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
zhouxuanhe@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 over 90% compared to cost-threshold dispatching and improves HyBench score by 15% over the cost-threshold method and 9% over the current SOTA BRAD.

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, 19(7): 1586 - 1599, 2026.
doi:10.14778/3801059.3801071

PVLDB Artifact Availability:
The source code, data, and/or other artifacts 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 [14, 17, 21, 29, 50, 55]. 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 [21, 55].

HTAP systems can be broadly classified into two categories: *single-engine* and *dual-engine* architectures. *Single-engine* systems like Oracle In-Memory [28], SQL Server Columnstore [29], SAP HANA [17], HyPer [25], and StarRocks [48] 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 [47].

In contrast, *dual-engine* systems have gained widespread adoption in production environments, including TiDB paired with TiFlash [21], PolarDB [50], MySQL HeatWave [10], and PostgreSQL paired with DuckDB [11, 39]. 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.

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. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801071

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, 45]. 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 [14, 29, 50] or hand-tuned heuristics [17]. 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] [8]. 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 [34, 35, 51, 58], 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 [24] 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 [36] scores to detect CPU/memory imbalances between engines. These scores are weighted based on system load estimated [22, 40]: 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 [50] 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 [32, 33]. 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) Experimental results show that AQD reduces average query latency by over 90% compared to cost-threshold dispatching when concurrently running randomly generated queries. AQD achieves the best HyBench score of 9.56, 15% improvement over the cost-threshold method and 9% improvement over the current SOTA BRAD [53]. Multi-phase workload execution demonstrates AQD's adaptation to workload drifts and resource efficiency.

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

[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).

production results. Section 6 analyses insights, limitations, and future work; Section 7 summarizes related work. Section 8 concludes the paper.

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 with feature vector $\mathbf{x} \in \mathbb{R}^d$ extracted from the query optimizer, we learn a dispatch 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, $a = 1$ the column engine, and $\ell_a(q)$ denoting execution latency on engine a. Incorrect dispatches incur regret $r(q, a) = w(q) \cdot \mathbf{1}[a \neq z]$, where $w(q) = |\ell_{\text{row}}(q) - \ell_{\text{col}}(q)|$ is the latency gap and $z = \mathbf{1}[\ell_{\text{col}} < \ell_{\text{row}}]$ indicates the optimal engine.

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. Define the average resource consumption and routing fraction for the row engine: $\bar{\mathbf{r}} = \frac{1}{T}\sum_{i=1}^{T} \mathbf{r}_i, \quad \bar{a} = \frac{1}{T}\sum_{i=1}^{T} \mathbb{1}_{\{a_i=0\}}$ 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, and zero represents perfect proportionality.

Constraint (1) sets this bias to target $\boldsymbol{\gamma}_T$. Common settings: $\boldsymbol{\gamma}_T = (0, 0)$ for balanced utilization; slightly positive to reserve row engine capacity for OLTP bursts; slightly negative to bias toward the 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 [32, 33] 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 [24] 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 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

[3] In the full version of this paper.

