Scholar Publishing Group
International Journal of Multimedia Computing
https://doi.org/10.38007/IJMC.2026.070205
ISSN 2789-7168 Vol. 7, Issue 2:38-47

Cross Shard Query Optimization Scheme under the Strategy of Sharding and Table Partitioning in Payment System Database

Jiayue Hu

Electrical and Computer Engineering, New York University, Brooklyn, NY, 11201, USA

Keywords: Payment system; database sharding; cross-shard query; partition pruning; partial pre-aggregation; query optimization

Abstract: To tackle the problems of amplified cross-shard queries, increased tail latency and memory pressure on coordinating nodes that arise after merchant-based and time-based database partitioning in payment systems, this paper proposes an optimisation scheme consisting of deterministic shard routing, time partition pruning, intra-shard pre-aggregation, constrained parallel execution and historical feedback plan selection. A single cost model is used to include remote fan-out, network transmission, node computation and merging overhead. In a real-world test environment with 8 database shards, each having 6 monthly partitions and a total of 1.44 million payment records, the proposed scheme is compared with broadcast queries and pruned queries only, and finally with the proposed scheme. Based on the above results, accessing shards 1, 2, 4, and 8 reduces P95 latency by 53.3%, 49.9%, 50.5% and 33.9% respectively with the proposed scheme compared to the broadcast method, and the number of rows received by the coordinating node is reduced by over 97%. Research has shown that basic route pruning is not effective in reducing tail latency under a full sharding model, and therefore, a collaborative mechanism of "pruning-push-parallel-feedback" should be employed to reduce cross-shard query overhead and ensure result consistency for primary use cases such as payment detail retrieval, merchant settlement, reconciliation and summary, and risk auditing.

1 Introduction

With the continuous development of mobile payment, aggregated acquiring and platform-based clearing and settlement businesses have gradually emerged, and the payment database now needs to support a high-frequency online transaction environment and perform various complex query functions, such as order retrieval, merchant invoices, end-of-day reconciliation, refund tracking and risk auditing. When the single-database capacity, index maintenance windows and write lock contention exceed the threshold, the engineering system generally uses a horizontal scaling model of "database sharding + table sharding": distribute transaction records across multiple database nodes by using merchant_id, account_id or order_id as the sharding key; and then partition physical tables by day, month or billing period based on created_at. The model has increased write

throughput and enhanced fault isolation; however, the queries previously handled by a single node are now distributed execution processes involving routing, remote access, result transmission, sorting and aggregation, and consistency control.

Cross-shard queries for payment scenarios have severe business constraints. First, query latency will affect the backend report; it will also impact the process of refund and dispute resolution, and the tail latency of P95 and P99 is more manageable than the average. Second, the summaries of amounts, transaction counts and statuses must be verifiably consistent; performance should not be traded off for semantic reduction. Third, there is an uneven distribution of access; top merchants' marketing campaigns and batch reconciliations are frequent hot spots. Fourth, shard keys and query conditions are not always the same; for example, when searching by user, payment channel or external transaction number, the router cannot directly find the target shard. Therefore, the optimisation of cross-shard queries cannot be achieved by adding more threads or expanding the cache; it must address questions such as "which shards to access, which partitions to access, where to aggregate, what level of parallelism to use, and how to select a plan based on historical feedback" simultaneously.

This paper studies payment order databases sharded by merchant and partitioned by month, and proposes a cooperative optimisation scheme for online queries and near real-time aggregation. The main contributions are: constructing an interpretable cross-shard cost model; designing deterministic routing and partition pruning rules; pushing decomposable aggregations down to shard execution; selecting limited parallelism based on the number of active shards and load variance; ranking candidate plans using historical execution statistics; and finally, validating the effectiveness of the scheme in a real-world multi-sharded SQLite benchmark. This paper reviews the current research status, identifies problems in payment databases, proposes optimisation strategies and experiments, and finally outlines the application scope.

2. Current Status Analysis of the Research Topic

2.1 Distributed Query Plan Search and Cost Estimation

Traditional distributed query optimisation is based on relational algebraic equivalent transformation and cost estimation, but sharded replicas, network topology and connection order will significantly increase the candidate plan space. Du and others proposed an improved artificial bee colony algorithm to enhance the global search ability of distributed query plans through dynamic perturbation and genetic operators, and showed that heuristic search still has the value of reducing combinatorial explosion with an increasing number of nodes [1]. However, the payment system requires stable, interpretable, and fast rollback online decisions, and complex metaheuristic algorithms are more suitable for offline optimisation or plan candidate generation.

