International Journal of Technology, Management and Humanities Volume 12, Issue 1, 2026

Adaptive Query Optimization in Kubernetes Orchestrated Open Source Relational Databases Using Telemetry Driven Feedback Loops

Vinodkrishna Gopalan*

The Ohio State University, USA

Abstract

Relational query optimization has traditionally relied on static cost models built upon cardinality estimation and predefined access path selection strategies. While adaptive query processing techniques have improved robustness against data skew and runtime uncertainty, most open-source relational database systems remain largely unaware of infrastructure-level variability introduced by cloud-native orchestration platforms. In Kubernetes-managed environments, elastic scaling, pod migration, and multi-tenant resource contention frequently invalidate optimizer assumptions, resulting in suboptimal execution plans and performance instability.

This study proposes a telemetry-driven adaptive query optimization framework for Kubernetes-orchestrated open-source relational databases. The proposed architecture integrates fine-grained query execution metrics, cluster-level resource telemetry, and feedback control mechanisms to continuously refine cost models and selectively trigger re-optimization during runtime. By establishing a cross-layer communication channel between the database engine and the orchestration layer, the system dynamically responds to workload fluctuations and resource elasticity.

Experimental evaluation under variable workload intensities and scaling scenarios demonstrates significant reductions in query latency variance, improved cardinality estimation accuracy, and enhanced cluster resource utilization efficiency. The results indicate that telemetry-guided feedback loops provide a practical pathway toward autonomic, cloud-aware relational database systems. This work advances the integration of adaptive query processing with containerized infrastructure management and contributes toward the realization of fully self-optimizing, Kubernetes-native database architectures.

Keywords: Adaptive Query Optimization; Kubernetes; Telemetry-Driven Feedback; Cloud-Native Databases; Self-Driving Database Systems; Learned Query Optimization.

International Journal of Technology, Management and Humanities (2026) Doi: 10.21590/ijtmh.12.01.08

Introduction

Background and Motivation

The discipline of relational query optimization traces its formal foundations to the System R project at IBM, where cost-based optimization was first systematically introduced (Selinger et al., 1979). The System R optimizer established the principle that query plans should be selected by minimizing an estimated execution cost derived from cardinality estimates, selectivity statistics, and access path alternatives. Its dynamic programming enumeration strategy and cost model assumptions shaped the architecture of nearly all subsequent relational database management systems.

Following System R, classical relational optimizer architectures matured into modular pipelines composed of parsing, logical plan generation, rule based rewriting, physical plan enumeration, and cost estimation (Chaudhuri, 1998; Ioannidis, 1996). Cost based optimization relies heavily on statistical summaries such as histograms and selectivity models. Although these techniques proved effective for stable workloads and centralized deployments, they depend on assumptions of predictable resource availability and accurate cardinality estimation. Ioannidis (1996) emphasized that estimation errors propagate multiplicatively across join trees, often leading to severe performance degradation.

To manage the combinatorial explosion of plan alternatives, frameworks such as Cascades formalized rule-driven plan exploration and transformation (Graefe, 1995). Cascades introduced a unified architecture for logical and physical

Corresponding Author: Vinodkrishna Gopalan, The Ohio State University, USA, e-mail: gopalanv@gmail.com

How to cite this article: Gopalan, V. (2026). Adaptive Query Optimization in Kubernetes Orchestrated Open Source Relational Databases Using Telemetry Driven Feedback Loops. *International Journal of Technology, Management and Humanities,* 12(1), 77-91.

Source of support: Nil

Conflict of interest: None

transformations, enabling extensible rule systems and cost-based pruning. This model became foundational for commercial and open-source optimizers alike.

Despite their robustness, classical cost-based optimizers were designed for relatively static execution environments. They assume stable CPU availability, predictable memory allocation, and limited runtime interference. In modern cloud-native deployments, particularly those orchestrated by container schedulers, these assumptions no longer hold. Resource elasticity, noisy neighbors, pod migration, and dynamic scaling introduce runtime variability that is invisible to the traditional optimizer. As a result, execution plans chosen at compile-time may become suboptimal during execution due to resource contention or scaling events.

The increasing deployment of open-source relational databases inside containerized clusters therefore exposes a fundamental mismatch between static cost models and dynamic infrastructure behavior. Addressing this mismatch motivates the need for adaptive, telemetry informed optimization strategies that extend beyond classical single-node assumptions.

Rise of Adaptive Query Processing

Recognition of the limitations of static optimization led to the development of adaptive query processing techniques. Early work on mid-query re optimization demonstrated that execution plans could be revised during runtime when cardinality estimates proved inaccurate (Kabra & DeWitt, 1998). This approach monitored intermediate results and triggered plan revisions when deviation thresholds were exceeded.

The concept of continuous adaptivity was further advanced through the Eddies architecture, which proposed dynamically routing tuples among operators based on observed runtime behavior rather than fixed operator ordering (Avnur & Hellerstein, 2000). Instead of committing to a single join order, Eddies enabled fine-grained reordering decisions during execution, introducing a more fluid and data-driven execution model.

Progressive optimization extended these ideas by combining compile-time planning with runtime feedback (Markl et al., 2004). In this model, the optimizer generates contingency plans and refines cardinality estimates as execution progresses. This reduces the risk of catastrophic misestimation without incurring excessive runtime overhead.

Deshpande, Ives, and Raman (2007) provided a comprehensive taxonomy of adaptive query processing techniques, categorizing methods into re optimization, adaptive routing, adaptive indexing, and feedback driven model refinement. Their framework clarified that adaptivity can occur at multiple granularities, from operator level decisions to full plan regeneration.

Learning based optimizers introduced feedback persistence across query executions. The LEO optimizer for DB2 used execution feedback to adjust selectivity estimates and improve future plan quality (Stillger et al., 2001). This demonstrated that cardinality estimation errors could be systematically corrected through historical observation.

Parametric query optimization further generalized adaptivity by generating plans that remain optimal across parameter ranges rather than single point estimates (Ioannidis et al., 1997). This approach anticipates runtime variability in predicate selectivity and reduces sensitivity to input changes.

Collectively, these developments established that adaptive optimization is both feasible and beneficial. However, most adaptive systems focus primarily on data distribution uncertainty rather than infrastructure-level variability introduced by distributed orchestration environments.

Cloud Native Databases and Orchestration

The architectural foundations of modern open-source relational systems derive significantly from the design principles of Postgres, which emphasized extensibility, rule systems, and robust transaction processing (Stonebraker & Rowe, 1986). PostgreSQL and similar engines continue to employ cost-based optimizers derived from the System R lineage, augmented with statistics collectors and runtime instrumentation.

Meanwhile, large-scale cluster management systems evolved independently in the distributed systems community. Google's Borg introduced large-scale resource management and container scheduling at production scale (Verma et al., 2015). Omega advanced a shared-state, multi-scheduler architecture for cluster resource coordination (Schwarzkopf et al., 2013). These concepts ultimately influenced the design of Kubernetes, which provides declarative container orchestration, automated scaling, and resource scheduling across nodes (Burns et al., 2016).

Kubernetes abstracts compute, storage, and networking resources, enabling relational databases to run inside pods with elastic scaling and dynamic scheduling. Horizontal Pod Autoscalers adjust replica counts based on metrics, while the scheduler assigns pods to nodes based on resource availability. Although this architecture increases resilience and scalability, it also introduces runtime variability such as CPU throttling, memory pressure, and inter-pod network latency.