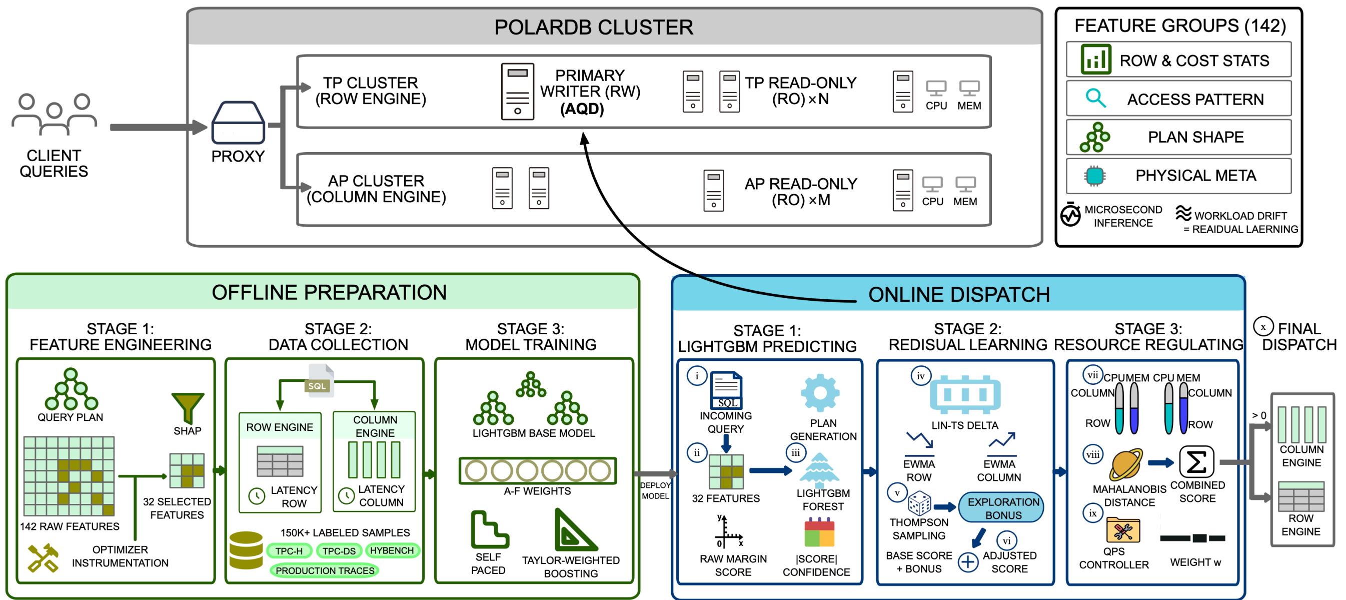

Figure 1: Framework of AQD

adjusted dispatch score that balances exploitation of known patterns with exploration of uncertain cases[4].

(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.

2.3 Integration with PolarDB Architecture

The entire PolarDB cluster contains a proxy, a TP cluster, and an AP cluster. The TP cluster contains a primary writer node (RW) where AQD works on and many normal read-only node (RO). The AP cluster contains many AP RO nodes. The proxy will at first forwards all queries to the primary writer node in the TP cluster, where AQD observes optimizer features and heartbeat telemtry from other nodes (CPU, memory, etc.). AQD then decides whether TP cluster (row engine) or AP cluster (column engine) will execute the query faster. The query will be rerouted to the proxy and the proxy will decide specifically which node in the corresponding cluster the query will be routed to based on load balancing policy.

[4]We choose residual learning other than fully retraining LightGBM becuase residual learning is lightweight and real-time responsive. Residual learning does not require additional dual execution on both engines to collect more training data

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.

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

Extending existing research on query plan representation learning [57], 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 as shown in Table 2.

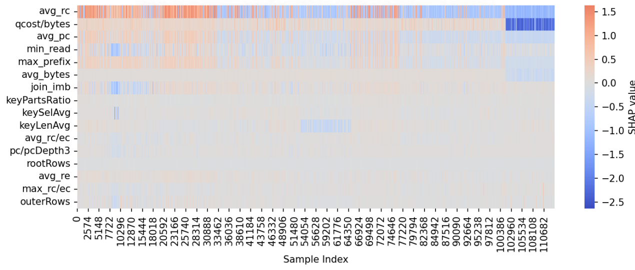

Figure 2: SHAP value heatmap: 16 of the selected 32 features. Darker colors indicate higher feature importance. Features are ordered by average importance.

To reduce computational overhead, we employ SHAP (SHapley Additive exPlanations) analysis [32, 33] during offline training (Section 3.3) to identify the most predictive features. We retain the top-32 features with highest SHAP values, achieving a 77% reduction in dimensionality with negligible no impact on prediction accuracy[5].

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[6].

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 (8 tables) and TPC-DS (26 tables) at scale factors 1, 10 and 100, and HyBench (8 tables) at scale factors of 1 and 10. To keep the mix realistic we added seven production traces [20]: Airline (19 tables), Credit (8 tables), Carcinogenesis (6 tables), Employee (6 tables), Financial (8 tables), Geneea (19 tables), Hepatitis (7 tables).

We generate query workloads by emulating real-world query patterns [20]. The workloads include queries with random joins adhering to schema definitions, predicates sampled from underlying data distributions, and complex logical expressions involving AND and OR operators. 97% of the queries contain `WHERE` predicates (majority with 1–9 predicates), 72% use aggregates (`COUNT/SUM/AVG/MIN/MAX`), 64% involve joins (0–9 joins with uniform distribution across 1–7 joins), and 53% use `GROUP BY/ORDER BY` clauses. Query complexity spans from simple single-table point queries (36%) to complex multi-table analytical queries (64% with 2–10 tables). 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.

3.3 Model Training

3.3.1 Model Selection. We evaluate four candidate models for query dispatch: CART (single decision tree) [31], Random Forest [13], feed-forward neural networks (FNN) [9, 41], and LightGBM [24]. 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 probability $p \in [0, 1]$ 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

[5]We also compare prediction results between PCA and SHAP. All 140 features have an average accuracy of 0.816, SHAP-top-32 has an average accuracy of 0.81, SHAP-top-16 has an average accuracy of 0.80, PCA that extracts 26 features (with variance 95%) has an average accuracy of 0.79. Therefore, we choose the top 32 features by SHAP analysis

[6]In the full version of this paper.

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.

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 [19]. 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 completes, we observe only the latency ℓ_t of the chosen engine (row if $a_t = 0$, column if $a_t = 1$), creating a fundamental challenge: how can we learn whether the other engine would have been faster without running the query twice? To address this counterfactual estimation problem, we develop two complementary estimators for the unobserved latency.

