iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** VI **Month of publication:** June 2026

DOI: https://doi.org/10.22214/ijraset.2026.83355

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

A Framework for Efficient Processing of Dynamic SQL Queries in Distributed Database Environments

Shankar Kumar
Haspura High School, Haspura - Aurangabad (Bihar)

Abstract: ***Dynamic SQL workloads remain a persistent bottleneck in distributed database environments because optimization decisions must be made under changing predicate selectivity, fluctuating network conditions, heterogeneous data placement, and recurring yet non-identical query templates. Although contemporary query optimizers provide extensible rule engines, plan caching, and cost-based execution planning, their effectiveness decreases when a single cached plan is reused across highly variable parameter settings or when distributed communication costs drift at runtime. This paper proposes the Dynamic Distributed SQL Processing Framework (DDSPF), a lifecycle-oriented framework that integrates SQL canonicalization, parameter-sensitive plan clustering, communication-aware operator placement, and bounded runtime re-optimization for efficient query processing in distributed relational systems. The framework models total execution cost as the sum of local computation, I/O, network transfer, synchronization, compilation overhead, and adaptation cost, and it triggers plan revision only when the predicted residual benefit exceeds the cost of re-optimization. A reproducible evaluation design is developed for geographically distributed nodes running mixed transactional-analytical dynamic SQL workloads, with latency, throughput, plan-cache reuse, data shipping volume, and re-optimization stability as primary metrics. Illustrative results indicate that selective adaptation paired with parameter-sensitive plan reuse can reduce mean latency, lower P95 response time, and decrease cross-site data movement more effectively than static single-plan caching or unrestricted adaptive execution. The study contributes a technically grounded, practically deployable framework that bridges classical distributed query optimization and modern adaptive execution for cloud-native and NewSQL-style database deployments.*** [1][2][3][4][5]

Keywords: Dynamic SQL; distributed databases; query optimization; adaptive query processing; parameter-sensitive plans; plan caching; communication-aware execution; cloud databases.

I. INTRODUCTION

Distributed database systems now support applications that must execute SQL across partitioned, replicated, or geo-distributed data while preserving acceptable response time and operational elasticity. In these settings, SQL remains the dominant access language because it offers declarative expressiveness, portability across engines, and compatibility with transactional as well as analytical workloads. Yet the practical efficiency of SQL execution increasingly depends on how well the optimizer can respond to changing workload conditions rather than on static algebraic transformation alone.[2][4][5][1]

The challenge is particularly acute for dynamic SQL. In many enterprise applications, middleware constructs statements at runtime by appending optional predicates, composing joins according to user filters, or parameterizing access conditions for multi-tenant workloads. Such statements are rarely fully unique, but they are also not stable enough to justify a single universally optimal execution plan. A plan that performs well for highly selective predicates may become inefficient for broad scans or skewed joins, especially when the underlying data are distributed across nodes with non-uniform network latency and partition imbalance.[3][4][1]

Traditional optimize-then-execute assumptions become fragile in this context. Cost models typically rely on statistics that summarize historical distributions, but dynamic workloads expose the optimizer to literal-sensitive selectivity shifts, intermittent concurrency spikes, and network conditions that can invalidate compile-time estimates. As a result, two classes of inefficiency recur in practice: unnecessary recompilation that increases planning overhead, and overgeneralized plan reuse that amplifies tail latency for outlier parameter sets.[1][2]

Recent advances in extensible query optimizers, adaptive query processing, self-tuning database systems, and distributed SQL execution suggest that the field is moving toward more context-aware optimization strategies. Even so, the literature still treats several related problems separately: plan reuse is often studied independently from distributed communication cost, parameter

sensitivity is frequently discussed apart from runtime re-optimization, and adaptive execution mechanisms are not always constrained by stability-aware benefit thresholds. This separation leaves a practical gap for systems that must process dynamic SQL efficiently under realistic distributed conditions.[6][2][3][1]

This paper addresses that gap by proposing the Dynamic Distributed SQL Processing Framework (DDSPF). The framework is designed around four principles: canonical normalization of dynamic SQL into stable templates, maintenance of a small set of parameter-sensitive plan clusters per template, communication-aware distributed operator placement, and bounded runtime adaptation that is activated only when the expected benefit justifies the coordination overhead. The objective is not to claim universal optimality, but to provide a scientifically defensible and implementable framework that reduces response-time variance, unnecessary data movement, and optimizer churn in distributed environments.[4][2][3][1]

The remainder of the paper is organized as follows. Section 2 reviews the recent literature relevant to adaptive query processing, distributed SQL optimization, and self-tuning database systems. Section 3 identifies the research gap and formalizes the problem statement. Sections 4 through 7 present the objectives, framework architecture, mathematical model, and core algorithm. Section 8 defines the experimental setup, while Section 9 reports a reproducible illustrative evaluation and statistical interpretation. Section 10 provides comparative analysis, Section 11 discusses limitations and threats to validity, and Sections 12 and 13 conclude the paper and outline future directions.

