# SERON: Smart Query Router for Multi-Primary Cloud-Native Databases with Shared Storage

Yuwei Huang
Department of Computer Science, Tsinghua University
hyw22@mails.tsinghua.edu.cn

Guoliang Li
Department of Computer Science, Tsinghua University
liguoliang@tsinghua.edu.cn

Jinyu Zhang
Gauss Lab
Huawei Company
zhangjinyu.zhang@huawei.com

## ABSTRACT

Cloud-native databases have become a major deployment paradigm for commercial applications and services in recent years. One of the most promising development trends in cloud-native databases is to support multi-primary architecture with shared storage layers. However, when different writer nodes simultaneously access overlapping data ranges, query conflicts naturally arise, potentially leading to a significant performance drop. Therefore, it is of great importance to design query routers that can smartly separate incoming workloads to mitigate inter-node conflicts, which is a problem not fully discussed in the existing literature.

To address this issue, in this paper, we present SERON, a smart query routing system customized for multi-primary cloud-native databases with shared storage. The system aims to minimize conflicts by directing each query to an optimal compute node. To this end, SERON generates workload-aware logical routing rules at different granularities using a two-stage framework, actively detects and handles physical data collisions through judicious conflict modeling, and constantly adapts its routing policies to provide the best runtime performance. The system has been integrated into GaussDB and deployed in real-world scenarios. Extensive experiments verify the effectiveness and cost-efficiency of SERON, which achieves up to 61.7% execution time reduction compared to the best baseline.

**PVLDB Reference Format:**
Yuwei Huang, Guoliang Li, and Jinyu Zhang. SERON: Smart Query Router for Multi-Primary Cloud-Native Databases with Shared Storage. PVLDB, 19(12): 4195 - 4208, 2026.
doi:10.14778/3827998.3828026

## 1 INTRODUCTION

Cloud-native databases are becoming increasingly popular among enterprises thanks to their elasticity, high availability, and cost-efficiency [3, 16, 24]. The prevalence of their application has attracted great research interests from both industry and academia. Well-known cloud-native database providers include Amazon, Microsoft, Google, Oracle, Alibaba, and Huawei [3, 24, 29, 31, 48].

While most cloud-native databases are based on a primary-standby architecture, there have also been studies focusing on multi-primary designs in order to overcome the write throughput bottleneck [3, 10, 12, 18, 24, 39, 48, 49]. Broadly speaking, multi-primary cloud-native databases can be classified into two categories: (i) shared-nothing architecture [10, 18, 39, 49], where data are physically partitioned onto different nodes, and (ii) shared-storage architecture [3, 12, 24, 48], where data are unified at the memory and storage layers. Due to the high overhead of distributed two-phase commit protocols in shared-nothing databases, it is generally acknowledged that shared-storage databases can achieve higher query efficiency with disaggregated compute and storage layers [6, 12].

In spite of promising prospects, shared-storage multi-primary databases still face performance challenges regarding cross-node conflicts [17, 24]. Specifically, when a primary needs to update a record, it must acquire ownership of the page containing the record beforehand, which requires costly global communication and synchronization. Therefore, if multiple writer nodes simultaneously access records within the same page, the overhead for page ownership transfer would accumulate rapidly, resulting in severe query performance degradation and even deadlocks. The aforementioned phenomenon necessitates the design of a mechanism that can alleviate such data conflicts [24, 48].

Several techniques have been proposed to address this problem. GaussDB [24] builds page-affinity graphs to capture internal data dependencies and then applies graph partitioning algorithms to logically assign pages onto each node. A dedicated ML model is used to establish the correspondence between SQL queries and pages. This method suffers from the intrinsic inference latency of the ML model. It also overlooks the difference between read and write operations as well as load balancing. Chimera [17] introduces sophisticated mechanisms for transaction scheduling and page ownership transfer in order to reduce query conflict penalty at runtime. This method primarily focuses on database execution engines and assumes that the workload is well-partitioned and balanced, which is not always the case. The complicated system design also makes it harder to be integrated into existing cloud-native databases as compared to a pluggable query router. Other query routers [19, 33, 35] are primarily designed for shared-nothing databases and read-only workloads, thus overlooking the cross-node conflicts arising in shared-storage systems. Moreover, the routing latency of these methods (usually sub-second) can be prohibitive for short-running OLTP queries. *In summary, existing approaches fail to simultaneously achieve routing rule generation, query conflict handling, and load balancing in hybrid read-write workloads.*