The first estimator, the **Exponentially Weighted Moving Average (EWMA)**, maintains for each engine a lightweight running latency estimate[7] yielding counterfactual estimates $\hat{\ell}^{\text{row}}$ and $\hat{\ell}^{\text{col}}$. The second estimator, the **Doubly Robust (DR)** method [15], combines inverse-propensity weighting with outcome regression using query-feature-based LightGBM predictors for latencies of two engines. The experiments (Figure 5 in Section 5.5.1) show that EWMA achieves comparable or better accuracy with substantially lower computational cost and variance under high-frequency, small-sample conditions. We therefore adopt EWMA as the default estimator in AQD's online learning module. We construct a signed residual Δ_t that is positive when the column engine is expected to be faster, with logarithmic transformation for numerical stability:

$$\Delta_t = \log(1 + \tilde{\ell}_{\text{row}}(t)) - \log(1 + \tilde{\ell}_{\text{col}}(t)) \tag{2}$$

where $\tilde{\ell}_e(t) = \ell_e(t)$ if engine e was chosen (observed latency), otherwise $\tilde{\ell}_e(t) = \hat{\ell}_e(t)$ (EWMA estimate). This unified formulation ensures consistent residual semantics: positive values indicate the row engine is slower (column is better), while negative values indicate the row engine is faster (row is better).

We cast online adaptation as a linear contextual bandit [2, 30] that learns to refine base LightGBM predictions using streaming residual observations. We model $\mathbb{E}[\Delta_t \mid \mathbf{x}_t] = \mathbf{x}_t^\top \boldsymbol{\theta}^\star$, where $\boldsymbol{\theta}^\star$ captures how query features correlate with prediction errors. Our LinTS-Delta algorithm employs Thompson Sampling [42, 49] to balance exploration and exploitation by maintaining a Bayesian posterior over $\boldsymbol{\theta}$ with regularization $\lambda = 0.5$: $V_t = \lambda I + \sum_{\tau=1}^{t} \mathbf{x}_\tau \mathbf{x}_\tau^\top$ and $\mathbf{b}_t = \sum_{\tau=1}^{t} \mathbf{x}_\tau \Delta_\tau$. The posterior mean is $\hat{\boldsymbol{\theta}}_t = V_t^{-1}\mathbf{b}_t$ with covariance V_t^{-1}. At each step, we sample $\tilde{\boldsymbol{\theta}}_t \sim \mathcal{N}(\hat{\boldsymbol{\theta}}_{t-1}, \sigma^2 V_{t-1}^{-1})$ and compute Thompson score $u_t = \mathbf{x}_t^\top \tilde{\boldsymbol{\theta}}_t$, where the covariance matrix quantifies uncertainty—high uncertainty induces exploration through diverse samples, while low uncertainty leads to exploitation. We then combine the static LightGBM prediction s_t with the learned residual through $z_t = \tanh(s_t + u_t) \in (-1, 1)$, allowing Thompson Sampling to refine rather than replace the base model. The tanh function ensures bounded outputs suitable for downstream resource-aware dispatch.

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 utilization from its target be $\mathbf{e}_t = \mathbf{r}_t - \boldsymbol{\gamma}_t = (e_t^{\text{cpu}}, e_t^{\text{mem}})^\top \in \mathbb{R}^2$, where $\boldsymbol{\gamma}_t = (\gamma_t^{\text{cpu}}, \gamma_t^{\text{mem}})$ represents the *target resource share* that the dispatcher aims to maintain (e.g., γ_t^{cpu}=γ_t^{mem}=0.5 indicates balanced usage). We compute the empirical covariance matrix Σ_t from the K most recent deviations and measure the imbalance magnitude using the Mahalanobis distance [36]:

$$d_t = \text{sgn}(e_t^{\text{cpu}}) \sqrt{\mathbf{e}_t^\top \Sigma_t^{-1} \mathbf{e}_t} \tag{3}$$

Finally we squash the signed distance to $[-1, 1]$ via $r_t = \tanh(d_t)$, where positive values indicate the row engine is over-utilizing resources relative to its target ($e_t^{\text{cpu}} > 0$), while negative values indicate under-utilization.

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

[7] $\hat{\ell}^{(e)} \leftarrow \alpha \ell_e + (1 - \alpha)\hat{\ell}^{(e)}, \quad \alpha = 0.03$

two scores with load-adaptive coefficient[8]:

$$s_t^{\text{final}} = \omega_t z_t + (1-\omega_t) r_t, \quad \omega_t = \omega_{\min} + \Delta\omega \cdot \sigma(\hat{\lambda}_t/\lambda_0 - b), \quad (4)$$

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

- Low load (QPS ≈ 0) $\Rightarrow \omega_t \approx \omega_{\min} \rightarrow$ prioritize resource parity.
- High load (QPS $\rightarrow +\infty$) $\Rightarrow \omega_t \approx \omega_{\min} + \Delta\omega \rightarrow$ prioritize latency.

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

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) [59] step on the target share:

$$\boldsymbol{\gamma}_{t+1} = \Pi_{[0,1]^2}(\boldsymbol{\gamma}_t - \beta g_t), \qquad \beta = \frac{c}{\sqrt{t}} \ (c > 0). \qquad (5)$$

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 $s_t^{\text{final}} = \omega_t z_t + (1-\omega_t) r_t$ uses load-adaptive weighting ω_t (Equation 4), 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).