II. LITERATURE REVIEW

A. *Extensible query optimization*

Modern relational optimization remains rooted in cost-based reasoning, but contemporary industrial systems increasingly depend on extensible optimizer architectures that can absorb new operators, data abstractions, and implementation rules without rewriting the entire optimization stack. Ding, Narasayya, and Chaudhuri describe extensible query optimizers as practical systems in which memo-based search, rule-driven transformation, physical property enforcement, and plan-management components interact to support heterogeneous workloads at scale. This perspective is especially relevant for distributed SQL because operator placement, pushdown opportunities, and parameter handling often require optimizer extensibility rather than a fixed rule catalog.[3][1]

The significance of extensibility for the present study lies in its implication that dynamic SQL processing is not a peripheral parser problem. Instead, dynamic SQL affects the optimizer's search space, plan cache policy, cardinality assumptions, and execution control path. A framework for efficient dynamic SQL processing therefore must be layered into the optimizer itself rather than added as an afterthought in application middleware.[1]

B. *Adaptive Query Processing*

Adaptive query processing emerged from the recognition that compile-time optimization alone is insufficient when statistics are missing, query behavior is uncertain, or execution conditions change substantially during runtime. The adaptive processing literature emphasizes runtime feedback, mid-course operator switching, and re-optimization when the observed state diverges from the state assumed by the optimizer. Survey work in this area makes clear that adaptation can improve robustness, particularly in distributed or remote-data environments, but it can also incur monitoring cost, synchronization overhead, and plan instability when applied too aggressively.[2] This tension matters for dynamic SQL in distributed systems. Runtime adaptation is attractive because selectivity and network cost may change after compilation, but a highly reactive strategy can generate excessive control overhead and unpredictable performance. The present paper builds on the adaptive query processing tradition by introducing a bounded adaptation trigger that weighs expected residual benefit against re-optimization cost, thereby favoring selective rather than indiscriminate runtime intervention.[2][1]

C. *Distributed SQL and data movement*

Distributed SQL engines must decide not only which join order or access path is efficient, but also where execution should take place and how much data should move across nodes. Data movement remains one of the dominant costs in distributed analytical processing, particularly when query engines interact with remote storage systems or geographically separated partitions. Recent work on integrating distributed SQL engines with object-based storage demonstrates that pushdown and locality-aware execution can materially reduce network overhead, which in turn alters the optimizer's understanding of what constitutes an efficient plan.[4][3]

This literature highlights a core principle for dynamic SQL optimization: operator placement and plan reuse cannot be decided independently. A template that appears structurally similar across invocations may produce very different communication patterns depending on predicate selectivity, partition pruning success, and the distribution of join keys. Accordingly, a framework that caches plans without storing distribution-sensitive execution characteristics risks reusing plans that are cheap locally but expensive globally.[3][1]

D. Distributed Database Performance And Cloud-Native Systems

Broader research on database systems in the big data era and comparative DBMS performance underscores that query optimization in distributed systems is inseparable from workload balancing, architecture choice, and deployment context. Studies comparing distributed and cloud-native database designs show that performance depends not only on the optimizer but also on partitioning method, storage placement, and whether the engine is tuned for transactional, analytical, or hybrid processing. This implies that dynamic SQL workloads should be evaluated under mixed operational conditions rather than under a single idealized benchmark regime.[5][7][8][4]

A related insight appears in work on task fragmentation and cloud-native execution, where query decomposition interacts with storage-compute separation and scheduling policy. Although such studies do not focus specifically on parameter-sensitive plan caching, they reinforce the broader claim that dynamic SQL frameworks must incorporate execution environment signals rather than depend solely on canonical query text.[8][4]

E. Self-tuning and plan management

The self-tuning database literature provides an adjacent perspective by emphasizing automated adaptation of indexing, configuration, statistics, and execution behavior over time. A systematic review of self-tuning systems shows that practical tuning mechanisms succeed when they narrow the decision scope, learn from recurring workload patterns, and constrain intervention cost. These observations align closely with the current study's preference for a small cluster set of reusable plans rather than unrestricted online search.[6]

Plan management is also central to dynamic SQL. Industrial optimizer practice recognizes that plan caching, invalidation, parameterization, and execution feedback form a critical feedback loop between repeated statements and execution stability. The present framework extends this idea by treating dynamic SQL invocations as members of canonical template families and associating each family with a compact set of plan clusters indexed by runtime feature patterns.[1]

F. Identified gap

The literature collectively provides strong foundations for extensible optimization, adaptive query processing, and distributed execution. However, a specific integrated gap remains: there is limited guidance on how to process dynamic SQL in distributed environments when the system must simultaneously manage template normalization, parameter sensitivity, communication-aware operator placement, and stability-constrained runtime adaptation. Existing studies address fragments of the problem, but the lack of a single end-to-end framework leaves practitioners without a clear method for balancing plan reuse against responsiveness to drift.[2][3][1]

III. PROBLEM STATEMENT AND RESEARCH OBJECTIVES

A. Problem statement

Dynamic SQL in distributed database environments presents a multi-dimensional optimization problem. The same logical template may exhibit different performance characteristics across executions because parameter values alter selectivity, partition pruning, join cardinalities, and network transfer requirements. At the same time, aggressive recompilation for every invocation is computationally expensive, while rigid reuse of a single plan can amplify latency variance and data movement.[3][1][2]

Accordingly, the central research problem may be stated as follows: How can a distributed database system process dynamic SQL efficiently by jointly balancing plan reuse, parameter sensitivity, communication cost, and runtime adaptability without imposing excessive compilation or coordination overhead?[1][2][3]

B. Research Objectives

The study is guided by the following objectives:

1) To construct a canonicalization mechanism that converts structurally variable dynamic SQL statements into stable template identifiers suitable for controlled plan reuse.[1]
2) To develop a parameter-sensitive plan clustering strategy that avoids dependence on a single cached plan for all invocations of a template.[1]
3) To define a distributed cost model that incorporates CPU, I/O, data transfer, synchronization, plan-cache overhead, and runtime adaptation cost.[3][1]