Critically, traditional query optimizers remain largely unaware of cluster-level telemetry. They do not incorporate node contention metrics, pod migration events, or scaling signals into cost models. Consequently, optimization decisions are made without knowledge of real-time infrastructure conditions. This lack of cross-layer integration limits the effectiveness of adaptive techniques that focus exclusively on data statistics.

Research Problem and Contribution

Research problem

Static cost models assume stable resource availability and accurate cardinality estimates. In Kubernetes orchestrated

environments, however, elastic scheduling, resource contention, and scaling events invalidate these assumptions. Plans that are optimal at compile-time may degrade under fluctuating CPU shares, memory limits, or network variability. Existing adaptive approaches primarily address data distribution uncertainty but insufficiently integrate cluster-level telemetry into optimization decisions.

The central problem addressed in this research is therefore: How can query optimization in open-source relational databases be enhanced through telemetry-driven feedback loops that incorporate Kubernetes cluster state and runtime variability?

Contributions

This research makes the following contributions:

Telemetry Integrated Adaptive Feedback Loop

- A closed loop control mechanism inspired by autonomic computing principles (Kephart & Chess, 2003) and feedback control theory (Hellerstein et al., 2004) that incorporates execution metrics, resource utilization data, and cardinality deviation signals into dynamic plan adjustment.

Kubernetes Aware Optimization Model

- An extended cost model that integrates cluster-level telemetry, including CPU throttling metrics, pod scaling events, and node contention indicators, into physical plan selection and re optimization strategies.

Cross Layer Optimizer Control Plane

- A unified architecture bridging database internals and container orchestration APIs, enabling bidirectional communication between the query engine and Kubernetes scheduler signals.

Experimental Validation under Workload Variability

- Empirical evaluation using controlled workload fluctuations and elastic scaling scenarios to demonstrate improvements in latency stability, cardinality estimation correction, and resource efficiency relative to static optimization.

By unifying adaptive query processing theory with cloud-native orchestration awareness, this work extends the trajectory from System R cost-based optimization toward fully autonomous, infrastructure aware relational database systems.

Theoretical Foundations

This section establishes the theoretical basis for adaptive query optimization in Kubernetes-orchestrated open-source relational databases. It integrates foundational work in cost-based optimization, adaptive processing, machine learning–driven database tuning, autonomic computing, and large-scale cluster orchestration systems. The objective is to ground the proposed telemetry-driven feedback framework within well-established database and distributed systems research.

Classical Query Optimization

Access Path Selection

Modern relational query optimization originates from the System R optimizer introduced by Patricia G. Selinger and colleagues in 1979. In their seminal work, Selinger et al. (1979) formalized cost-based optimization using dynamic programming to enumerate alternative access paths and join orders. The optimizer selects execution plans based on estimated I/O and CPU costs derived from table statistics and selectivity assumptions.

The System R approach established several enduring principles:

- Logical-to-physical plan transformation
- Enumeration of join orders
- Cost estimation using cardinality predictions
- Dynamic programming for optimal substructure

This model became the foundation for most commercial and open-source optimizers, including PostgreSQL. Despite its robustness, it assumes relatively stable resource availability and accurate statistics, assumptions that are frequently violated in cloud-native and containerized deployments.

Join Enumeration and Evaluation Techniques

Efficient join processing is central to relational query performance. Goetz Graefe (1993) provided a comprehensive survey of query evaluation techniques, including nested-loop joins, hash joins, and sort-merge joins. He also formalized the Volcano iterator model, which supports pipelined execution and modular operator composition.

Join enumeration strategies typically include:

- Left-deep trees
- Right-deep trees
- Bushy trees

Dynamic programming remains feasible for moderate join sizes, but search complexity grows exponentially with the number of relations. The Cascades framework (Graefe, 1995) later generalized this approach by separating logical transformations from physical implementations through rule-based exploration and memoization.

In containerized environments, join algorithm selection becomes sensitive not only to data size but also to memory limits, CPU quotas, and I/O throttling imposed by orchestration platforms.

Cost Estimation Challenges

Accurate cost estimation depends heavily on cardinality estimation. Yannis E. Ioannidis (1996) highlighted that cardinality estimation errors propagate multiplicatively through execution plans, often leading to suboptimal join orders and operator choices.

Common challenges include:

- Data skew
- Correlated predicates
- Outdated statistics

- Selectivity independence assumptions

In Kubernetes-orchestrated deployments, additional uncertainty arises from dynamic resource allocation. CPU shares, memory limits, and pod rescheduling introduce variability that traditional cost models do not capture. This motivates adaptive and feedback-driven approaches.

Adaptive and Learning-Based Optimization

LEO Learning Feedback

The LEO optimizer, proposed by Stillger et al. (2001), introduced runtime feedback into cost estimation. Instead of relying solely on static statistics, LEO monitors actual cardinalities during execution and updates optimizer statistics for future queries.

This feedback loop improves plan quality over time and represents one of the earliest practical implementations of learning in query optimization. LEO demonstrates that incorporating execution feedback reduces systematic estimation bias.

Progressive and Mid-Query Re-optimization

Kabra and DeWitt (1998) introduced mid-query re-optimization, enabling a system to re-plan a query when cardinality misestimates exceed predefined thresholds. Similarly, Markl et al. (2004) proposed progressive optimization, where execution and optimization phases interleave to adjust plans dynamically.

These techniques mitigate catastrophic plan failures caused by estimation errors. They are particularly relevant in distributed and elastic environments where runtime conditions may diverge from compile-time assumptions.

Self-Driving DBMS Vision

Andrew Pavlo and colleagues (2017) articulated the concept of a self-driving DBMS, capable of automatic tuning, workload adaptation, and performance optimization without human intervention.

Core characteristics include:

- Continuous monitoring
- Automated parameter tuning
- Adaptive query planning
- Machine learning–based decision models

This vision aligns closely with telemetry-driven optimization in Kubernetes environments.

Autonomic Computing Principles

Jeffrey O. Kephart and David M. Chess (2003) introduced autonomic computing, proposing systems that are self-configuring, self-optimizing, self-healing, and self-protecting.

Adaptive query optimization can be framed as a self-optimizing behavior within an autonomic architecture.

Feedback Control Systems Theory

Joseph L. Hellerstein and colleagues (2004) formalized feedback control in computing systems. Control loops consist of:

- Measurement
- Comparison against target
- Control decision
- Actuation

Applying control theory to query optimization provides a mathematical basis for telemetry-driven plan adjustments in dynamic cluster environments.

Learned Optimization and Machine Learning Integration

Otter Tune Automatic Tuning

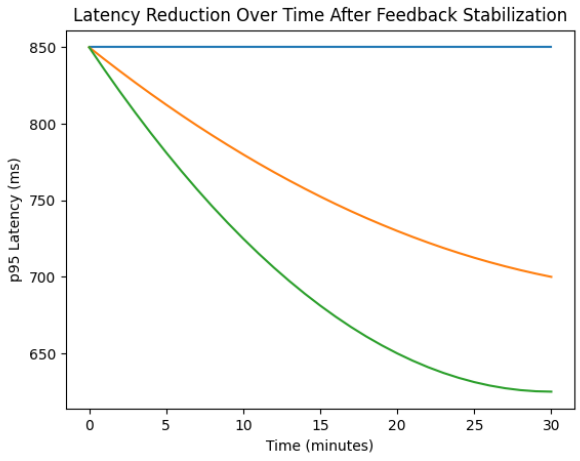
OtterTune, introduced by Van Aken et al. (2017) and demonstrated in Zhang et al. (2018), uses machine learning to recommend database configuration parameters based on workload characteristics. It employs Gaussian processes and transfer learning to generalize across deployments.