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.

[8]In our implementation: $\omega_{\min} = 0.4$, $\Delta\omega = 0.3$, $\lambda_0 = 5$, $b = 1$.

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

Require: LightGBM model f (offline–trained); LinTS hyper-parameters (λ, σ^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: $\Sigma_t \leftarrow \text{COV}(\mathcal{B})$
19: *lastCov*←Now
20: **end if**
21: $d_t \leftarrow \text{sgn}(e_t^{\text{cpu}})\sqrt{\mathbf{e}_t^\top \Sigma_t^{-1} \mathbf{e}_t}$
22: $r_t \leftarrow \tanh(d_t)$
23: $\omega_t \leftarrow \omega_{\min} + \Delta\omega \cdot \sigma(\hat{\lambda}_t/\lambda_0 - b)$
24: $s_t^{\text{final}} \leftarrow \omega_t z_t + (1-\omega_t) r_t$
25: $a_t \leftarrow \mathbf{1}\left[s_t^{\text{final}} > 0\right]$ ▷ 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: $\tilde{\ell}_{\text{row}} \leftarrow \begin{cases} \hat{\ell}^{\text{row}} & \text{if } a_t = 1 \\ \ell_{a_t} & \text{if } a_t = 0 \end{cases}$, $\tilde{\ell}_{\text{col}} \leftarrow \begin{cases} \ell_{a_t} & \text{if } a_t = 1 \\ \hat{\ell}^{\text{col}} & \text{if } a_t = 0 \end{cases}$
29: $\Delta_t \leftarrow \log(1+\tilde{\ell}_{\text{row}}) - \log(1+\tilde{\ell}_{\text{col}})$ ▷ residual (Eq. 2)
30: $V \leftarrow V + \mathbf{x}_t \mathbf{x}_t^\top$; $\mathbf{b} \leftarrow \mathbf{b} + \mathbf{x}_t \Delta_t$
31: **if** Now − *lastOCO* $\geq S$ **then**
32: $\beta_t \leftarrow c/\sqrt{t}$
33: $\boldsymbol{\gamma}_{t+1} \leftarrow \Pi_{[0,1]^2}\left[\boldsymbol{\gamma}_t - \beta_t(\mathbf{r}_t - \boldsymbol{\gamma}_t)\right]$
34: *lastOCO*←Now
35: **end if**
36: **end for**

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 [24]. The trained model (approximately 10 MB) is loaded by the engine. Feature extraction and inference combined add about 500 μs overhead per query.

5.2 Baselines

We compare two variants of AQD (LightGBM Static and LightGBM Dynamic) against five 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[9].
(4) **Hybrid Optimizer**: Uses linear regression to map optimizer costs to actual runtimes, then selects the engine with lower predicted latency [12].
(5) **BRAD** [53]: A decision-forest-based query router that automatically selects between row-optimized and column-optimized engines. We implement BRAD's routing component by training its decision forest model on the same workload traces used for our offline preparation.

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 (≥ 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 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 Table 6.

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.

5.5.1 Multi-phase Workload Drift Experiments. To comprehensively evaluate online adaptation under diverse workload conditions, we design seven benchmark phases that systematically test different aspects of query dispatch: OLAP saturation (150 QPS with TPC-H/TPC-DS sf10), OLTP burst (200 QPS with transactional workloads), workload shift (75% OLAP queries), scale drift (mixing sf1, sf10, and sf100), mixed contention (simultaneous OLAP+OLTP at 120 QPS), memory pressure (sf100 datasets), and OLTP dominance (180 QPS row-heavy workload).

[9] In practice, this rule works in conjunction with PolarDB's resource controls: the row engine's thread-pool admission mechanism and the column engine's concurrency and DOP limits [4–7].

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

Metric ↑/↓	Dataset														
	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. In the second and third part of the table, the metrics are related to LightGBM model.

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

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 ↑	**0.91**	0.78	0.42	0.49
Macro F_1 ↑	**0.91**	0.78	0.30	0.43
F_1 (hard) ↑	**0.72**	0.46	0.12	0.27
Overall impr. ↑	**41.5%**	30.5%	0.5%	−2.1%
Impr. → optimal ↑	**96.8%**	71.1%	1.3%	−4.8%

Figure 3 compares 6 dispatch methods, specifically demonstrating the similar performance of EWMA and Doubly-Robust estimation. The figure also shows our LightGBM-based dispatch methods outperform BRAD, cost threshold rule, and hybird optimizer especially in phase 6. We adopt EWMA as our default counterfactual estimator in LightGBM Dynamic in later experiments as it's simpler and has slightly better performance than Doubly-Robust.

To visualize the online learning dynamics, Figure 4 plots the residual evolution throughout the benchmark execution. The immediate residuals (light blue) show the raw learning signal at each query, while the smoothed curve (orange, computed via exponential moving average) reveals the system's adaptation pattern. Among queries 0-1500, there is initial negative bias. Then comes a transition period with elevated positive residuals, stabilization near zero between query 3000-4000, and another fluctuation after query 4000.