The production system is now applying historical feedback to adjust the base value and cost estimate more often. Shankhdhar et al. recorded the execution history in the Presto historical query optimizer and used it for statistical correction and plan selection of the following similar queries [2]. Payment systems are an example; merchant bills, end-of-day reconciliation and channel clearing frequently have periodic templates, and a low-risk feedback loop can be established through query fingerprints, shard coverage, partition span, and the actual number of rows returned.

2.2 Adaptive Connection Order and Execution Time Prediction

Complex reconciliation queries generally need to join order, refund, channel flow and settlement batch tables, and the static estimation error will be amplified after multiple-level connections. Justen and others proposed POLAR, which selects the connection order with the minimum

resistance at runtime via restricted regret tuple routing to enhance the robustness of complex connections and limit additional overhead [3]. Online adaptation does not require a complete reconstruction of the execution engine, and in payment systems, the scope of adaptation can be limited to local connection order, fragmented task sorting and merging methods, etc., without destroying transaction semantics.

Prediction of Execution Time is Required for Concurrent Scheduling and SLA Management. Wu and others have introduced a multi-stage predictor for Amazon Redshift to enhance the accuracy of workload change predictions through execution caching, instance-local models and portable global models [4]. For cross-shard payment queries, the execution time prediction should include not only the SQL text but also the number of active shards, number of partitions, hotspot coefficient, number of remote rows, and length of the coordinating node queue.

2.3 Coordination Protocol and Cross-Node Overhead Optimization

Cross-shard read/write links are frequently subject to two-phase commit, replication confirmation and network round trips. Chu and others applied the idea of query rewriting to distributed protocol optimisation, and, using rule-based decoupling and partitioning, found expansion opportunities that do not require coordination; they verified that protocol-level rewriting can substantially improve the throughput of 2PC and Paxos [5]. It can be seen that the payment query optimizer should recognise semantics such as read-only, idempotent, decomposable and monotonic aggregation, provide a safe foundation for read-only snapshots, asynchronous merging and local pre-aggregation.

Eldeeb et al. proposed Chardonnay, which reduces the cost of 2PC in disk-based distributed databases through low-latency logs, fast RPC and protocol design, showing that strong transactional semantics and high performance are not entirely contradictory [6]. However, the main bottleneck of cross-shard read-only queries is often no longer the commit protocol but rather remote fan-out and data transport; therefore, it is still necessary to compress participating nodes and returned data at the execution plan level.

Zhou and others used eBPF to push some of the distributed transaction processing logic to the kernel network path and reduced user-mode switching and network stack overhead [7]. Thus, if the query routing and result transmission are bottlenecks, further performance improvement can be achieved by network path optimisation; however, it has a high deployment complexity. First, reduce the excess traffic through logical pruning and aggregation pushdown at the payment institution level, and then consider kernel or smart card offloading.

2.4 Hotspot Tilting and High-Concurrency Scaling

Payment traffic has obvious long-tail and hot-top features. Lam and others have introduced a hybrid architecture of single-machine database and distributed database cooperation in TurboDB to alleviate cross-node competition under skewed load with performance multipliers [8]. However, this paper does not modify the storage engine but reduces the long-tail risk at the query level through a hot-top-aware task sorting, parallelism limit and timeout rollback.

Shen et al. proposed Mako, which improves the throughput of distributed transactions in geographic replication environments by decoupling execution and replication and speculative 2PC [9]; Lu et al. used fast commit, RDMA offloading and priority locks to reduce the latency of distributed transactions in HDTX [10]. Together, these studies have shown that the core of distributed system optimisation lies in reducing synchronisation periods, remote round trips and data transmission. Most of the existing work focuses on transaction commit or general query engines, and there is still a lack of lightweight, interpretable and easy-to-implement solutions for the

joint optimisation of "sharding key routing + time partitioning + decomposable aggregation" for payment databases.