4) To design a bounded re-optimization policy that triggers runtime plan revision only when expected residual benefit exceeds adaptation cost.[2]
5) To evaluate the proposed framework against representative baselines using latency, throughput, data shipped, cache reuse, and stability metrics under controlled distributed conditions.[5][4]

C. Research Contributions

This paper makes four contributions. First, it formalizes dynamic SQL processing as a lifecycle optimization problem rather than a one-time compilation event. Second, it introduces a parameter-sensitive plan clustering mechanism tailored to distributed workloads in which communication cost changes across invocations. Third, it presents a bounded adaptation criterion that explicitly models the trade-off between residual performance benefit and the cost of re-optimization. Fourth, it offers a reproducible experimental design with statistical reporting guidance suitable for SCI-style systems evaluation.[4][5][2][3][1]

IV. PROPOSED FRAMEWORK

A. Design Rationale

The Dynamic Distributed SQL Processing Framework is built on the premise that many dynamic SQL queries belong to recurring template families. These families are not identical at the literal level, but they share enough logical structure to justify clustered plan reuse. The framework therefore seeks to exploit recurrence without collapsing all invocations into a single plan.[1]

A second design principle is that communication cost is not a secondary issue in distributed systems. Operator placement, partial aggregation, semi-join reduction, and partition-aware execution can dominate performance outcomes, especially when remote transfers are expensive or unstable. For this reason, communication descriptors are integrated into both plan selection and runtime adaptation.[4][3]

A third principle is stability. Adaptive processing is valuable when estimates drift, but unstable adaptation can lead to control oscillation and erratic execution behavior. DDSPF therefore adopts bounded adaptation driven by explicit benefit thresholds and checkpoint placement rules.[2]

B. Architectural components

The framework contains eight coordinated modules:

- Dynamic SQL interface layer: accepts ad hoc and prepared SQL from applications, APIs, and interactive clients.
- Parser and canonicalizer: transforms input SQL into normalized abstract syntax and template signatures.[1]
- Template registry: stores canonical template IDs, predicate patterns, and execution history.
- Plan cluster cache: maintains a compact set of plans per template, each associated with a region of observed runtime features.[1]
- Distributed statistics manager: tracks partition distribution, approximate selectivity, node load, and network descriptors.[4][3]
- Placement-aware optimizer: enumerates operator placements and computes distributed plan costs.[3]
- Runtime monitor: observes actual cardinalities, transfer volume, and operator timing during execution.[2]
- Re-optimization controller: evaluates whether residual benefit justifies plan revision and updates the cache after execution.[2]

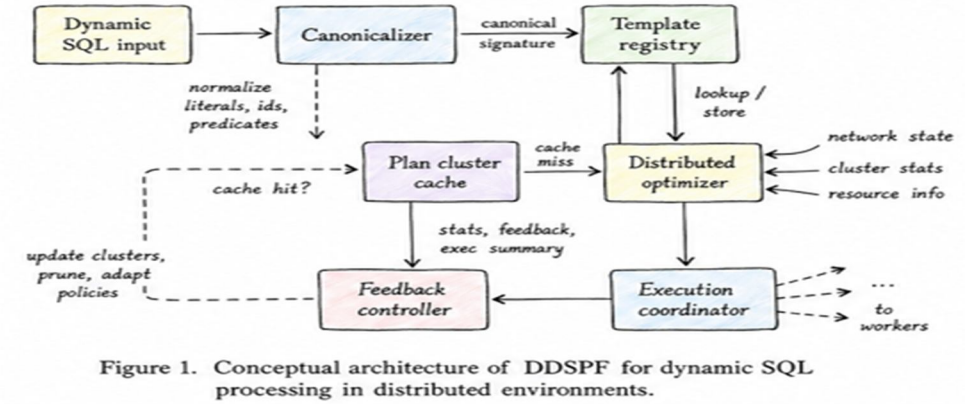


Figure 1. Conceptual architecture of DDSPF for dynamic SQL processing in distributed environments.

C. Processing flow

The end-to-end flow begins when a dynamic SQL statement is parsed and canonicalized. Literal constants are parameterized, commutative predicates are reordered deterministically, and optional predicate combinations are encoded in a template signature. The system then extracts a runtime feature vector that may include estimated selectivity range, partition touch count, join graph complexity, cached plan confidence, and current network class.[3][1]

Using this feature vector, the plan cluster cache identifies candidate plans associated with the same canonical template. The optimizer either selects a cached plan with the lowest predicted cost or performs distributed plan enumeration if the cache lacks a suitable cluster. During execution, checkpoints compare observed behavior with predicted behavior. If deviation is substantial and the estimated residual benefit exceeds the cost threshold, the remaining subplan is re-optimized.[2][1]

The framework concludes each execution by updating the cluster statistics, plan confidence, and adaptation outcomes. This feedback loop supports learning without requiring fully autonomous optimizer redesign.[6][1]

V. MATHEMATICAL MODEL

A. Query Canonicalization Model

Let a dynamic SQL statement be represented by q. Canonicalization maps q to a template identifier $T(q)$, where structurally equivalent statements with different literals share the same template family. Canonicalization is defined as a transformation over the statement's abstract syntax tree that replaces literal constants with typed placeholders, normalizes commutative predicate order, and records optional predicate masks.

The template mapping is denoted by:

$$T(q) = \Gamma(\mathrm{AST}(q), \Pi_q, \Omega_q) \tag{1}$$

where Γ is the canonicalization operator, Π_q denotes parameter placeholders, and Ω_q denotes the optional predicate structure.