Figure 5 shows resource utilization across multi-phase workload execution. LightGBM-based approaches (`lightgbm_dynamic_ewma`, `lightgbm_dynamic_dr`) maintain 2–3% total CPU usage with balanced load across engines (1–2% each), while traditional methods (`cost_thresh`, `hybrid_opt`, `BRAD`) consume 6–8% CPU with larger spikes. LightGBM methods use 85–115GB memory versus 20–45GB for cost-threshold baseline, reflecting efficient column engine utilization for analytical queries (lower CPU, higher memory).

5.5.2 Performance Comparison Across 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 [50], 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 7 shows that learning-based dispatch consistently outperforms traditional baselines. Relative to Cost Threshold, LightGBM Dynamic reduces makespan by 33.0–68.5%, average latency by 92.0–98.4%, and P95 latency by 97.3–99.0% across all concurrency levels. Comparing dynamic to its static counterpart, the makespan gains are 9.9%, 21.2%, and 1.6% at 200, 400, and 600 queries, respectively, peaking at 400 queries where online adaptation best handles emerging contention. Under higher concurrency, dynamic delivers a 44.0% makespan reduction at 800 queries and a 5.7% reduction at 1000 queries. At 1000 queries, dynamic achieves

Table 6: Ablation Study of Offline LightGBM Training (↑ = 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

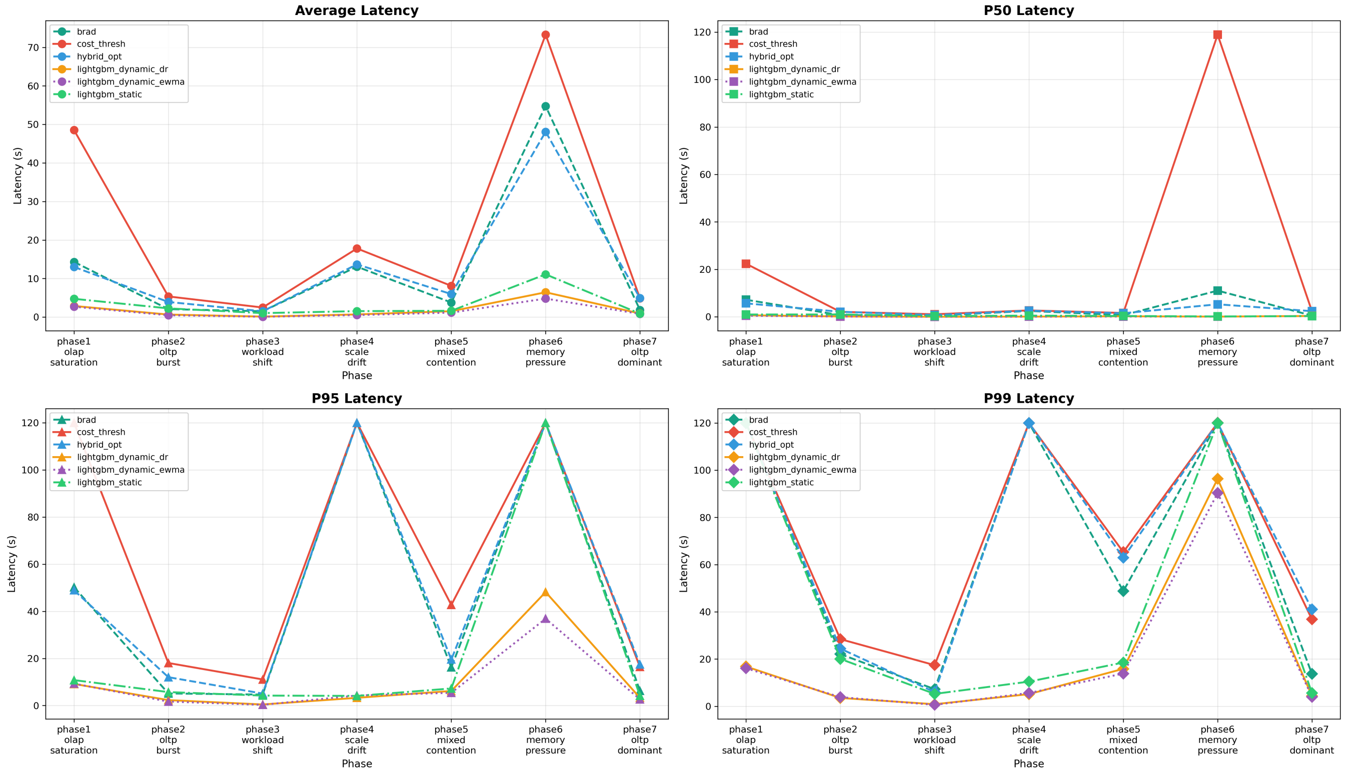

Figure 3: Comparison between 6 Dispatch Methods over 7 Workload Phases

17.9% lower average latency than static (0.69s vs 0.84s) with comparable P95 latency (1.38s vs 1.36s); versus traditional methods, it yields 98.2% lower average latency and 98.9% lower P95 latency. These results indicate that LightGBM Dynamic's online adaptation provides robust, workload-aware performance across a wide range of operating points.