**Challenges.** There are three main challenges in designing smart query routers for multi-primary cloud-native databases with shared storage. First, it is challenging to generate workload-aware routing rules that can simultaneously mitigate cross-node logical data contention and prevent compute-node hotspots **(C1)**. Second, while


Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828026

routing rules can achieve logical data separation, physical conflicts may still emerge from concurrent queries even when they target different data ranges, and it is non-trivial to detect and handle such conflicts in real time **(C2)**. Third, as query workload patterns can shift over time, there is a need for continuous runtime adaptation of routing policies to maintain optimal load balancing. This is particularly challenging for complex transactions targeting highly-contended ranges that involve both read and write operations, as they exhibit different parallel behaviors **(C3)**.

**Our Approach.** To tackle the above challenges, we propose *Seron*, a **S**mart qu**E**ry **RO**uter for multi-primary cloud-**N**ative databases with shared storage. Specifically, to address **C1**, we propose a two-stage framework for logical rule generation at different granularities based on historical workloads, which is more flexible in terms of rule expressivity compared to traditional partitioning schemes used in distributed databases (Section 4). To address **C2**, we model index distributions and correlations with carefully selected statistics, which help to detect and handle potential physical data collisions (Section 5). To address **C3**, we constantly monitor query workload drifts at runtime and adjust the routing policies, which includes using dedicated regression models to capture the parallel behaviors of complex transactions (Section 6). All these components contribute to faster query execution and better load balancing as verified by extensive experiments in Section 7.

**Contributions.** We make the following contributions:

- We identify the performance challenges in multi-primary cloud-native databases with shared storage and propose Seron, a smart query router designed to address these challenges.
- We introduce a flexible two-stage logical rule generation framework that can partition user workloads at different granularities based on historical workloads;
- We design conflict handling techniques that can actively detect and avoid potential physical data collisions in real time;
- We propose online adaptation mechanisms that can adjust the routing policies based on runtime workload drift characteristics in order to promote load balancing;
- We integrate Seron into GaussDB and conduct extensive experiments on four different datasets where Seron achieves up to 61.7% execution time reduction with negligible extra overhead.

## 2 PRELIMINARIES

In this section, we first briefly review some core techniques used in multi-primary shared-storage cloud-native databases that are related to our work, including compute-storage disaggregation (§ 2.1) and data consistency guarantee (§ 2.2). We then present and discuss the problem of query routing under this setting (§ 2.3).

### 2.1 Compute-Storage Disaggregation

One of the many advantages offered by cloud-native databases is their high elasticity, which enables the cloud system to allocate or release resources based on current load conditions, thereby achieving optimal cost-efficiency. To this end, many cloud service providers have adopted the **compute-storage disaggregation** architecture where compute and storage nodes scale independently according to different user demands [32, 44, 50]. Several designs have taken one step further by introducing a separate memory layer

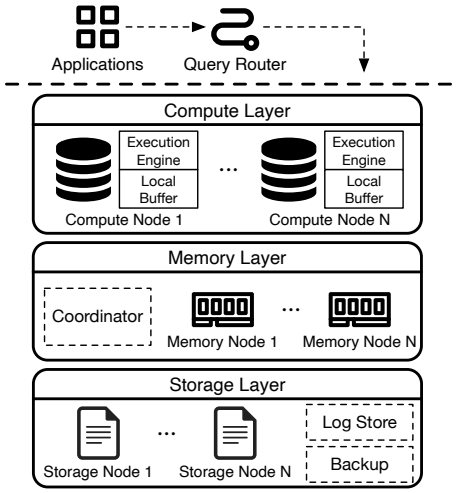


**Figure 1: A typical cloud-native database architecture with compute-memory-storage disaggregation.**

that is shared by all compute nodes, also known as the **compute-memory-storage disaggregation** architecture [23, 24, 45, 48].