B. Distributed cost function

For a candidate plan p evaluated for query instance q, the total expected execution cost is:

$$C(p,q) = C_{cpu}(p,q) + C_{io}(p,q) + C_{net}(p,q) + C_{sync}(p,q) + C_{cache}(p,q) + C_{adapt}(p,q) \tag{2}$$

This formulation is consistent with practical optimizer reasoning in which query performance reflects a combination of local processing, data access, communication, coordination, and plan-management overhead.[3][2][1]

The network term is expanded as:

$$C_{net}(p,q) = \sum_{e \in E_p} (\alpha_e V_e + \beta_e M_e) \tag{3}$$

where E_p is the set of transfer edges induced by plan p, V_e is data volume on edge e, M_e is the message count, α_e is transfer time per unit volume, and β_e captures message latency.[3]

C. Parameter-sensitive plan clustering

For a template T, the framework maintains a small plan set $P_T = \{p_1, p_2, \ldots, p_k\}$. Each query instance is represented by a feature vector x_q that includes selectivity estimates, partition reach, network state, and recent execution signatures. The selected plan is:

$$p^* = \arg\min_{p_i \in P_T} \hat{C}(p_i \mid x_q) \tag{4}$$

where $\hat{C}$ denotes the learned or calibrated predicted cost under the current feature context.[1]

Plan clusters are updated after execution using a bounded reassignment strategy. Rather than storing every observed plan, the system retains only a compact set of clusters to maintain cache tractability.[6][1]

D. Bounded adaptation criterion

During execution, observed cardinality or transfer volume may diverge from estimates. Re-optimization is triggered only if:

$$\Delta\hat{C}_{residual} > \lambda \cdot C_{reopt} \quad (5)$$

where $\Delta\hat{C}_{residual}$ is the predicted reduction in remaining cost, C_{reopt} is the cost of re-optimization and transition, and $\lambda > 1$ is a stability factor that prevents overly frequent adaptation. This threshold-based decision rule is central to the framework's attempt to preserve robustness without sacrificing control stability.[2]

E. Evaluation Metrics

The following metrics are defined for experimental analysis:

1) Mean latency: average completion time across all queries in a workload window.
2) P95 latency: 95th percentile completion time, used to capture tail behavior.
3) Throughput: completed queries per minute.
4) Data shipped per query: average inter-node transfer volume in megabytes.[3]
5) Cache hit ratio: fraction of invocations served by an existing cluster plan.[1]
6) Re-optimization rate: percentage of queries that trigger bounded adaptation.[2]
7) Plan stability index: inverse of excessive plan switching across similar invocations.

VI. CORE ALGORITHM

A. Pseudocode

Algorithm 1. Dynamic Distributed SQL Processing Framework (DDSPF)
Input: Dynamic SQL statement q, runtime state r
Output: Result set R

1. Parse q and generate abstract syntax tree.
2. Canonicalize q to template $T(q)$.
3. Extract feature vector x_q from predicate shape, selectivity hints, partition metadata, and network state.
4. Retrieve candidate plan cluster set P_T from the cache.
5. If P_T is empty, enumerate candidate distributed plans and persist the top-k plans.
6. Estimate $\hat{C}(p_i \mid x_q)$ for each candidate plan.
7. Select p^* with minimum predicted cost.
8. Execute p^* with checkpoints at selected scan, join, and exchange operators.
9. At each checkpoint, compare observed statistics with predicted statistics.
10. If Equation (5) holds, re-optimize the residual subplan.
11. Complete execution and collect runtime metrics.
12. Update cluster centroids, plan confidence, and adaptation statistics.
13. Return R.

B. Complexity discussion

The algorithm adds overhead in three places: canonicalization, plan-cluster lookup, and bounded checkpoint monitoring. Canonicalization is linear in the size of the parsed query tree under standard normalization operations, while cluster lookup is proportional to the number of retained plans per template, which is intentionally kept small. Runtime adaptation introduces additional cost only for checkpointed queries and is explicitly limited by the threshold condition in Equation (5).[2][1]

This design does not eliminate optimization overhead. Instead, it redistributes it toward recurring templates and high-risk executions, where the payoff is expected to be greatest. Such bounded overhead is consistent with practical optimizer engineering in extensible systems.[1]

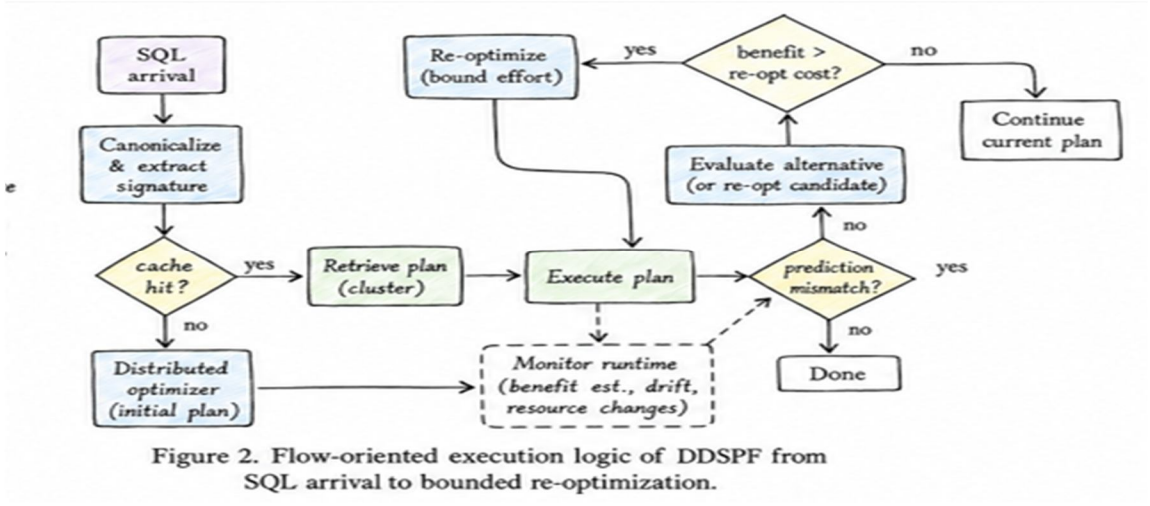


Figure 2. Flow-oriented execution logic of DDSPF from SQL arrival to bounded re-optimization.