Table 8 compares five dispatch strategies on the HTAP benchmark `hybench` [54]. Our LightGBM-based methods (`LightGBM Static` and `LightGBM Dynamic`) achieve the highest overall HTAP scores, with `LightGBM Dynamic` best at 9.56 and `LightGBM Static` at 9.14, outperforming the current SOTA method (`BRAD`, 8.77) by 9.0% and 4.2%, respectively. By metric, `LightGBM Dynamic` leads AP-QPS (1.87, +9.4% vs. the best baseline at 1.71) and XP-TPS (5.68, +5.8% vs. the best baseline at 5.37), `LightGBM Static` leads TP-TPS (64.67), and `BRAD` leads XP-QPS (3.37). These results indicate that `LightGBM Dynamic` best balances analytical and mixed workloads to maximize overall HTAP performance, while `LightGBM Static` excels in pure transactional throughput.

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 architecture balances predictive accuracy with runtime adaptability through complementary phases.

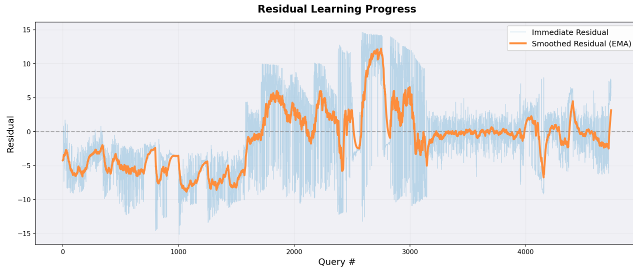

Figure 4: Residual learning progress during workload execution.

Table 7: Performance Comparison of Dispatch Strategies under Varying Concurrency Levels.

Dispatch Strategy	Makespan (s)↓	Avg Latency (s)↓	P95 Latency (s)↓
200 Concurrent Queries			
Cost Threshold	118.20	6.62	41.43
Hybrid Optimizer	99.54	5.40	33.53
BRAD	100.19	4.32	33.31
LightGBM Static	48.44	0.90	2.79
LightGBM Dynamic	**43.66**	**0.53**	**1.05**
Improvement	63.06%	92.00%	97.47%
400 Concurrent Queries			
Cost Threshold	137.86	11.09	56.72
Hybrid Optimizer	134.40	15.22	63.47
BRAD	134.75	14.13	51.02
LightGBM Static	55.13	0.69	3.09
LightGBM Dynamic	**43.42**	**0.45**	**1.54**
Improvement	68.50%	95.94%	97.28%
600 Concurrent Queries			
Cost Threshold	149.58	17.63	74.13
Hybrid Optimizer	150.04	26.22	81.51
BRAD	149.85	22.36	56.53
LightGBM Static	52.46	0.41	0.94
LightGBM Dynamic	**51.60**	**0.33**	**0.89**
Improvement	65.50%	98.13%	98.80%
800 Concurrent Queries			
Cost Threshold	161.44	34.06	120.00
Hybrid Optimizer	161.41	36.49	104.54
BRAD	161.53	43.25	113.89
LightGBM Static	122.19	1.21	2.25
LightGBM Dynamic	**68.44**	**0.54**	**1.15**
Improvement	57.61%	98.41%	99.04%
1000 Concurrent Queries			
Cost Threshold	172.04	37.28	120.00
Hybrid Optimizer	172.08	40.11	120.00
BRAD	166.73	51.32	120.00
LightGBM Static	122.19	0.84	**1.36**
LightGBM Dynamic	**115.20**	**0.69**	1.38
Improvement	33.04%	98.15%	98.85%

Note: Bold values indicate best performance for each metric. Improvement is relevant to LightGBM Dynamic with Cost Threshold.

Offline, we distill 142 features into 32 SHAP-selected signals and train a 10MB LightGBM ensemble with regret-weighted boosting, capturing stable query-to-engine mappings that achieve up to 87%

Table 8: Hybench Benchmark Results for Different Dispatch Strategies.

Method	AP-QPS ↑	TP-TPS ↑	XP-QPS ↑	XP-TPS ↑	HTAP-Score ↑
Cost Threshold	1.71	41.87	2.65	5.37	8.31
Hybrid Optimizer	1.71	42.65	2.87	4.97	8.30
BRAD	1.59	55.87	**3.37**	4.22	8.77
LightGBM Static	1.62	**64.67**	2.53	4.75	9.14
LightGBM Dynamic	**1.87**	56.95	2.52	**5.68**	**9.56**

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.

of optimal performance on isolated queries. Online, LinTS-Delta learns residual prediction errors while the OCO-based regulator maintains resource balance via Mahalanobis distance scoring, reducing makespan by up to 52% vs. static dispatch at high concurrency (Table 7). This separation — where offline learning captures query-intrinsic properties (selectivity, join patterns) and online adaptation handles system-extrinsic factors (load, resource availability) — enables sub-millisecond decisions with theoretical guarantees on 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 makes binary engine choices per query. Operator-level dispatch could improve performance for queries with mixed OLTP/OLAP characteristics.