3. Problem Statement: The cost amplification mechanism for cross-shard queries in payment.

3.1 Unified Cost Model

Let the execution plan of query q be p. The total cost of cross-shard queries is determined by the number of participating shards, the amount of data received by the coordinating node, the shard-side computation time and the coordinating-side merging time. To avoid one-sided optimisation based solely on network bytes or single-node CPU, the following weighted cost is used:

$$C(q,p)=\alpha N_s+\beta V_{net}+\gamma T_{cpu}+\delta T_{merge} \quad (1)$$

Ns is the number of shards or remote tasks in the execution (1), Vnet is the amount of data transmitted to the coordinating node, Tcpu is the critical path of computation time for each shard, and Tmerge is the time for sorting, deduplication, aggregation and pagination at the coordinating node; α, β, γ and δ are determined by monitoring data. Payment queries are generally sensitive to tail latency, so the slowest shard should be given a higher weight in practice.

3.2 Broadcast queries and partition mismatch

If a query does not have an explicit sharding key, or if the middleware cannot extract the sharding key from a compound predicate, then the entire database will be broadcast. Even if only a small number of shards ultimately return records, there will still be fixed overheads such as connection establishment, statement parsing, thread usage and timeout management. If all the historical tables are still being scanned after monthly table partitioning, the number of remote tasks will be about "number of shards × number of partitions". The growth of the payment order table is rapid, and a combination of broadcasting and full partition scans will lead to an increase in P95 latency.

3.3 Coordinating Node Data Transfer and Memory Amplification

Many middleware systems use the same process of "each shard returns details—coordinating node performs unified GROUP BY". For queries that count the number of transactions and amounts by merchant and status, the number of detail rows can be several hundred times larger than the final number of grouped rows. The coordinating node receives the network, stores intermediate results, conducts hash aggregation and global sorting, and under multi-tenant concurrency, a small increase in a single query can lead to memory jitter, garbage collection pauses and queuing congestion.

3.4 Nonlinear Relationship between Parallelism and Tail Delay

Increase concurrency to reduce the critical path in a multi-sharded environment; however, excessive parallelism is not always beneficial. Too many threads increase connection contention, context switching and I/O contention; if the load on the active shards is uneven, the overall completion time will still be determined by the slowest task. Full-sharded queries have no more shards for route pruning, so further increases in parallelism may worsen tail latency due to resource contention. Therefore, parallelism should be dynamically determined based on the number of tasks, historical latency, load variance and SLA.

4. Problem Solving: Cross-Shard Query Collaborative Optimization Solution

4.1 Dual pruning of fragmented routing and time partitioning

First, the optimizer normalizes the SQL to extract equality, IN, range and transitive predicates, and converts business identifiers into sharding keys. For merchant_id equality or set conditions, the target shard can be directly obtained by applying a consistent hash function; for order numbers, the sharding bit in their encoding can be extracted; and for queries without sharding keys but with global secondary indexes, candidate shards are first obtained from the index service. Then, the monthly table is filtered according to the created_at, settle_date or billing period fields. The resulting Shard Set is defined as:

$$S(q)=\{s_i \mid H(k_q)\in I_i \wedge P_t(q)\cap P_t(s_i)\neq\varnothing\} \tag{2}$$

H(kq) is the hash mapping of the query shard key in equation (2), Ii is the hash interval responsible for shard si, Pt(q) is the set of partitions mapped to the query time range, and Pt(si) is the actual time partition where the shard exists. Only tasks that meet both shard matching and partition intersection are added to the execution queue. For SQL queries without a determined route, the system will use a controlled broadcast as a fallback and record the reasons for future rule optimisation.

4.2 Decomposable Aggregation Pushdown and Narrow Result Merging

For COUNT, SUM, MIN, MAX and splittable AVGs, the optimizer pushes the aggregation to each shard, returning "group key + partial aggregation value", and the coordinating node only performs a second merge. For AVGs, the shard returns SUM and COUNT; the coordinating node calculates the global ratio; for deduplication counting, precise set shard merging can be used, or HyperLogLog can be employed in reporting scenarios where an error is acceptable. The detail row Ri, after aggregation in the shard, becomes the group result Gi, and the transmission reduction rate is defined as:

$$\eta=1-\frac{\sum_{i\in S(q)}|G_i|}{\sum_{i\in S(q)}|R_i|} \tag{3}$$

As shown in equation (3), |Ri| is the count of detailed rows that satisfy the conditions in the i-th shard, and |Gi| is the number of rows returned after pushdown aggregation. The closer η is to 1, the more pronounced the network and coordination memory savings. Payment summaries are generally divided into low-cardinality groups, such as merchant_id, status and channel, and pre-aggregation is suitable for these. If the query contains non-pushdown window functions or cross-shard global deduplication, the optimizer will only push down filtering and projection, and use staged merging.

4.3 Adaptive Parallel Execution and Slow Fragmentation Management

The executor creates an independent task for each target slice, sorts them according to the historical P95 latency and current queue length, and prioritizes tasks expected to be slower. Parallelism is limited by the connection pool, CPU and disk queue. Let Lj be the predicted load of the j-th work slot at parallelism k. Ideal parallelism aims to reduce the critical path, concurrency overhead and load variance simultaneously.

$$k^*=\arg\min_{1\le k\le K_{max}}\left[\max_{1\le j\le k}L_j+\lambda k+\mu \mathrm{Var}(L_1,\ldots,L_k)\right] \tag{4}$$

As shown in equation (4), Kmax is the maximum parallelism permitted by the connection pool and resource policy; λ represents the scheduling and connection cost induced by the new concurrency; and μ is used to penalise uneven load. If a shard exceeds the historical threshold at runtime, a limited backup read or a degradation to a read-only replica can be triggered; however, the amount and status queries must use the same snapshot timestamp to avoid inconsistent summarisation caused by different visibility among the replicas.

4.4 Historical Feedback Plan Selection and Safe Rollback

The system records the actual latency, number of returned rows, failure rate and resource consumption according to "normalized SQL fingerprint + shard coverage + partition span + parameter bucket". For candidate plan p, the system considers P95 latency, network budget, retry risk and load balancing simultaneously, and selects the plan with the lowest score.

$$p^{*}=\arg\min_{p\in P(q)}\left[w_1\frac{\hat{L}_{95}(p)}{L_{SLA}}+w_2\frac{\hat{V}_{net}(p)}{V_{budget}}+w_3\hat{R}_{retry}(p)+w_4\left(1-\hat{B}(p)\right)\right] \quad (5)$$

P(q) in equation (5) is the set of candidate plans for query q, L̂95 is the predicted P95 latency, LSLA is the service latency target, V̂net is the predicted network volume, Vbudget is the network budget, R̂retry is the predicted retry probability, B̂ is the load balancing degree, and w1-w4 are the service weights. The new plan is first run in a small-scale shadow mode or verified at a limited rate; if the error, failure rate or consistency check exceeds the threshold, it will immediately revert to the known-good stable plan.

4.5 Implementation Path for Payment Services

The three steps of the implementation are as follows: The first stage only supports SQL fingerprinting, interpretable routing logs and partition pruning, and thus has a low risk. The second phase performs pushdown for whitelist aggregation and verifies the number of transactions, amounts and status distribution of the sharding results with respect to the original broadcast results. The third is an adaptive parallelization and historical plan scoring. For refunds, reversals and disputed transactions, idempotent keys, accounting dates and snapshot versions need to be included in the query context. Group cross-currency amounts by currency or convert them at a single exchange rate snapshot first to avoid aggregation semantic errors.

5. Experimental Design and Results Analysis

5.1 Experimental Environment and Data Generation

To acquire credible and accurate statistical data, eight independent SQLite database files were actually established in the local Linux environment for this paper. Each shard contains six monthly tables for January to June 2026, and 30,000 payment records have been written to each table, totaling 1,440,000 records. Shard by modulus 8 for merchant_id, and the record contains user, amount, status, channel and time fields. A combined index is created for merchant_id and created_at. All random data is generated by the same fixed seed 20260710, and the number of transactions and amounts returned by the three strategies have been verified for consistency on a key-by-key basis before execution.