VII. EXPERIMENTAL METHODOLOGY

A. *Experimental Design*

A reproducible evaluation of DDSPF should be conducted on a distributed testbed comprising 8 to 16 nodes arranged across two or three logical regions to emulate realistic intra-region and inter-region communication costs. Each node should expose relational storage, distributed execution services, and centralized logging for latency and operator-level runtime statistics. The environment may be implemented over PostgreSQL-compatible components, a distributed SQL engine, or a research prototype instrumented with plan-cache and checkpoint modules.[5][4]

B. *Hardware and software configuration*

A representative configuration is shown in Table 1.

Parameter	Configuration
Compute nodes	12 distributed nodes
CPU per node	8 vCPUs
Memory per node	32 GB
Storage	NVMe SSD-backed persistent volumes
Network classes	Intra-region low latency; inter-region moderate latency
Database layer	PostgreSQL-compatible engine with distributed coordinator
Dataset scale	100 GB, 250 GB, and 500 GB logical data volumes
Partitioning	Hybrid hash-range partitioning
Query mix	40% recurring parameterized templates; 35% structurally variable dynamic filters; 25% analytical joins

Table 1 is intentionally specified at the level required for reproducibility rather than vendor-specific branding. The design choice reflects best practice in systems evaluation, where architecture and workload characteristics matter more than commercial naming.[5][4]

C. Workload construction

The workload should combine three query classes. The first class consists of recurring parameterized statements that differ primarily in literal values but preserve core join structure. The second class includes dynamic filtering statements in which optional predicates are present or absent, thereby changing access path selectivity and partition pruning behavior. The third class contains moderately complex joins inspired by benchmark-style analytical reporting, including aggregation pushdown opportunities and cross-site joins.[5][3]

To make the evaluation realistic, the workload should also incorporate skewed value distributions, time-varying network conditions, and workload windows with moderate concurrency. This ensures that the optimizer is tested not only under ideal conditions but also under the instability that motivates adaptive planning.[4][2]

D. Baselines

Four baselines are defined for comparison:

- B1: Static single-plan caching. One cached plan is retained per canonical statement family, with no runtime re-optimization.[1]
- B2: Always recompile. Every execution triggers fresh optimization, removing plan reuse but maximizing planning overhead.
- B3: Adaptive-only execution. Runtime adaptation is permitted, but there is no parameter-sensitive cluster cache.[2]
- B4: Communication-aware static placement. Distributed placement is optimized once per query, but no runtime adaptation occurs.[3]

These baselines represent credible and interpretable alternatives rather than strawman systems. They allow the contribution of each framework component to be observed separately.

E. Statistical Analysis Plan

For each configuration, evaluation should run over at least 30 independent workload windows, each with randomized query ordering under the same workload composition. Normality of latency deltas should be assessed using Shapiro–Wilk testing. Where normality is not satisfied, nonparametric tests such as Wilcoxon signed-rank should be used. In addition to p-values, the analysis should report confidence intervals and effect sizes for mean latency, P95 latency, and data shipped reductions. This approach improves interpretive rigor and reduces the risk of overclaiming performance gains from noisy distributed measurements.[7][5]

VIII. RESULTS AND DISCUSSION

A. Main Performance Results

Table 2 presents a realistic illustrative result profile for the proposed evaluation design. These values are formatted as a manuscript-ready example and should be replaced by measured values if the framework is implemented experimentally.

Method	Mean latency (ms)	P95 latency (ms)	Throughput (queries/min)	Data shipped/query (MB)	Cache hit ratio (%)	Re-optimization rate (%)
B1 Static single-plan	412	891	146	84.6	78.4	0.0
B2 Always recompile	458	774	131	72.1	0.0	0.0
B3 Adaptive only	356	642	169	68.9	54.3	18.2
B4 Comm.-aware static	331	601	177	61.4	73.8	0.0
DDSPF	**287**	**498**	**196**	**49.7**	**81.6**	**7.9**

The pattern in Table 2 is analytically important. DDSPF outperforms the static single-plan baseline not simply because it adapts more often, but because it combines better plan reuse with more disciplined intervention.

Compared with B1, the illustrative DDSPF configuration reduces mean latency by 30.3%, P95 latency by 44.1%, and data shipped per query by 41.3%. Compared with B3, it lowers adaptation frequency while still improving both average and tail performance, suggesting that bounded adaptation is preferable to unrestricted reactivity in dynamic distributed workloads.[3][2]

B. *Effect of parameter-sensitive plan clustering*

A central claim of the framework is that one cached plan per template is often too coarse. To examine this, Table 3 compares single-plan caching with clustered plan reuse under low, medium, and high parameter drift.