Figure 1 illustrates a typical cloud-native database architecture with disaggregated compute, memory and storage. The compute layer consists of multiple independent compute nodes that are responsible for processing user queries. Each compute node usually have its own buffer pool to accelerate local query execution. The memory layer, on the other hand, serves as a shared buffer pool for all the compute nodes. It typically contains several memory nodes and a coordinator, which handles cross-node communication via high-speed network. The storage layer is where all the data are persisted, including both page store and log store. Replication and backup techniques are commonly adopted at this layer to strengthen the reliability of the database.

Despite improved elasticity, separating storage from compute inevitably brings concurrency issues. Specifically, a compute node needs to be informed of the latest changes in a page before reading it, and needs to obtain exclusive page ownership before writing it. Without carefully designed concurrency control protocols, there can be risks of violated data consistency or degraded query performance. We will elaborate more on data consistency guarantee techniques in the next section.

### 2.2 Data Consistency Guarantee

Generally speaking, there are two categories of concurrency control protocols in database systems, namely Optimistic Concurrency Control (OCC) and Pessimistic Concurrency Control (PCC). Both types of protocols have seen application in existing multi-primary cloud-native databases [3, 24].

In an OCC protocol, transactions are executed in a lock-free manner, and data consistency is ensured through commit-time conflict check. Amazon Aurora [3] has adopted this type of protocol. OCC protocols are most suitable for workloads with few cross-node conflicts, because otherwise it may lead to frequent transaction aborts which severely impact system performance.

In a PCC protocol, data consistency is guaranteed through explicit lock management. This protocol is more commonly employed

thanks to its robust performance under conflict-intensive scenarios [12, 24, 48]. A typical PCC implementation includes a hierarchical page-row locking mechanism. Page locks represent current page ownership and are necessary for a compute node to perform write operations, thus ensuring physical data consistency. Row locks, along with distributed clocking techniques such as the Lamport Clock [22], indicate recent update information of a record and help to ensure logical data consistency among transactions. Global lock information is usually held and managed by a centralized metadata server inside the coordinator of the memory layer, which can become the performance bottleneck in case of critical lock contention.

## 2.3 Problem Statement

As we can see, both OCC and PCC protocols suffer from cross-node data conflicts, either due to transaction aborts or because of lock contention. Therefore, it is of great importance to avoid such conflicts during query execution. While designing sophisticated protocols and modifying database internals are seemingly straightforward directions, in this work, we propose to address the problem at the query router module as it is much easier to deploy and less dependent on specific database implementation.

Achieving smart query routing is essentially an optimization problem whose objective is to minimize query response time (or to maximize query throughput). From the perspective of a query router, there are two major factors that can be controlled to approach this goal, i.e., **where** and **when** to send the queries for execution. Formally, given a query universe $Q$, a query router can be expressed as a function $r$ that maps each $q \in Q$ along with its arrival time $t_a \in \mathbb{R}_{\geq 0}$ to a compute node indexed by $i \in \mathbb{N}$ (where) and execution start time $t_s \in \mathbb{R}_{\geq 0}$ (when). If we denote the response time of query $q$ as $\text{Resp}(q; i, t_s)$ where $i$ and $t_s$ represent the routing decisions as described above, then for a workload $W$ consisting of $(q, t_a)$ pairs, the problem of query routing can be formulated as:

$$\min_{r:Q\times\mathbb{R}_{\geq 0}\to\mathbb{N}\times\mathbb{R}_{\geq 0}} \sum_{(q,t_a)\in W} \text{Resp}\big(q; r(q, t_a)\big). \tag{1}$$

To answer the **where** question, Seron establishes query-node mappings (i.e., routing rules) that simultaneously mitigate logical data contention and avoid load imbalance. To answer the **when** question, Seron further detects physical data collisions and prevents conflicting queries from concurrent execution. Both these optimization techniques facilitate the objective of Eq. (1) by either accelerating query execution or reducing long-tail latency. We will provide more details in the following sections.

# 3 OVERVIEW

In this section, we present an overview of our smart query router Seron, illustrating both its architecture (§ 3.1) and workflow (§ 3.2).