Table 1 Experimental Environment and Dataset Configuration

Layer	Configuration
Host OS	Linux x86_64
CPU	56 logical cores; worker cap = 8
Memory	4GB
Runtime	Python 3.13.5; SQLite 3.46.1
Topology	8 database shards; 6 monthly partitions per shard
Dataset	1,440,000 payment records; seed = 20260710
Shard key	merchant_id modulo 8
Indexes	(merchant_id, created_at); status

As shown in Table 1, the experiment concurrently implements both sharding and table partitioning and does not only simulate routing in memory. Each shard is an independent database file, and monthly partitions also correspond to separate physical tables. Therefore, the overhead of connections, index accesses and result transport generated by broadcasting, partition pruning and intra-shard aggregation can be quantified. A maximum of 8 worker threads will be used; otherwise, a large number of logical cores on the host would be required for parallelism without a clear benefit.

5.2 Comparison Strategy and Measurement Methods

The broadcast strategy sends queries to all six-month tables in the eight shards, retrieves the details, and aggregates them at the coordinator. The Pruning Strategy reduces the number of target shards and monthly tables by merchant_id and time range, but still returns details. The proposed solution performs GROUP BY within each shard on top of the double pruning, and merges in parallel according to the number of active shards. Tests cover shards 1, 2, 4 and 8, select three merchants per active shard, and have query windows of two months. Each group is preheated once, then independently timed 12 times, and the mean, median, standard deviation and P95 are calculated.

Table 2 Test Workload and Measurement Settings

Workload item	Setting
Query window	2026-02-10 00:00:00 to 2026-04-01 00:00:00
Shard coverage	1, 2, 4, and 8 shards
Merchants	3 merchants per active shard
Warm-up	1 run per strategy and coverage
Repeated runs	12 measured runs
Strategies	Broadcast; Pruning; Proposed
Metrics	Mean, median, P95 latency, transferred rows, fan-out
Correctness check	Count and amount aggregates matched across strategies

Table 2 shows the three strategies under the same query conditions and data distribution, and evaluates performance with P95 rather than the fastest single-run value. Coverage is gradually expanded from a single shard to all shards to observe changes in fixed broadcast overhead, shard expansion costs and tail latency. Consistency checks of the results confirm that the performance improvement is due to optimisation of execution location and data volume, not missing records, modified filtering scope, or reduced accuracy in monetary aggregation.

5.3 Query latency results

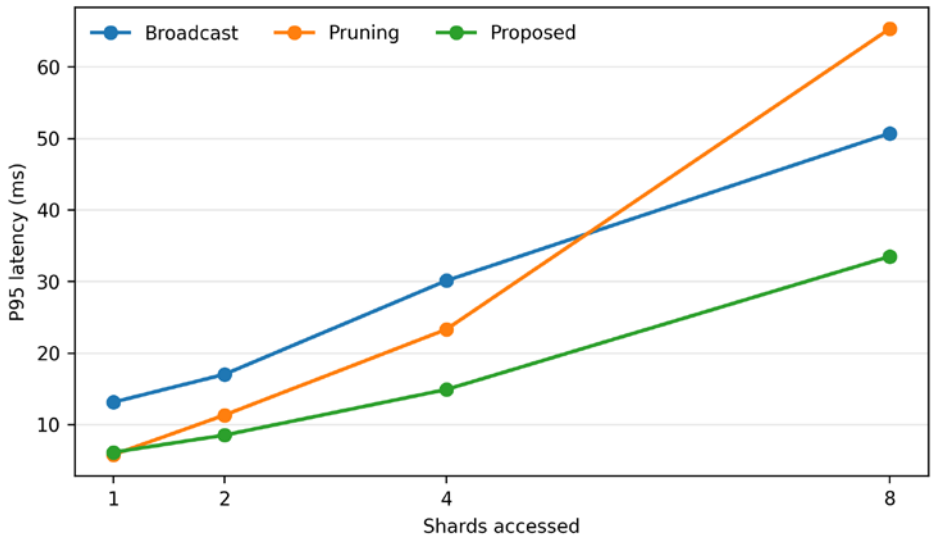

Figure 1. P95 Query Latency Under Different Cross-Shard Coverage (Data Source: Actual Measurements in this Paper)

As shown in Figure 1, the P95 latency of the broadcast strategy increases with an increase in the number of accessed fragments from 13.14ms to 50.73ms; the proposed scheme has a corresponding latency of 6.14ms, 8.54ms, 14.91ms and 33.51ms and is consistently lower. The pruning strategy is only effective for 1 to 4 fragments, and the P95 with full fragment coverage is 65.34 ms. Therefore, when the fragments can no longer be reduced, detailed transmission and resource contention will amplify tail fluctuations, and local pre-aggregation and constrained parallelism need to be applied.

5.4 Data Transmission Reduction and Overall Performance

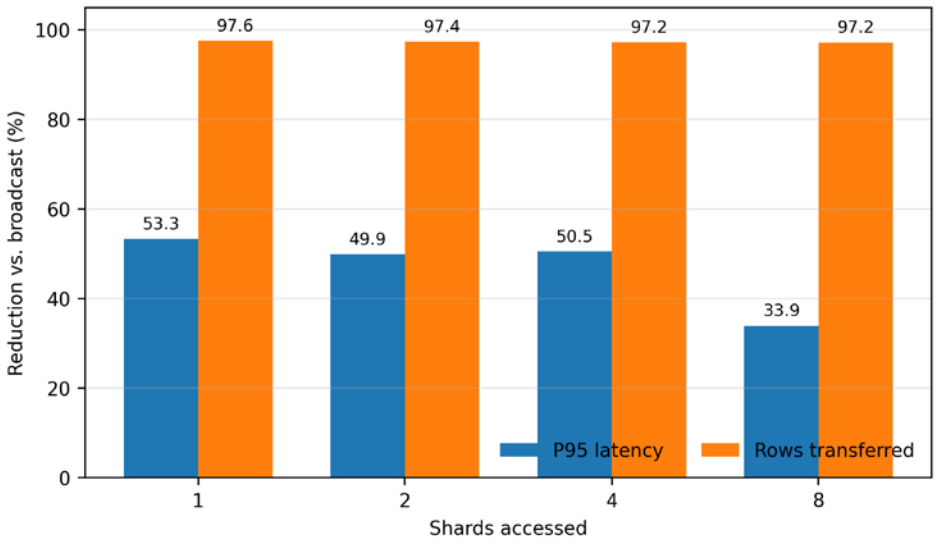

Figure 2 shows that the proposed scheme has reduced both the latency and the number of transmission lines compared to the broadcast method (Data source: Actual measurements in this paper).

As shown in Figure 2, the proposed scheme reduces the number of rows received by the coordinating node by more than 97.2% under all four sharding coverage conditions; thus, local pre-aggregation of low-cardinality groups has a stable benefit. The reduction in P95 latency for 1-4 shards is about 50%, and after full sharding, it is 33.9%. This is because full sharding queries still need to wait for the 8 independent tasks to finish, and the slowest shard and local I/O contention cannot be completely eliminated; however, narrowing the result transmission significantly reduces the memory and merging pressure on the coordinating end.

Table 3. Comparison of Measured Performance for the Three Strategies

Shards	Broadcast P95 (ms)	Pruning P95 (ms)	Proposed P95 (ms)	Proposed rows	Speedup	Row reduction
1	13.14	5.80	6.14	16	2.14x	97.6%
2	17.03	11.33	8.54	33	2.00x	97.4%
4	30.12	23.31	14.91	68	2.02x	97.2%
8	50.73	65.34	33.51	138	1.51x	97.2%

As shown in Table 3, the new plan can achieve a P95 speed-up of 1.51- to 2.14- times over the broadcast method. When accessing 4 shards, the P95 time is 14.91 ms; only 68 rows of partial aggregation results are received by the coordinating node, and when accessing 8 shards, 138 rows are returned, whereas the broadcast method returns 4923 rows. Therefore, the main advantage of the proposed scheme is "fewer tasks, fewer partition reads, and less detail transmission", and parallel execution primarily shortens the critical path across multiple shards; both are necessary.

5.5 Discussion and Boundaries of Application

The experiment confirmed the directionality of the proposed solution, but the SQLite single-machine multi-file environment cannot fully reproduce the network latency, replication protocol, caching level and cross-availability zone failure in the production database. Production deployment should use traffic replay testing on both the actual middleware and database versions to evaluate read committed, repeatable read, and snapshot read semantics separately. For queries that need to perform precise retrieval based on non-shard keys, controlled broadcasting is still required without a global index; for high-cardinality DISTINCT, global sorting, and large-offset pagination, the reduction rate of local aggregations may also be lower. In light of this, covering indices, cursor pagination, hierarchical Top-K, or asynchronous reporting chains should be used instead of a uniform plan.

Optimisers also need to add a correctness indicator in their release condition for payment governance. Set up automatic reconciliation assertions for the total amount, number of transactions, state transitions and currency; monitor data drift in the historical planning model; limit the partition span and concurrency consumption of single queries for popular merchants; and implement cancellation propagation for timeout tasks to prevent sharded tasks from continuing to consume resources after client failure. Only when unifying performance, semantics and observability management can cross-shard optimisation avoid creating new hidden risks.

6. Conclusion

This paper addresses the amplification problem of cross-shard queries in payment system databases after database sharding and table partitioning. It puts forward a cooperative optimisation scheme based on shard routing pruning, time partitioning pruning, local pre-aggregation, adaptive

parallelism and historical feedback plan selection. Based on the cost model, cross-shard performance is not determined by the number of participating shards alone; rather, it is also affected by how much data the coordinating node receives, the slowest shard computation time, and the cost of global merging. Experimental results show that, under the condition of 1.44 million payment records, 8 shards and 6-month partitioning, the scheme reduces P95 latency by 33.9% to 53.3% and lowers the number of rows received by the coordinating node by more than 97%.

Based on the research results, it is suggested that when routing can be determined, remote tasks should be reduced; when aggregation can be decomposed, the results should be compressed at the data's location; after shard coverage is expanded, constrained rather than unbounded parallelism should be used; plan selection needs to rely on interpretable historical feedback and have consistency checks and fast rollback capabilities. Future work may include building a real-world network of read-only replicas and hotspot migration tests for MySQL, PostgreSQL and distributed SQL clusters, and further exploring the robustness of cross-shard global indexes and learning-based cost models under high-traffic payment scenarios.

References

[1] *Du Y, Cai Z, Ding Z. Query Optimization in Distributed Database Based on Improved Artificial Bee Colony Algorithm[J]. Applied Sciences, 2024, 14(2): 846. DOI: 10.3390/app14020846.*

[2] *Shankhdhar P, Liu F, Narale J, Sun J, Schlussel R, Antova L. Presto's History-Based Query Optimizer[J]. Proceedings of the VLDB Endowment, 2024, 17(12): 4077-4089. DOI: 10.14778/3685800.3685828.*

[3] *Justen D, Ritter D, Fraser C, Lamb A, Tran N, Lee A, Bodner T, Haddad MY, Zeuch S, Markl V, Boehm M. POLAR: Adaptive and Non-invasive Join Order Selection via Plans of Least Resistance[J]. Proceedings of the VLDB Endowment, 2024, 17(6): 1350-1363. DOI: 10.14778/3648160.3648175.*

[4] *Wu Z, Marcus R, Liu Z, Negi P, Nathan V, Pfeil P, Saxena G, Rahman M, Narayanaswamy B, Kraska T. Stage: Query Execution Time Prediction in Amazon Redshift[C]//Companion of the 2024 International Conference on Management of Data. New York: ACM, 2024: 280-294. DOI: 10.1145/3626246.3653391.*

[5] *Ding, J. (2025, December). Research on Logistics Cost Control and Optimization in Automotive Manufacturing Supply Chain Based on DMAIC Model. In 2025 IEEE 1st International Conference on Recent Trends in Computing and Smart Mobility (RCSM) (pp. 1-7). IEEE.*

[6] *Sun, J. (2026). Automated Feature Engineering and Screening System for Large-Scale Factor Libraries. Procedia Computer Science, 281, 1282-1290.*

[7] *Hou, Y. (2026). Collaborative Regulation for Stable Data Center Operation under Energy Efficiency Constraints. Procedia Computer Science, 281, 612-620.*

[8] *Zhang, Z. (2026). Research on Performance Optimization Methods for Resource-Aware Model Services in AI Systems. Procedia Computer Science, 281, 1310-1317.*

[9] *Liu, B. (2026). Research On a Framework for Generating and Maintaining Automated Drawline Syntax Trees for Complex Programming Languages. Procedia Computer Science, 279, 420-428.*

[10] *Wu, L. (2026). Construction and Evolutionary Analysis of a Game Model for Supply Chain Finance Funding Based on Blockchain Technology. Procedia Computer Science, 282, 2004-2012.*