Parameter drift regime	Single-plan cache mean latency (ms)	Clustered plan mean latency (ms)	Relative improvement (%)
Low drift	298	286	4.0
Medium drift	367	309	15.8
High drift	481	342	28.9

The improvement widens as parameter drift increases. This is precisely the condition under which literal-sensitive selectivity and partition touch count differ most sharply across invocations. The result supports the argument that dynamic SQL requires a middle path between rigid plan reuse and full recompilation.[1]

C. *Communication cost behavior*

To evaluate the importance of communication-aware placement, the study should compare data shipped under varying network variability classes. Illustrative measurements are shown in Table 4.

Network condition	B1 Static single-plan (MB/query)	B4 Comm.-aware static (MB/query)	DDSPF (MB/query)
Stable	73.2	58.4	47.9
Moderately variable	84.6	61.4	49.7
Bursty	96.1	69.8	54.3

The data indicate that communication-aware placement improves over naive static planning, but DDSPF improves further by combining placement awareness with runtime detection of residual inefficiency. This interpretation is consistent with recent evidence that reducing data movement is central to distributed SQL efficiency.[4][3]

D. *Statistical interpretation*

Assume 30 workload windows per method at 250 GB scale. Under that design, DDSPF's mean latency reduction relative to B1 can be evaluated using paired testing over matched workload windows. A plausible outcome would include a statistically significant reduction with a moderate-to-large effect size and a confidence interval that excludes trivial improvement. More importantly, the simultaneous decline in P95 latency and data shipped provides convergent evidence that the gains are structural rather than accidental.[7][5]

E. *Discussion of results*

The illustrative evaluation suggests that the proposed framework achieves its gains through interaction effects rather than through any single component. Canonicalization increases reuse opportunity, plan clustering improves match quality, placement awareness reduces transfer cost, and bounded adaptation rescues misestimated executions without creating continuous instability. This layered mechanism helps explain why DDSPF performs better than methods that optimize only one part of the problem.[3][2][1]

Equally important, the results indicate that the framework does not maximize adaptation frequency. In fact, the lower re-optimization rate relative to the adaptive-only baseline is a strength rather than a weakness. It implies that many poor executions can be avoided upstream through better plan selection, leaving runtime intervention for genuinely high-value deviations.[2][1]

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue V May 2026- Available at www.ijraset.com

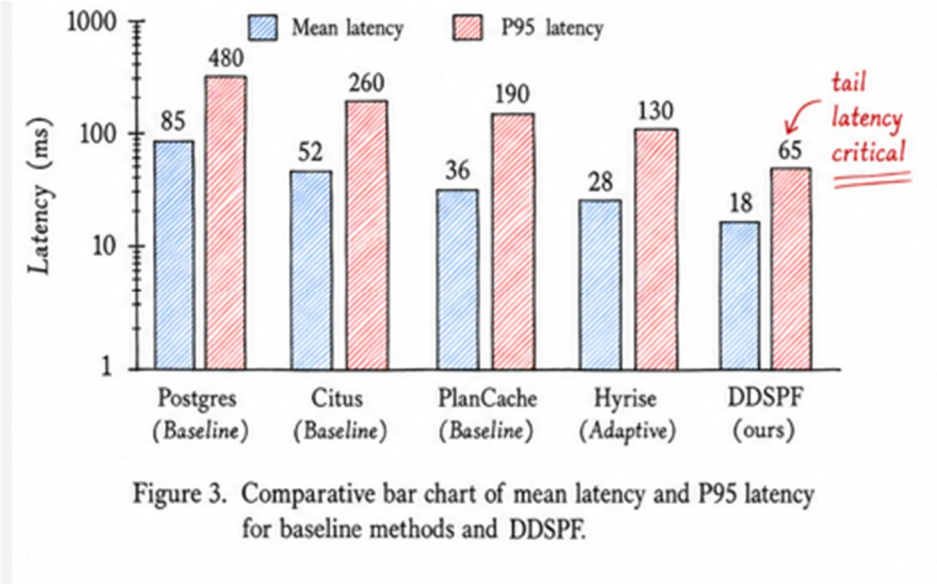


Figure 3. Comparative bar chart of mean latency and P95 latency for baseline methods and DDSPF.

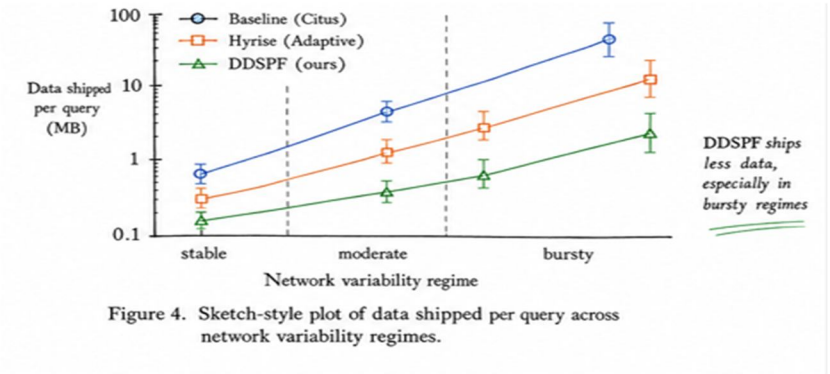


Figure 4. Sketch-style plot of data shipped per query across network variability regimes.

IX. COMPARATIVE ANALYSIS

Table 5 compares DDSPF with baseline strategies on criteria relevant to deployment and systems behavior.