(2) **Model Maintenance.** While our system includes mechanisms for online adaptation, long-term model drift remains a challenge. We plan periodic retraining triggered by validation regret thresholds, combined with incremental learning during low-utilization periods (selectively re-executing queries on both engines for fresh labels via mini-batch gradient boosting).

(3) **Generalization across Systems.** AQD's core principles (feature engineering, regret-weighted training, bandit learning) apply to other dual-engine HTAP systems like TiDB, MySQL HeatWave, Postgres-DuckDB, and ShannonBase [46]. Only feature extraction requires system-specific adaptation.

7 RELATED WORK

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

HTAP Architectures. HTAP systems [47, 55] can be classified into: (i) Single-Engine Systems. These maintain both row and column formats within one engine. SAP HANA [18] pioneered unified storage with per-operator format selection. Oracle Database In-Memory [28] and SQL Server Columnstore [16, 29] add columnar capabilities to row engines. HyPer [25] uses fork-based snapshots for isolation. StarRocks [48] 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 [21] uses Raft-based replication, ByteHTAP [14] achieves sub-second freshness, and PolarDB [50] introduces row-column fusion operators. SingleStore [38] and Snowflake Unistore [23] follow similar patterns.

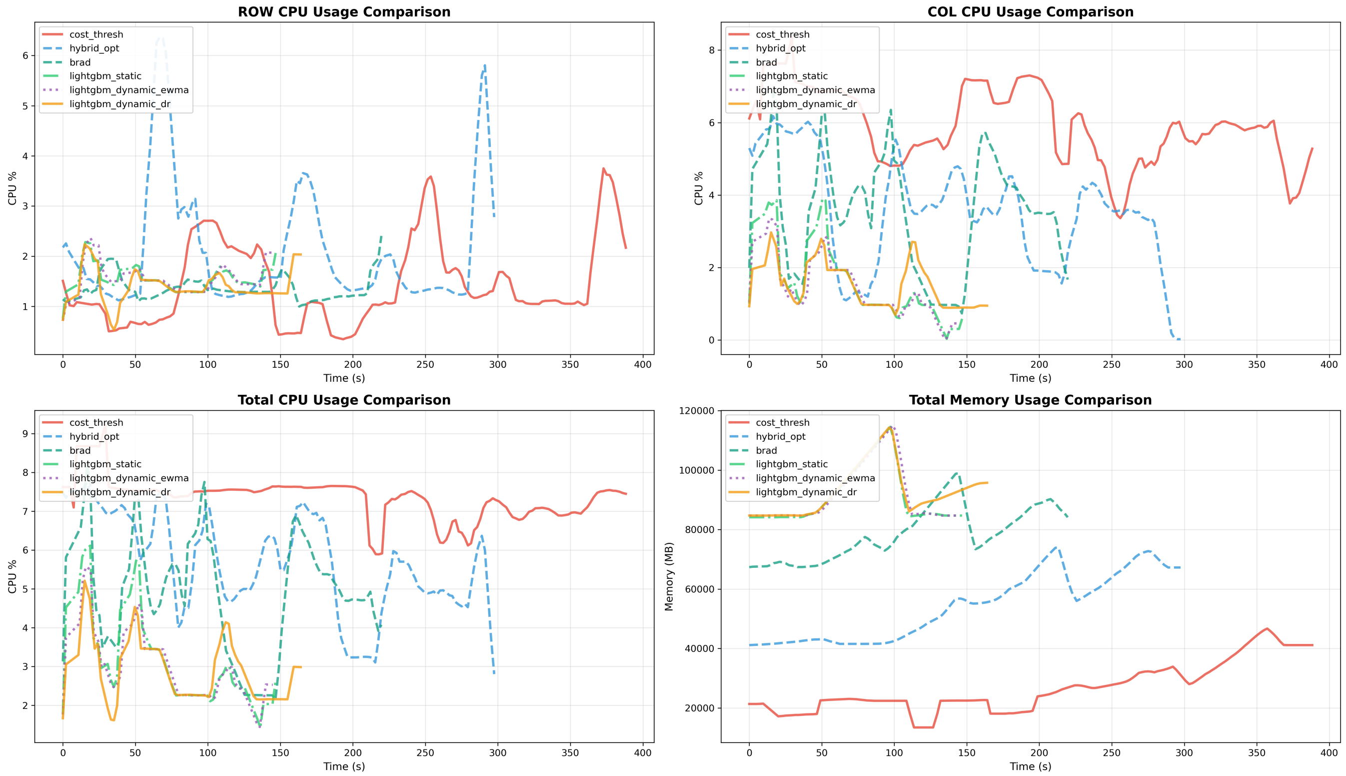

Figure 5: Resource utilization during workload execution across multiple phases.

Query Dispatch Strategies. Plan-level dispatch, exemplified by MySQL HeatWave [10], dispatches entire queries to a single engine, balancing flexibility with implementation simplicity. Operator-level systems (including SQL Server [29], Oracle In-Memory [28], SAP HANA [27], TiDB [21], and PolarDB [50]) enable fine-grained per-operator engine selection, maximizing performance potential at the cost of increased optimization complexity.

Most HTAP databases employ cost models [29, 50] or manually tuned heuristics [17]. Recent work explores machine learning: ByteHTAP [14] mentions that it applies tree-CNNs for plan-level dispatch decisions but does not provide experimental results, while BRAD [53] uses decision-forest based model to route queries between row-optimized engine and column-optimized engine. 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 [34, 35], while Wu et al. apply graph neural networks for query memory prediction [51]. Online learning approaches include deep RL for join ordering [26] and multi-armed bandits for index selection [37, 52]. Resource-aware schedulers such as Auto-WLM [44], LSched [43], and buffer-sensitive RL [56] 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

We present AQD, an online adaptive query dispatcher for dual-engine HTAP databases that combines offline LightGBM training with online LinTS-Delta residual learning and resource-aware regulation. Evaluation on PolarDB shows AQD reduces query latency by over 90% compared to cost-threshold dispatching and improves HyBench score by 15% over the cost-threshold method and 9% over the current SOTA BRAD, with sub-millisecond dispatch overhead suitable for production deployment. AQD demonstrates that learned query dispatch with online adaptation is effective for real-world HTAP systems.

ACKNOWLEDGMENTS

Yong Zhang and Xuanhe Zhou are co-corresponding authors. This work was supported in part by National Key RD Program of China ("Research on Collaborative Intelligent Construction Technology and Application of Ecological Environment Knowledge Base for the Beijing-Tianjin-Hebei Region"), NSF of China (62502304, U25A20437), Alibaba Research Intern Program, Beijing National Research Center for Information Science and Technology (BNRist), Shanghai Jiao Tong University AI for Engineering Initiative, Shanghai Artificial Intelligence Laboratory.

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. 2024. Concurrency Control - PolarDB for MySQL User Guide. https://www.alibabacloud.com/help/en/polardb/polardb-for-mysql/user-guide/concurrency-control. Accessed: 2025-10-17.

[5] Alibaba Cloud. 2024. Technical Architecture of IMCI - PolarDB for MySQL. https://www.alibabacloud.com/help/en/polardb/polardb-for-mysql/user-guide/technical-background-and-architecture-of-column-store-index. Accessed: 2025-10-17.

[6] Alibaba Cloud. 2025. IMCI in PolarDB for MySQL - Overview. https://www.alibabacloud.com/help/en/polardb/polardb-for-mysql/user-guide/overview-29. Accessed: 2025-10-17.

[7] Alibaba Cloud. 2025. Thread Pool - PolarDB for MySQL User Guide. https://www.alibabacloud.com/help/en/polardb/polardb-for-mysql/user-guide/thread-pool. Accessed: 2025-10-17.

[8] 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 Accessed: 2025-10-17.

[9] 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.

[10] 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 Accessed: 2025-10-17.

[11] 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.

[12] Beilou. 2022. *400x Faster HTAP Real-time Data Analysis with PolarDB.* Alibaba Cloud ApsaraDB. https://www.alibabacloud.com/blog/598985 Accessed: 2025-10-17.

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

[14] 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.

[15] Miroslav Dudík, Dumitru Erhan, John Langford, and Lihong Li. 2014. Doubly robust policy evaluation and optimization. In *Statistical Science*, Vol. 29. Institute of Mathematical Statistics, 485–511.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-shot cost models for out-of-the-box learned cost prediction. *Proc. VLDB Endow.* 15, 11 (July 2022), 2361–2374.

[21] 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.

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

[23] Snowflake Inc. 2022. Introducing Snowflake Unistore. https://www.snowflake.com/blog/introducing-unistore/. Accessed: 2025-10-17.

[24] 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).

[25] 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.

[26] 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).

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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).

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

[34] 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.

[35] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: a learned query optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718.

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

[37] 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.

[38] 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.

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

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

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] Shannon Data. [n.d.]. ShannonBase: An Open Source MySQL Branch with Native Support for Rapid Column Store. https://github.com/Shannon-Data/ShannonBase. Accessed: 2025-10-28.

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] Geoffrey X Yu, Ziniu Wu, Ferdi Kossmann, Tianyu Li, Markos Markakis, Amadou Ngom, Samuel Madden, and Tim Kraska. 2024. Blueprinting the Cloud: Unifying and Automatically Optimizing Cloud Data Infrastructures with BRAD. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3629–3642.

[54] 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.

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

[56] 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).

[57] Yue Zhao, Zhaodonghui Li, and Gao Cong. 2023. A comparative study and component analysis of query plan representation techniques in ML4DB studies. *Proceedings of the VLDB Endowment* 17, 4 (2023), 823–835.

[58] 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).

[59] Martin Zinkevich. 2003. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning* (Washington, DC, USA) *(ICML'03)*. AAAI Press, 928–935.