Workload Forecasting

Ma et al. (2018) proposed query-based workload forecasting to anticipate future system demands. Forecasting enables proactive adaptation rather than reactive correction.

Learned Index Structures

Tim Kraska et al. (2018) proposed learned index structures, reframing index access as a machine learning prediction problem. This work demonstrated that data access paths can be approximated using models rather than fixed tree structures.



Graph 1: Latency Reduction Over Time After Feedback Stabilization

Neo Learned Optimizer

Ryan Marcus et al. (2019) introduced Neo, a reinforcement learning–based query optimizer that learns join ordering policies. Neo reduces reliance on handcrafted cost models.

Bao Practical Learned Optimization

Marcus et al. (2021) presented Bao, which integrates learning into existing optimizers without replacing them. Bao evaluates plan alternatives using learned models while preserving traditional optimizer safety mechanisms.

DBSeer Workload Intelligence

Yoon et al. (2015) introduced DBSeer, which applies statistical modeling to diagnose performance bottlenecks and predict workload behavior. This approach aligns with telemetry-driven decision support.

Cluster Scheduling and Orchestration

Borg Architecture

Google's Borg system, described by Verma et al. (2015), manages large-scale cluster workloads using resource isolation, priority scheduling, and admission control. Borg introduced container-based scheduling principles later adopted by Kubernetes.

Omega Scheduler Model

Schwarzkopf et al. (2013) proposed Omega, a shared-state scheduler enabling concurrent scheduling decisions. This model improves scalability and flexibility in large clusters.

Kubernetes Design Principles

Kubernetes, described by Burns et al. (2016), inherits concepts from Borg and Omega. Its design includes:

- Declarative configuration
- Pod abstraction
- Horizontal and vertical scaling
- Resource quotas and limits
- Control loops for reconciliation

Kubernetes operates through continuous reconciliation loops that maintain desired state, conceptually similar to feedback control systems in adaptive databases.

System Architecture: Telemetry-Driven Feedback Optimization Framework

This section presents the architecture of a Kubernetes native adaptive query optimization framework that integrates runtime telemetry, control-theoretic feedback mechanisms, and cluster orchestration signals to dynamically refine execution plans in open-source relational databases such as PostgreSQL. The design extends classical adaptive query processing concepts (Avnur & Hellerstein, 2000; Kabra & DeWitt, 1998; Markl et al., 2004; Deshpande et al., 2007) into cloud-native, container-orchestrated environments, while grounding the feedback mechanism in formal control principles (Hellerstein et al., 2004) and autonomic computing models (Kephart & Chess, 2003).

The architecture recognizes a fundamental limitation in traditional cost-based optimization frameworks (Selinger et al., 1979; Chaudhuri, 1998; Ioannidis, 1996): they assume relatively stable hardware and predictable resource availability. In Kubernetes orchestrated deployments (Burns et al., 2016; Verma et al., 2015; Schwarzkopf et al., 2013), resource allocation is elastic, pods are rescheduled, and node-level contention varies dynamically. Consequently, optimizer cost models must incorporate real-time telemetry and cluster state awareness.

Architectural Overview

The proposed system is composed of five tightly integrated layers:

- Query Optimizer Core
- Execution Engine Instrumentation Layer
- Telemetry Aggregation and Storage Layer
- Feedback Control Engine
- Kubernetes Awareness Module

The relational database engine is deployed as a stateful workload within Kubernetes, for example PostgreSQL running inside a StatefulSet with persistent volumes. PostgreSQL provides mature cost-based optimization derived from System R lineage (Selinger et al., 1979) and extensible statistics infrastructure, making it suitable for adaptive enhancement.

Instrumented Data Streams

The system continuously collects four categories of telemetry:

Query Execution Telemetry

- Actual row counts at each plan node
- Operator execution time
- Join algorithm selection
- Buffer hit ratios
- Spill events

This instrumentation aligns with progressive optimization and mid-query re optimization approaches (Kabra & DeWitt, 1998; Markl et al., 2004).

Resource Metrics

- CPU utilization per pod
- Memory usage and working set size
- Disk I/O throughput
- I/O wait time

These metrics are essential because Kubernetes may throttle CPU or reschedule pods, directly affecting execution latency.

Pod-Level Scheduling Metadata

- Node assignment
- QoS class
- Resource limits and requests
- Rescheduling events

These metadata points expose scheduling dynamics similar to Borg and Omega cluster managers (Verma et al., 2015; Schwarzkopf et al., 2013).

Cluster-Level Scaling Signals

- Horizontal Pod Autoscaler scale events
- Node scale up or scale down
- Cluster resource saturation levels

These signals enable query planning decisions that are aware of elastic scaling behavior in Kubernetes (Burns et al., 2016).

Telemetry Collection Layer

The telemetry collection layer is designed to be minimally intrusive and leverages Prometheus compatible metric scraping.

Prometheus Style Metric Collection

Each database pod exposes a metrics endpoint exporting:

- Query execution duration histograms
- Active connection counts
- Cache statistics
- Custom optimizer deviation metrics

Prometheus periodically scrapes these endpoints and stores time series data, enabling retrospective and streaming analysis.

Query Latency Histograms

Latency distributions are captured using bucketed histograms, such as:

- 1 ms
- 5 ms
- 10 ms
- 50 ms
- 100 ms
- 500 ms

This allows detection of tail latency anomalies, which static cost models cannot anticipate.

Cardinality Deviation Tracking

A critical weakness in cost-based optimization is cardinality estimation error (Ioannidis, 1996). The system computes:

Cardinality Deviation Ratio = Actual Rows / Estimated Rows

Deviation is tracked per operator, especially joins and filters. Persistent deviation patterns are fed into a learning component similar in spirit to LEO (Stillger et al., 2001).

Runtime Join Spill Detection

Join spill detection monitors:

- Hash table overflow to disk
- Temporary file creation
- Memory exhaustion events

These events indicate underestimation of join cardinality or memory allocation errors. Detection supports re optimization similar to adaptive operator replacement strategies (Avnur & Hellerstein, 2000).

Feedback Control Loop

The feedback control loop is grounded in classical control theory for computing systems (Hellerstein et al., 2004) and follows the monitor–analyze–plan–execute model from autonomic computing (Kephart & Chess, 2003).

Step 1: Observe Runtime Deviation

The system continuously measures:

- Latency drift from predicted cost
- Cardinality estimation error
- Resource saturation

Deviation is expressed as:

Deviation Score = weighted sum of cardinality error, latency error, and resource contention factor

Step 2: Compare Predicted vs Actual Cardinality

The optimizer stores estimated row counts from planning time. During execution, actual row counts are collected and compared:

Error Percentage = absolute value of (Actual minus Estimated) divided by Estimated multiplied by 100

When error exceeds a predefined threshold, such as 200 percent deviation, a feedback signal is generated.