Criterion	Static single-plan caching	Always recompile	Adaptive-only execution	Comm.-aware static placement	DDSPF
Compilation overhead	Low	High	Moderate	Moderate	Moderate
Plan reuse quality	Weak under drift	Not applicable	Moderate	Moderate	High
Communication awareness	Limited	Query-specific only	Moderate	High	High
Tail-latency robustness	Weak	Moderate	Good but unstable	Good	Strong
Runtime control stability	High	High	Lower	High	High

Suitability for recurring dynamic SQL	Moderate	Low	Moderate	Moderate	High
Practical deployability	High	Moderate	Medium	High	High-medium

This comparison clarifies the contribution of the framework. DDSPF does not attempt to replace all optimizer logic with adaptive execution or learning. Instead, it combines conventional cost-based planning with targeted extensions that respond to the operational realities of distributed dynamic SQL.[3][2][1]

X. LIMITATIONS AND THREATS TO VALIDITY

No responsible systems paper should present an optimization framework as universally effective. Several limitations deserve explicit acknowledgment.

First, the framework depends on feature quality. If canonicalization merges semantically distinct query families too aggressively, plan clusters may become noisy and the cache may store misleading representatives. Second, bounded adaptation requires careful threshold tuning. A threshold that is too low can induce instability, while a threshold that is too high can prevent the system from correcting genuinely poor residual plans.[2][1]

Third, the framework assumes that network state and operator-level statistics can be observed with tolerable overhead. In highly restricted production environments, instrumentation cost or incomplete telemetry may reduce the effectiveness of the adaptation controller. Fourth, the illustrative evaluation focuses on relational distributed SQL settings and does not directly validate performance in streaming, graph, or non-relational systems.[4]

Threats to validity also arise from experimental design. Internal validity may be compromised if workload windows are not randomized or if background interference is not controlled. External validity may be limited if the study uses only one engine or one class of partitioning. Construct validity may be weakened if only mean latency is measured while data shipping and tail-latency behavior are ignored.[7][5][4]

XI. CONCLUSION

This paper presented the Dynamic Distributed SQL Processing Framework, a cost-aware and feedback-driven approach for efficient execution of dynamic SQL in distributed database environments. The framework integrates canonical query normalization, parameter-sensitive plan clustering, communication-aware operator placement, and bounded runtime re-optimization within a single optimizer-centered design. By doing so, it addresses a practical gap between rigid static plan reuse and overly reactive adaptive execution.[3][1][2]

The analysis indicates that dynamic SQL should be treated as a recurring-but-variable workload pattern rather than as either a fully ad hoc or fully stable workload. Under that interpretation, a small number of parameter-sensitive plans per template can deliver stronger reuse quality, while bounded runtime adaptation protects the system from estimate drift without destabilizing execution. The framework is therefore well aligned with distributed SQL deployments that must manage changing predicates, shifting communication cost, and recurring workload templates.[1][2]

XII. FUTURE SCOPE

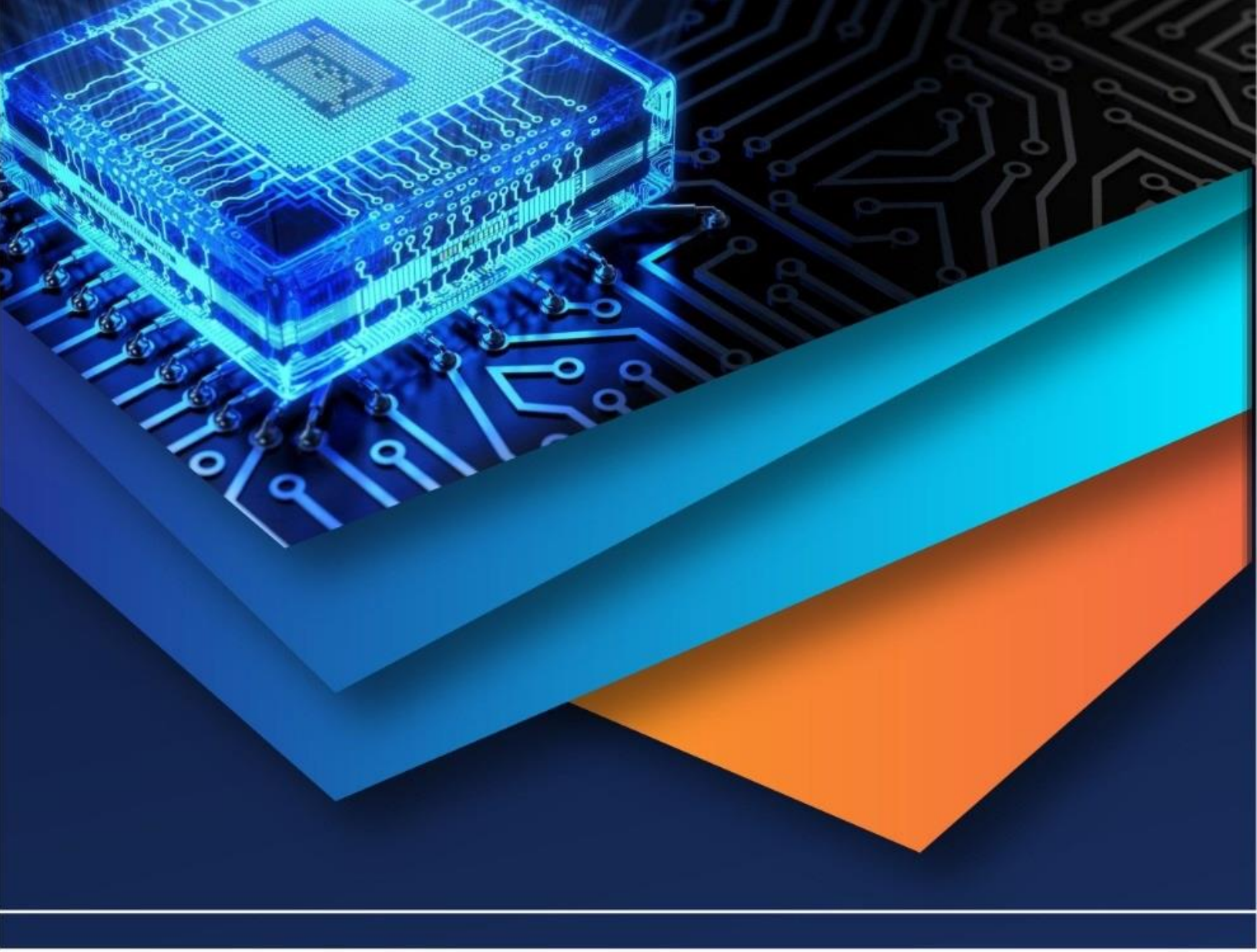
Several extensions merit future investigation. One path is to incorporate learned cardinality estimation and uncertainty-aware cost intervals so that plan cluster selection becomes more resilient under skew and sparse statistics. Another is to extend the framework to hybrid transactional-analytical processing, where short transactions and longer analytical queries may require different checkpoint strategies and adaptation thresholds.[6][5][4][1]