## 3.1 Seron Architecture

Figure 2 shows the overall architecture of our Seron system. The system operates as a query router and can be easily integrated into existing database drivers inside JDBC/ODBC. Essentially, Seron serves as an intermediate layer between clients and servers, which accepts user queries as input and directs them to appropriate compute nodes. Currently, Seron supports single-table and multi-table join queries with equality and range predicates. Seron also supports

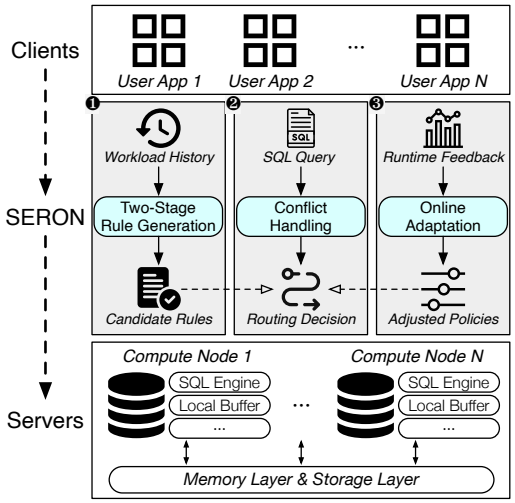


**Figure 2: Seron architecture.**

stored procedures in which each input parameter maps directly to a base-table attribute referenced by the procedure's SQL and/or to a predicate value used in the procedure's internal WHERE or JOIN conditions. Stored procedures that use parameters indirectly – such as by combining them in expressions, constructing dynamic SQL strings, or altering control flow in ways that change the executed SQL – are left for future work. Note that Seron's deployment is independent of backend cloud servers, which emphasizes its flexibility towards various database implementations.

Inside the architecture of Seron there are three major components, namely two-stage rule generation, conflict handling, and online adaptation, each corresponding to one of the challenges summarized in Section 1. We will provide a brief description to these components in the following paragraphs.

**Two-Stage Rule Generation.** At the core of a query router is a set of routing rules, and the rule generation component is responsible for producing candidate rules at different granularities. Specifically, Seron collects workload history during runtime, including both SQL statements and their execution latencies. Then, based on these historical data, the component performs logical partitioning on the workload such that each partition corresponds to one of the compute nodes and the overall load falling on each compute node is balanced. The partitioning process is first conducted at the table level using a dedicated access graph, and if that fails to produce balanced partitions, we turn to row-level range partition to obtain the final result. By creating logical partitions, the component establishes a mapping between the data a query may access to the compute node that can execute the query, which functions essentially as a set of query routing rules.

**Conflict Handling.** The rule generation component provides logical guidelines to separate the workload, but there are still physical conflicts that remain to be addressed. While databases are able to handle such conflicts through locking, the execution engine often lacks a global view of all active queries, thereby incurring frequent page ownership transfer due to intense lock contention. To mitigate this issue, the conflict handling module detects both conflict patterns (see Section 5) and introduces corresponding mechanisms to resolve them at the router level. By preventing conflicting queries

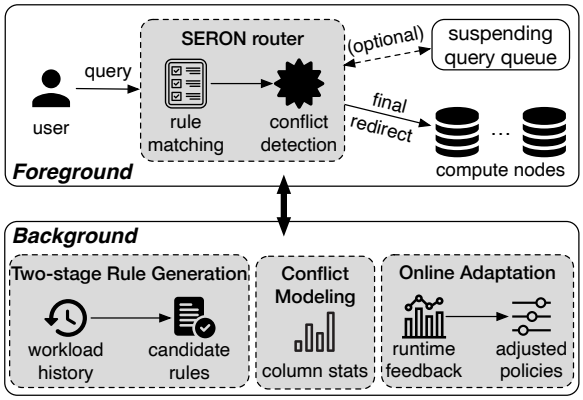


**Figure 3: Seron workflow.**

from concurrent execution, the component effectively reduces page ownership transfer and addresses the **when** question.

**Online Adaptation.** In real scenarios, it is common that query workloads may shift over time, making the routing rules calculated from historical workloads no longer applicable, thereby violating load balancing requirements. To address this issue, Seron uses an online adaptation component to constantly detect workload drifts and adjust the routing policies. Specifically, when a compute node is experiencing significantly heavier load, we shrink its data range to redirect some queries to other nodes. If all current queries are concentrated on a data range that is too small to split, we turn to dedicated regression models to predict their parallel behaviors before optionally offloading them to multiple compute nodes.