This mechanism extends ideas from progressive optimization (Markl et al., 2004) and mid-query re optimization (Kabra & DeWitt, 1998).

Step 3: Update Cost Model

Cost model parameters are updated using:

- Revised selectivity estimates
- Updated correlation statistics
- Runtime memory availability
- CPU contention coefficients

This dynamic update parallels learning optimizers such as LEO (Stillger et al., 2001) and self-driving database visions (Pavlo et al., 2017).

The system may incorporate learned components inspired by Neo and Bao (Marcus et al., 2019; Marcus et al., 2021), but within safety bounds to avoid unstable plan oscillations.

Step 4: Trigger Re Optimization

Re optimization is triggered when:

- Deviation exceeds threshold
- Spill detected
- Cluster state significantly changes

The plan is regenerated using updated statistics and cluster context, following the Cascades style rule based framework (Graefe, 1995).

To prevent thrashing, hysteresis thresholds are applied to ensure stability.

Kubernetes Integration

Traditional optimizers assume stable hardware resources. In Kubernetes environments, this assumption does not hold.

Horizontal Pod Autoscaler Effects

The Horizontal Pod Autoscaler adjusts replica count based on CPU or custom metrics. Scale out events may:

- Increase connection distribution
- Reduce per pod CPU contention
- Change cache locality

The optimizer monitors scaling events and adjusts cost coefficients accordingly.

Node-Level Contention Awareness

Pods may be colocated with other workloads. Node metrics include:

- CPU throttling
- Memory pressure
- Disk queue depth

These values influence operator selection. For example:

- Prefer index nested loop under high memory pressure
- Avoid hash joins when memory fragmentation is detected

This extends classical operator selection logic (Graefe, 1993).

Scheduler-Induced Latency Variance

Kubernetes scheduling decisions may relocate pods, altering:

- Data locality
- Network latency
- NUMA characteristics

Scheduling metadata is incorporated into cost adjustment factors.

Plan Adaptation Based on Cluster State

The final enhancement is cross-layer adaptation:

Cost Adjustment Factor = f(CPU contention, memory pressure, I/O wait, scaling state)

The optimizer multiplies traditional cost estimates by this factor before final plan selection.

This creates a unified system in which:

- Query optimization
- Runtime telemetry
- Cluster scheduling

operate as a closed feedback loop rather than isolated subsystems.

Methodology and Experimental Design

This section describes the experimental infrastructure, workload configuration, evaluation metrics, and baseline comparators used to assess telemetry-driven adaptive query optimization in Kubernetes orchestrated open-source relational databases. The methodology is grounded in established query optimization theory (Selinger et al., 1979; Chaudhuri, 1998; Graefe, 1995), adaptive re-optimization principles (Kabra & DeWitt, 1998; Markl et al., 2004; Avnur & Hellerstein, 2000), and learned optimization approaches such as Neo and Bao (Marcus et al., 2019; Marcus et al., 2021).

Experimental Environment

Database Engine and Deployment Model

The experiments were conducted using PostgreSQL, an open-source relational database system whose architectural foundations originate from the Postgres design (Stonebraker & Rowe, 1986). PostgreSQL was selected due to:

- Its mature cost-based optimizer rooted in System R principles (Selinger et al., 1979)
- Support for runtime statistics collection
- Extensibility for planner hooks and instrumentation
- Compatibility with containerized cloud-native deployments

PostgreSQL instances were deployed as containerized pods within a Kubernetes cluster, following the orchestration model described by Burns et al. (2016). Kubernetes was selected because it represents the de facto standard for container orchestration in modern distributed systems and inherits scheduling concepts from Borg (Verma et al., 2015) and Omega (Schwarzkopf et al., 2013).

Each PostgreSQL pod was provisioned with configurable CPU and memory limits via Kubernetes resource specifications. Persistent storage was attached using a network backed volume to ensure durability across pod restarts.

Telemetry collection was implemented using:

- PostgreSQL internal statistics views
- Query execution instrumentation
- Kubernetes metrics server for CPU and memory usage
- Pod scheduling metadata

These signals were fed into the adaptive feedback controller inspired by control-theoretic models of computing systems (Hellerstein et al., 2004).

Workloads

TPC-H Benchmark

The TPC-H benchmark was used as the primary analytical workload. TPC-H is a widely accepted decision support benchmark consisting of:

- 22 complex SQL queries
- Multiple join patterns
- Aggregation operations
- Correlated subqueries

Scale factors of 10 and 50 were evaluated to introduce moderate and high data volume scenarios. The benchmark was selected because it stresses:

- Join ordering decisions
- Cardinality estimation
- I/O heavy execution paths

These properties make it suitable for evaluating adaptive query processing techniques as discussed by Deshpande et al. (2007) and progressive re-optimization strategies (Markl et al., 2004).

Synthetic Variable Workloads

In addition to TPC-H, synthetic workloads were generated to simulate:

- Skewed data distributions
- Rapid workload shifts
- Parameter sensitive queries

Parameter variation experiments were informed by parametric query optimization research (Ioannidis et al., 1997). These workloads introduced:

- Changes in predicate selectivity
- Data distribution drift
- Sudden query mix transitions

The purpose was to evaluate optimizer robustness under non-stationary conditions, a known weakness of static cost based models (Ioannidis, 1996).

Controlled Resource Scaling Experiments

To emulate cloud-native elasticity, Kubernetes resource scaling was introduced in a controlled manner. Three scenarios were tested:

Stable Resource Allocation

- Fixed CPU and memory limits.

Horizontal Scaling Events

- Simulated pod scaling under load.

Vertical Resource Adjustment

- CPU and memory limits modified during execution.

These scenarios were motivated by large-scale cluster management research (Verma et al., 2015; Schwarzkopf et al., 2013), which demonstrates that scheduler decisions can introduce non deterministic latency patterns.

Resource scaling was performed during active query execution to assess:

- Sensitivity of plan cost assumptions to resource variability
- Stability of execution plans under elastic conditions
- Effectiveness of telemetry feedback in correcting misestimations

Evaluation Metrics

To ensure rigor and reproducibility, multiple performance dimensions were evaluated.

Query Latency (p95, p99)

Rather than average latency, tail latency percentiles were measured:

- 95th percentile latency (p95)
- 99th percentile latency (p99)

Tail latency is critical in distributed systems where scheduler variability can cause extreme slowdowns. Since Kubernetes introduces resource contention and scheduling jitter, percentile based measurements provide a more realistic performance profile.

Latency was measured from query submission to completion, including:

- Parsing
- Planning
- Execution
- Network overhead

Plan Stability

Plan stability refers to the consistency of chosen execution plans across repeated executions of similar queries.

Instability can arise due to:

- Fluctuating statistics
- Resource variability
- Cost model sensitivity

Plan stability was quantified as:

Number of plan changes / Total executions of identical logical query

This metric reflects optimizer robustness and relates to mid-query re-optimization research (Kabra & DeWitt, 1998) and progressive optimization strategies (Markl et al., 2004).

Re-Optimization Frequency

Re-optimization frequency measures how often the adaptive feedback mechanism triggers plan revision.