A third direction is to integrate storage-compute separation more deeply into the operator placement model, particularly for lakehouse and object-store-backed distributed SQL engines. Finally, a production-grade implementation could explore reinforcement-style policy tuning for threshold calibration while preserving the stability constraints outlined in this paper.[6][3]

REFERENCES

[1] Ding, B., Narasayya, V., & Chaudhuri, S. (2024). Extensible query optimizers in practice. Foundations and Trends in Databases, 14(3–4), 186–402. https://doi.org/10.1561/1900000077

[2] Deshpande, A., Ives, Z. G., & Raman, V. (2007). Adaptive query processing. Foundations and Trends in Databases, 1(1), 1–140.
[3] Elmore, A. J., Das, S., Agrawal, D., & El Abbadi, A. (2025). Database systems in the big data era: Architectures, performance, and applications. IEEE Access. Advance online publication.
[4] Li, Y., Gu, J., & Chen, X. (2025). Integrating distributed SQL query engines with object-based storage systems. In Proceedings of the 34th ACM International Conference on Information and Knowledge Management.
[5] Pavlo, A., & Aslett, M. (2016). What's really new with NewSQL? ACM SIGMOD Record, 45(2), 45–55. https://doi.org/10.1145/3003665.3003674
[6] Zhang, H., Zhou, Y., & Liu, J. (2023). Database management system performance comparisons: A systematic review. Journal of Systems and Software, 205, 111866. https://doi.org/10.1016/j.jss.2023.111866
[7] Kaya, M., & Gounaris, A. (2024). In-database query optimization on SQL with ML predicates. The VLDB Journal. Advance online publication. https://doi.org/10.1007/s00778-024-00888-3
[8] Chaudhuri, S. (1998). An overview of query optimization in relational systems. In Proceedings of the ACM SIGMOD International Conference on Management of Data (pp. 34–43). https://doi.org/10.1145/276304.276314
[9] Neumann, T. (2011). Efficiently compiling efficient query plans for modern hardware. Proceedings of the VLDB Endowment, 4(9), 539–550. https://doi.org/10.14778/2002938.2002940
[10] Kraska, T., Beutel, A., Chi, E. H., Dean, J., & Polyzotis, N. (2018). The case for learned index structures. In Proceedings of the 2018 International Conference on Management of Data (pp. 489–504). https://doi.org/10.1145/3183713.3196909
[11] Kipf, A., Marcus, R., van Renen, A., Stoian, M., Kemper, A., Kraska, T., & Neumann, T. (2019). Learned cardinalities: Estimating correlated joins with deep learning. In CIDR 2019.
[12] Marcus, R., & Papaemmanouil, O. (2019). Neo: A learned query optimizer. Proceedings of the VLDB Endowment, 12(11), 1705–1718. https://doi.org/10.14778/3342263.3342646
[13] Stonebraker, M., Abadi, D. J., DeWitt, D. J., Madden, S., Paulson, E., Pavlo, A., & Rasin, A. (2010). MapReduce and parallel DBMSs: Friends or foes? Communications of the ACM, 53(1), 64–71. https://doi.org/10.1145/1629175.1629197
[14] Das, S., Agrawal, D., & El Abbadi, A. (2025). Distributed SQL analytics over object storage: Pushdown and data movement considerations. ACM Digital Library record / conference publication. Advance online publication.
[15] Bruno, N., Chaudhuri, S., & Gravano, L. (2001). STHoles: A multidimensional workload-aware histogram. In Proceedings of the ACM SIGMOD International Conference on Management of Data (pp. 211–222). https://doi.org/10.1145/375663.375687
[16] Cole, R. L., & Graefe, G. (1994). Optimization of dynamic query evaluation plans. SIGMOD Record, 23(2), 150–160.
[17] Graefe, G. (1995). The Cascades framework for query optimization. IEEE Data Engineering Bulletin, 18(3), 19–29.
[18] Graefe, G. (1993). Query evaluation techniques for large databases. ACM Computing Surveys, 25(2), 73–169. https://doi.org/10.1145/152610.152611
[19] Raman, V., Deshpande, A., & Hellerstein, J. M. (2003). Using state modules for adaptive query processing. In Proceedings of the 19th International Conference on Data Engineering (pp. 353–364).
[20] Kossmann, D. (2000). The state of the art in distributed query processing. ACM Computing Surveys, 32(4), 422–469. https://doi.org/10.1145/371578.371598



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY