[51] Chao Zhang, Guoliang Li, Jintao Zhang, Xinning Zhang, and Jianhua Feng. 2024. HTAP databases: A survey. *IEEE Transactions on Knowledge and Data Engineering* (2024).

[52] Chi Zhang, Ryan Marcus, Anat Kleiman, and Olga Papaemmanouil. 2020. Buffer pool aware query scheduling via deep reinforcement learning. *arXiv preprint arXiv:2007.10568* (2020).

[53] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2020. FLAT: fast, lightweight and accurate method for cardinality estimation. *arXiv preprint arXiv:2011.09022* (2020).

[54] Martin Zinkevich. 2003. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the 20th international conference on machine learning (icml-03)*. 928–936.