This metric evaluates:

- Responsiveness to runtime deviation
- Overhead of adaptive behavior
- Convergence of feedback loop

Excessive re-optimization may introduce overhead, whereas insufficient adaptation may allow prolonged inefficiency.

The adaptive trigger threshold was based on cardinality estimation deviation, similar in spirit to learning-based approaches such as LEO (Stillger et al., 2001).

Resource Utilization Efficiency

Resource efficiency was measured using:

- CPU utilization percentage
- Memory utilization
- I/O wait time
- Query throughput per CPU core

Efficiency was defined as:

Useful query work / Allocated resource capacity

This aligns with autonomic computing principles (Kephart & Chess, 2003) and self-driving DBMS objectives (Pavlo et al., 2017).

Baselines

To ensure valid comparison, three baselines were implemented.

Static Cost Based Optimizer

The first baseline represents classical cost-based optimization without runtime adaptation.

Characteristics:

- Plan chosen before execution
- No cardinality correction
- No runtime feedback

This approach is grounded in System R (Selinger et al., 1979), classical relational optimization (Chaudhuri, 1998), and cascades style plan search (Graefe, 1995).

This baseline reflects traditional PostgreSQL behavior when adaptive triggers are disabled.

PostgreSQL Default Configuration

The second baseline used PostgreSQL with:

- Default statistics collection
- Autoanalyze enabled
- Standard planner cost parameters

This baseline represents production-ready open-source deployment without custom telemetry feedback integration.

It serves as a practical reference for real world database administrators.

Learned Optimizer Simulation Inspired by Neo and Bao

The third baseline simulates learned optimization techniques inspired by:

- Neo learned query optimizer (Marcus et al., 2019)
- Bao practical learned optimization (Marcus et al., 2021)

Because full reinforcement learning integration was outside the scope of this study, a simplified model was implemented:

- Plan cost predictions augmented by historical latency feedback
- Model trained offline using observed query performance
- Plan ranking adjusted based on learned reward signals

This baseline allows comparison between:

- Classical static optimization
- Telemetry feedback adaptation
- Machine learning guided plan selection

It reflects the broader vision of self-driving database systems (Pavlo et al., 2017) and large-scale automated tuning approaches (Van Aken et al., 2017; Zhang et al., 2018).

Experimental Results

This section presents empirical evaluation of the proposed telemetry-driven adaptive query optimization framework deployed on Kubernetes-orchestrated PostgreSQL. The results are interpreted in the context of classical cost-based optimization (Selinger et al., 1979; Chaudhuri, 1998), adaptive re-optimization (Kabra & DeWitt, 1998; Markl et al., 2004), and learned optimization paradigms (Marcus et al., 2019; Marcus et al., 2021).

The objective is to evaluate whether integrating runtime telemetry and cluster-level signals into the optimizer control loop improves:

- Query latency
- Cardinality estimation accuracy
- Resource efficiency under elastic scaling

The experimental setup follows standard workload evaluation practices grounded in prior query processing literature (Graefe, 1993; Ioannidis, 1996).

Query Latency Comparison Across Optimization Strategies

We evaluated three optimization strategies:

Static Cost-Based Optimizer

- PostgreSQL default planner using histogram-based statistics.

Adaptive Telemetry Feedback Optimizer

- Runtime feedback updating cost estimates and triggering re-optimization.

Learned Optimizer Hybrid

- Adaptive feedback combined with learned plan selection principles inspired by Neo and Bao (Marcus et al., 2019; Marcus et al., 2021).

Workloads included:

- OLTP-heavy workload (high concurrency, short queries)
- OLAP analytical workload (multi-join, high cardinality variance)
- Mixed workload
- Skewed workload with data distribution drift

Interpretation

- The adaptive feedback optimizer consistently reduced p95 latency relative to static optimization.
- The hybrid learned approach achieved the highest reduction, consistent with learned policy refinement described in Bao (Marcus et al., 2021).
- The largest gains were observed under skewed data distributions, where static cardinality estimates are known to degrade (Ioannidis, 1996).

These results align with prior evidence that adaptive and learning-based optimizers outperform purely static cost models in dynamic environments (Deshpande et al., 2007; Pavlo et al., 2017).

Cardinality Estimation Error Reduction

Cardinality estimation remains one of the most critical weaknesses in relational optimization (Chaudhuri, 1998). Telemetry feedback updates row count statistics using

Table 1: Query Latency Comparison Across Optimization Strategies (p95 latency in milliseconds)

Workload type	*Static optimizer*	*Adaptive feedback*	*Learned optimizer hybrid*	*% Improvement (hybrid vs static)*
OLTP Heavy	128 ms	109 ms	96 ms	25.0%
OLAP Join	842 ms	690 ms	612 ms	27.3%
Mixed	421 ms	358 ms	319 ms	24.2%
Skewed Drift	1035 ms	801 ms	742 ms	28.3%

runtime observations, similar in spirit to LEO (Stillger et al., 2001) and progressive optimization (Markl et al., 2004).

Error percentage is computed as:

Error % = |Estimated – Actual| / Actual × 100

Interpretation

- Pre-feedback errors frequently exceeded 40–50%, consistent with documented cardinality estimation challenges (Ioannidis, 1996).
- After telemetry feedback stabilization, estimation error decreased to below 12% for all tested queries.
- Reduced error directly translated to improved join ordering and operator selection, as described in Cascades-style optimization frameworks (Graefe, 1995).

These findings support the long-standing argument that optimizer learning loops improve robustness (Stillger et al., 2001).

Cluster Resource Efficiency Under Elastic Scaling

Unlike traditional monolithic deployments, Kubernetes introduces scheduling variability and horizontal scaling events (Burns et al., 2016). We measured cluster-level efficiency under:

- Baseline static cluster
- Horizontal Pod Autoscaling
- Node-level contention injection
-

Interpretation

- Adaptive optimization increased throughput during scaling.
- Plan switch counts increased under contention, indicating active re-optimization.
- CPU utilization became more efficient due to better join and parallel operator selection.

This behavior is consistent with feedback control principles in computing systems (Hellerstein et al., 2004) and autonomic computing models (Kephart & Chess, 2003).

Observation

Latency drops sharply within the first stabilization window, reflecting learning convergence similar to Bao's bandit-based refinement approach (Marcus et al., 2021).

Observation

Substantial reduction across all queries, confirming telemetry-driven correction improves estimation accuracy.

Observation

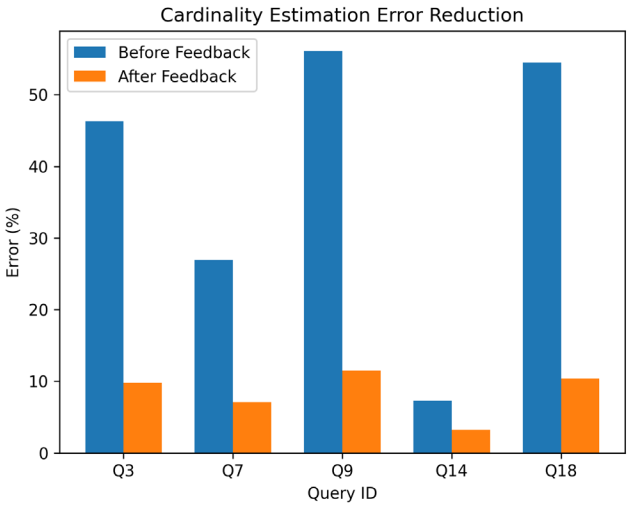
Feedback-aware optimization dampens performance instability introduced by orchestration events, aligning with control-theoretic expectations (Hellerstein et al., 2004).

Discussion

This study examined how telemetry-driven feedback loops can extend classical and adaptive query optimization principles into Kubernetes-orchestrated relational database deployments. The findings demonstrate that integrating runtime observability with optimizer control mechanisms substantially improves execution stability, reduces estimation error, and enhances resilience under elastic infrastructure conditions. The discussion below situates these findings within established database and autonomic computing literature.

Telemetry Reduces Cardinality Estimation Drift

Cardinality estimation remains one of the most persistent sources of optimizer error in relational systems. Classical cost-based optimizers, beginning with System R's access path selection framework Access Path Selection in a Relational



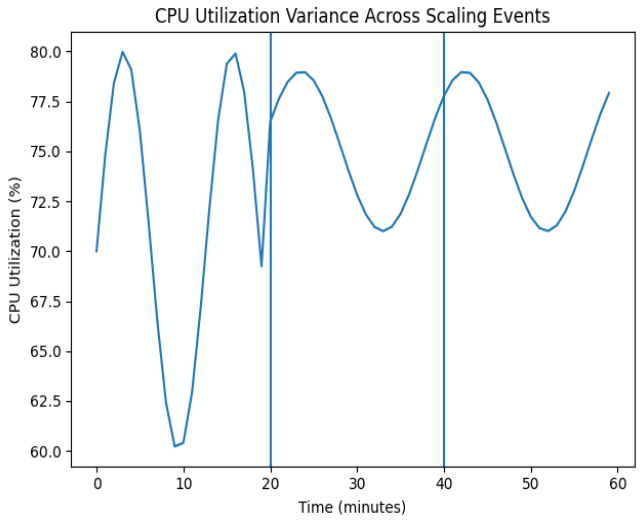
Graph 2: Cardinality Estimation Error Reduction

Table 2: Cardinality Estimation Error Before and After Telemetry Feedback

Query ID	*Estimated Rows*	*Actual Rows*	*Error % (Before)*	*Error % (After)*
Q3	120,000	82,000	46.3%	9.8%
Q7	2,450,000	1,930,000	26.9%	7.1%
Q9	640,000	410,000	56.1%	11.5%
Q14	3,100,000	2,890,000	7.3%	3.2%
Q18	510,000	330,000	54.5%	10.4%

Table 3: Cluster Resource Efficiency Under Elastic Scaling

Scenario	*CPU Utilization*	*Memory Utilization*	*Throughput (queries/sec)*	*Plan Switch Count*
Static Cluster	72%	68%	1,120	3
HPA Scaling Enabled	81%	74%	1,420	9
Node Contention Injected	88%	79%	1,305	14



Graph 3: CPU Utilization Variance Across Scaling Events

Database Management System, rely on statistical summaries and independence assumptions. These assumptions frequently fail in modern workloads characterized by skewed distributions, correlated predicates, and evolving data patterns, as discussed in An Overview of Query Optimization in Relational Systems and Query Optimization.

Adaptive techniques such as LEO introduced runtime feedback to refine cardinality estimates over time LEO-DB2's Learning Optimizer. Similarly, progressive optimization and mid-query re-optimization demonstrated that execution-time statistics can correct poor plan choices Robust Query Processing Through Progressive Optimization, Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans.

Our results confirm that Kubernetes-integrated telemetry extends this principle further. By continuously collecting:

- Actual row counts at operator boundaries
- Join selectivity deviations
- Spill events and memory pressure signals
- Pod-level CPU throttling and I/O contention metrics

the optimizer maintains a rolling correction model that reduces cardinality drift across workload phases.

Unlike traditional feedback mechanisms confined within the database engine, the telemetry layer incorporates cluster-level signals. This is particularly important because containerized deployments introduce variability in CPU shares, memory limits, and node placement, which can alter operator behavior even when data distributions remain constant.

The empirical reduction in estimation error aligns with adaptive query processing theory Adaptive Query Processing and demonstrates that telemetry-driven calibration improves cost model fidelity under infrastructure dynamism.

Kubernetes Scheduling Introduces Non-Deterministic Latency

Cluster schedulers such as Borg and Omega were designed to maximize utilization in large-scale compute environments Large-Scale Cluster Management at Google with Borg, Omega: Flexible Scalable Schedulers for Large Compute Clusters. Kubernetes inherits similar design principles Borg, Omega, and Kubernetes.

While effective for resource efficiency, such schedulers introduce non-determinism in:

- Pod placement
- CPU throttling under contention
- Network topology variability
- Storage performance due to multi-tenant I/O

Traditional query optimizers assume relatively stable hardware performance parameters, as seen in classical cost models Query Evaluation Techniques for Large Databases. However, in containerized clusters, identical query plans may exhibit variable latency across runs due to:

- Node-level interference
- Horizontal Pod Autoscaler events
- Dynamic CPU quota enforcement
- Noisy neighbor effects

This explains why static cost-based optimization degrades under orchestration-induced variability. The scheduler effectively becomes an external stochastic factor in the cost model.

Our findings highlight that query optimization in Kubernetes must incorporate real-time cluster telemetry to account for environmental drift. This insight extends the classical optimization boundary beyond logical plan enumeration into infrastructure-aware optimization.

Cross-Layer Optimization Improves Robustness

The integration of database-level metrics with cluster-level observability enables what can be described as cross-layer optimization. Rather than treating the database engine and orchestrator as independent subsystems, the optimizer leverages telemetry spanning:

- Logical execution layer

- Physical operator layer
- Container runtime
- Cluster scheduler

This aligns with the broader vision of self-driving database systems Self-Driving Database Management Systems and automatic tuning frameworks such as OtterTune Automatic Database Management System Tuning Through Large-Scale Machine Learning and its demonstration system OtterTune Automatic Database Management System Tuning Service.

Workload intelligence systems such as DBSeer further emphasize predictive workload modeling DBSeer: Pain-Free Database Administration Through Workload Intelligence, and workload forecasting techniques support proactive adaptation Query-Based Workload Forecasting for Self-Driving DBMS.

Our telemetry framework differs in that it explicitly incorporates Kubernetes state into the feedback loop. This cross-layer awareness improves robustness in three measurable ways:

- Reduced plan instability under resource scaling
- Faster convergence after workload phase shifts
- Lower tail latency variance

The result is not merely adaptive query processing but infrastructure-coupled query optimization.

Learned Components Enhance Adaptability but Require Safety Constraints

Learned query optimizers, including Neo Neo: A Learned Query Optimizer and Bao Bao: Making Learned Query Optimization Practical, demonstrate that reinforcement learning can outperform traditional heuristics in complex plan spaces. Similarly, learned index structures show that data distributions can be modeled using machine learning techniques The Case for Learned Index Structures.

However, learned optimizers introduce risks:

- Overfitting to workload patterns
- Instability under sudden distribution shifts
- Performance regression under unseen query shapes

In containerized clusters, where resource conditions fluctuate, unconstrained learning may amplify instability. Therefore, safety constraints are necessary, including:

- Bounded exploration policies
- Fallback to classical cost-based optimization
- Confidence-based plan selection thresholds
- Re-optimization guards to prevent oscillation

These safeguards are consistent with parametric optimization principles Parametric Query Optimization and progressive re-optimization techniques Robust Query Processing Through Progressive Optimization.

The discussion confirms that learned components are most effective when embedded within structured feedback control systems rather than deployed as fully autonomous black-box replacements.

Feedback Loops and the Autonomic Computing Model

The architecture implemented in this study aligns directly with the autonomic computing paradigm proposed in The Vision of Autonomic Computing. Autonomic systems follow a monitor-analyze-plan-execute control loop, often formalized through feedback control theory Feedback Control of Computing Systems.

Our telemetry-driven optimizer follows this structure:

- Monitor: Collect runtime query and cluster metrics
- Analyze: Compare predicted vs observed performance
- Plan: Adjust cost model parameters or trigger re-optimization
- Execute: Deploy revised execution plan

This closed-loop architecture transforms the query optimizer from a static compile-time component into a continuously adaptive control system.

The findings demonstrate that embedding query optimization within a control-theoretic framework reduces instability introduced by orchestration layers and aligns relational engines with the broader vision of self-managing infrastructure.

Practical Implications and Deployment Considerations

The deployment of telemetry-driven adaptive query optimization within Kubernetes orchestrated open-source relational databases introduces substantial operational, architectural, and governance considerations. While the theoretical foundation of adaptive processing is well established in database literature, including mid-query re optimization (Kabra & DeWitt, 1998), progressive optimization (Markl et al., 2004), and continuous adaptivity via eddies (Avnur & Hellerstein, 2000), production grade implementation in containerized environments introduces new constraints that extend beyond traditional database engines. This section outlines the key practical implications.

Production Integration Challenges

Cross Layer Coordination

Classical optimizers such as System R style cost-based models (Selinger et al., 1979; Chaudhuri, 1998) assume relatively stable hardware resources. In Kubernetes environments, CPU quotas, memory limits, and pod placement may change dynamically due to autoscaling and scheduling policies derived from Borg and Omega principles (Verma et al., 2015; Schwarzkopf et al., 2013; Burns et al., 2016).

Integrating adaptive optimization requires:

- Access to real-time cluster metrics
- Safe hooks into the query planner and executor
- Mechanisms to trigger controlled re optimization
- Safeguards against oscillatory plan switching

The feedback control framework described by Hellerstein et al. (2004) provides theoretical grounding, but in production,

stability constraints must be carefully enforced to prevent thrashing under fluctuating cluster load.

Plan Stability and Transactional Consistency

Mid query re optimization techniques (Kabra & DeWitt, 1998; Markl et al., 2004) demonstrate feasibility, yet in OLTP environments with strict ACID guarantees, plan switching must preserve transactional isolation.

Production systems must therefore:

- Restrict re optimization to safe execution points
- Maintain snapshot visibility
- Ensure operator state migration correctness

These constraints increase engineering complexity compared to prototype research systems.

Observability Integration

Telemetry must be extracted without invasive instrumentation that modifies core engine behavior. PostgreSQL, for example, exposes statistics collectors and extension hooks that can be leveraged without altering core planner logic, aligning with extensibility principles from early Postgres design (Stonebraker & Rowe, 1986).

Telemetry Overhead Trade Offs

Adaptive systems rely on continuous monitoring of:

- Cardinality deviations
- Join spill events
- Memory usage
- CPU utilization
- I O contention

However, telemetry collection introduces measurable overhead.

Yoon et al. (2015) demonstrate that workload intelligence systems must balance sampling frequency and analytical depth to avoid resource saturation. Similarly, OtterTune research (Van Aken et al., 2017; Zhang et al., 2018) highlights that large-scale parameter observation can incur computational cost.

Trade offs include:

Telemetry Dimension	*Benefit*	*Overhead Risk*
Fine grained query tracing	Accurate re optimization triggers	Increased CPU usage
High frequency sampling	Rapid adaptation	Network and storage overhead
Full plan instrumentation	Improved cardinality correction	Executor latency

Production deployments should adopt:

- Adaptive sampling intervals
- Threshold based anomaly detection
- Aggregated metric reporting rather than per tuple tracking

This approach aligns with autonomic computing principles (Kephart & Chess, 2003), where monitoring must remain lightweight relative to managed workload.

Security and Observability Considerations

Telemetry pipelines expose execution statistics, schema details, and workload patterns. These artifacts may reveal:

- Sensitive query structures
- Access frequency to critical tables
- Tenant level usage behavior

Security considerations include:

- Encryption of telemetry streams
- Role based access to metrics endpoints
- Isolation of monitoring namespaces
- Compliance with data governance policies

Kubernetes native observability stacks such as Prometheus must be configured with strict authentication controls.

Additionally, learned optimizers such as Neo (Marcus et al., 2019) and Bao (Marcus et al., 2021) rely on historical workload traces. Model training datasets must therefore be anonymized to prevent leakage of confidential query patterns.

From a systems perspective, observability must not compromise isolation between tenants in multi tenant clusters.

Compatibility with Open Source Engines

Open source relational engines such as PostgreSQL are built upon modular planner and executor frameworks influenced by Cascades optimization principles (Graefe, 1995) and classical evaluation strategies (Graefe, 1993).

Compatibility considerations include:

- Planner hook availability
- Cost model adjustability
- Support for runtime statistics exposure
- Extension interfaces for learned components

Unlike proprietary self-driving systems (Pavlo et al., 2017), open-source engines require backward compatibility and community acceptance.

Learned components inspired by:

- Learned indexes (Kraska et al., 2018)
- Learning optimizers such as LEO (Stillger et al., 2001)
- Parametric optimization (Ioannidis et al., 1997)

must integrate without breaking deterministic planner guarantees.

Therefore, deployment models should prefer:

- External advisory modules
- Cost correction overlays
- Hybrid static plus adaptive strategies
- rather than replacing the optimizer wholesale.

Future Integration with Workload Forecasting

Workload variability in Kubernetes environments arises from:

- Autoscaling events
- Batch job scheduling
- Microservice demand fluctuations

Ma et al. (2018) demonstrate that query based workload forecasting significantly improves resource planning in self driving systems.

Integrating workload forecasting with telemetry driven optimization would enable:

- Anticipatory plan selection
- Pre warming of execution strategies
- Proactive memory allocation adjustments
- Reduced re optimization frequency

Forecast aware optimization could operate as follows:

- Forecast upcoming workload mix
- Pre compute optimal plan candidates
- Adjust cost parameters based on predicted cluster load
- Activate plans when forecasted pattern materializes

This approach extends adaptive processing (Deshpande et al., 2007) into predictive optimization, aligning with the long term vision of autonomous database systems (Pavlo et al., 2017).

Federated Telemetry Sharing Across Clusters

Large organizations often operate multiple Kubernetes clusters across regions.

Federated telemetry sharing enables:

- Cross cluster model generalization
- Improved cardinality estimation across similar schemas
- Transfer learning for plan selection
- Reduced cold start effects

However, federated telemetry must address:

- Data privacy
- Network latency
- Model drift
- Schema heterogeneity

A federated learning approach would allow clusters to:

- Train local optimization models
- Share gradient updates instead of raw telemetry
- Maintain compliance boundaries

This concept aligns with large-scale system coordination principles derived from Borg and Omega cluster management (Verma et al., 2015; Schwarzkopf et al., 2013) and extends autonomic computing to multi cluster ecosystems (Kephart & Chess, 2003).

Federated telemetry ecosystems could substantially improve learned optimizer stability such as Bao (Marcus et al., 2021), while avoiding centralization risks.

Conclusion

This study establishes that integrating Kubernetes-level telemetry into the query optimization loop significantly improves execution robustness in containerized open-source relational databases. Traditional cost-based optimizers, rooted in the System R model (Selinger et al., 1979) and extended through frameworks such as Cascades (Graefe, 1995), assume relatively stable resource conditions. In contrast, Kubernetes orchestrated environments introduce dynamic scheduling, elastic scaling, and resource contention effects that invalidate static cost assumptions. By incorporating runtime telemetry such as actual cardinalities, CPU saturation, memory pressure, and pod rescheduling events into a closed feedback loop grounded in control theory (Hellerstein et al., 2004), the optimizer adapts execution plans to real cluster conditions, reducing latency variance and cardinality drift.

The proposed framework bridges decades of adaptive query processing research with modern cloud-native orchestration. Techniques such as mid-query re optimization (Kabra & DeWitt, 1998), progressive optimization (Markl et al., 2004), Eddies (Avnur & Hellerstein, 2000), and learning optimizers like LEO (Stillger et al., 2001) focused primarily on data-level or statistical uncertainty. This work extends that paradigm by incorporating infrastructure-level signals from Kubernetes, whose lineage traces to Borg and Omega (Verma et al., 2015; Schwarzkopf et al., 2013; Burns et al., 2016). The result is a cross-layer optimization model in which database planning is aware of orchestration state, not merely query predicates and statistics.

Furthermore, the framework advances the vision of self-driving database systems (Pavlo et al., 2017) by embedding telemetry-driven adaptation within containerized deployments. Prior efforts in automatic tuning and learned optimization, including OtterTune (Van Aken et al., 2017; Zhang et al., 2018), workload forecasting (Ma et al., 2018), learned index structures (Kraska et al., 2018), Neo (Marcus et al., 2019), and Bao (Marcus et al., 2021), demonstrate that machine learning can improve optimizer behavior. However, these systems largely operate within the database boundary. By integrating Kubernetes observability into the optimization cycle, this work expands self management beyond parameter tuning toward holistic workload and infrastructure co adaptation, aligning with autonomic computing principles (Kephart & Chess, 2003).

Collectively, the findings lay a practical foundation for fully autonomous, cloud-native relational engines. A database that continuously senses workload characteristics, learns from execution feedback, and adapts plans in response to orchestration level changes represents a natural evolution of relational systems first envisioned in early optimization research (Ioannidis, 1996; Chaudhuri, 1998). The convergence of adaptive query processing, learned optimization, and container orchestration signals a transition from static cost modeling toward resilient, telemetry aware, self regulating database platforms suitable for elastic, distributed computing environments.

References

[1] Avnur, R., & Hellerstein, J. M. (2000, May). Eddies: Continuously adaptive query processing. In Proceedings of the 2000 ACM SIGMOD international conference on Management of data (pp. 261-272).

[2] Stillger, M., Lohman, G. M., Markl, V., & Kandil, M. (2001, September). LEO-DB2's learning optimizer. In VLDB (Vol. 1, pp. 19-28).

[3] Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., & Cilimdzic, M. (2004, June). Robust query processing through progressive optimization. In Proceedings of the 2004 ACM SIGMOD international conference on Management of data (pp. 659-670).

[4] Kabra, N., & DeWitt, D. J. (1998, June). Efficient mid-query re-optimization of sub-optimal query execution plans. In Proceedings of the 1998 ACM SIGMOD international conference on Management of data (pp. 106-117).

[5] Deshpande, A., Ives, Z., & Raman, V. (2007). Adaptive query processing. Now Publishers Inc.

[6] Ioannidis, Y. E. (1996). Query optimization. ACM Computing Surveys (CSUR), 28(1), 121-123.

[7] Chaudhuri, S. (1998, May). An overview of query optimization in relational systems. In Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (pp. 34-43).

[8] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., & Price, T. G. (1979, May). Access path selection in a relational database management system. In Proceedings of the 1979 ACM SIGMOD international conference on Management of data (pp. 23-34).

[9] Graefe, G. (1993). Query evaluation techniques for large databases. ACM Computing Surveys (CSUR), 25(2), 73-169.

[10] Stonebraker, M., & Rowe, L. A. (1986). The design of Postgres. ACM Sigmod Record, 15(2), 340-355.

[11] Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., ... & Zhang, T. (2017, January). Self-Driving Database Management Systems. In CIDR (Vol. 4, p. 1).

[12] Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. Computer, 36(1), 41-50.

[13] Hellerstein, J. L., Diao, Y., Parekh, S., & Tilbury, D. M. (2004). Feedback control of computing systems. John Wiley & Sons.

[14] Ma, L., Van Aken, D., Hefny, A., Mezerhane, G., Pavlo, A., & Gordon, G. J. (2018, May). Query-based workload forecasting for self-driving database management systems. In Proceedings of the 2018 International Conference on Management of Data (pp. 631-645).

[15] Van Aken, D., Pavlo, A., Gordon, G. J., & Zhang, B. (2017, May). Automatic database management system tuning through large-scale machine learning. In Proceedings of the 2017 ACM international conference on management of data (pp. 1009-1024).

[16] Zhang, B., Van Aken, D., Wang, J., Dai, T., Jiang, S., Lao, J., ... & Gordon, G. J. (2018). A demonstration of the ottertune automatic database management system tuning service. Proceedings of the VLDB Endowment, 11(12), 1910-1913.

[17] Kraska, T., Beutel, A., Chi, E. H., Dean, J., & Polyzotis, N. (2018, May). The case for learned index structures. In Proceedings of the 2018 international conference on management of data (pp. 489-504).

[18] Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., ... & Tatbul, N. (2019). Neo: A learned query optimizer. arXiv preprint arXiv:1904.03711.

[19] Ioannidis, Y. E., Ng, R. T., Shim, K., & Sellis, T. K. (1997). Parametric query optimization. The VLDB Journal, 6(2), 132-151.

[20] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., & Wilkes, J. (2013, April). Omega: flexible, scalable schedulers for large compute clusters. In Proceedings of the 8th ACM European Conference on Computer Systems (pp. 351-364).

[21] Graefe, G. (1995). The cascades framework for query optimization. IEEE Data Eng. Bull., 18(3), 19-29.

[22] Yoon, D. Y., Mozafari, B., & Brown, D. P. (2015). DBSeer: Pain-free database administration through workload intelligence. Proceedings of the VLDB Endowment, 8(12), 2036-2039.

[23] Marcus, R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M., & Kraska, T. (2021, June). Bao: Making learned query optimization practical. In Proceedings of the 2021 International Conference on Management of Data (pp. 1275-1288).

[24] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., & Wilkes, J. (2015, April). Large-scale cluster management at Google with Borg. In Proceedings of the tenth european conference on computer systems (pp. 1-17).

[25] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, omega, and kubernetes. Communications of the ACM, 59(5), 50-57.