To summarize, the two-stage rule generation component answers **where** a query should be sent, while the conflict handling component controls **when** a query can start executing. The online adaptation component, on the other hand, ensures Seron's load balancing capability amidst workload drifts. All these components contribute to Seron's increased routing performance.

### 3.2 Seron Workflow

Figure 3 shows the workflow of Seron. As we can see, in the foreground, Seron takes user queries as input, matches them with pre-computed candidate rules, and detects and handles physical contention before temporarily suspending conflicting queries or finally redirecting them to backend compute nodes. Note that we only retain necessary routing steps in the foreground in order to minimize the additional overhead added to the end-to-end query latency. In the background, Seron collects workload history, column statistics, and runtime feedbacks that are used by the corresponding components in the foreground. Specifically, the two-stage rule generation process produces candidate routing rules, the conflict modeling process collects database statistics used for conflict detection, and the online adaptation process adjusts existing rules to quickly respond to workload drifts. All their calculated results are periodically synchronized to the foreground to adapt the router to the latest workload and data patterns.

## 4 TWO-STAGE RULE GENERATION

The two-stage rule generation component is responsible for producing logical routing rules from workload history. In this section, we first clarify the concept of routing rules within the context of this paper (§ 4.1), then describe in detail how Seron performs the generation process, including both table-level rule generation (§ 4.2) and row-level rule generation (§ 4.3), and how they are combined into a unified partitioning framework (§ 4.4).

### 4.1 Routing Rule Definition

From the perspective of a query router, a routing rule is essentially a mapping between queries and backend compute nodes. Since Seron primarily focuses on reducing data conflicts, we characterize each input query by its accessed data, as specified by the tables and predicates within the query (for stored procedures, we derive this information from input parameters that directly correspond to tables/predicates of their internal SQL statements). In this light, routing rules can also be viewed as data-to-node mappings generated through a process analogous to database partitioning. Formally, let $D$ represent the database and $n$ be the number of compute nodes. Seron first establishes a mapping $p : D \to \{1, \ldots, n\}$ by partitioning $D$ into disjoint subsets $\{P_1, \ldots, P_n\}$ based on workload history, and then transforms $p$ into operational routing rules through the natural correspondence between a query and its accessed data. The detailed partitioning process follows a hierarchical structure, starting with coarse-grained table-level rule generation and subsequently refined with fine-grained row-level adjustments, which we describe in the following sections.

### 4.2 Table-Level Rule Generation

In order to partition the workload at the granularity of tables, we first need to capture the table-level dependencies within the workload history. To this end, Seron builds a Table Access Graph (TAG) based on all historical queries (including both single-table and multi-table join queries). Specifically, each vertex in the TAG represents a table in the database, and the edges between vertices hold the co-access count of tables. Seron initializes the TAG by assigning each vertex with weight 0 and frequency 0. Then for each recorded query, Seron extracts its tables and updates the frequency of corresponding vertices in the TAG. Seron also distributes the query's execution latency evenly onto the weights of these vertices. If the query involves multiple tables, Seron increases the co-access count of the edges between all table pairs. Intuitively, tables with higher vertex frequencies tend to appear more often, and edges with larger co-access counts indicate stronger table correlation.

After constructing the graph, Seron is now ready for table-level partitioning. Specifically, Seron attempts to split the vertices into subsets corresponding to the number of compute nodes, with the aim of balancing partition weights and minimizing dependencies between different partitions. This process is known as the Graph Partition Problem, which is provably NP-hard. Therefore, in order to generate table partitions efficiently, we design a heuristic-based greedy algorithm, which consists of three steps. First, we define the dependency score of table $s$ on table $t$ as

$$\mathrm{Dep}(s,t) := \frac{\mathrm{CoAccess}(s,t)}{\mathrm{Frequency}(s)}, \tag{2}$$

which takes values between $[0, 1]$. A table pair $(s, t)$ is considered as *dependent* if $\mathrm{Dep}(s,t)$ or $\mathrm{Dep}(t,s)$ exceeds a certain threshold (e.g., 25%). We then remove the edges between non-dependent tables and identify the connected components (CCs) in the TAG as the

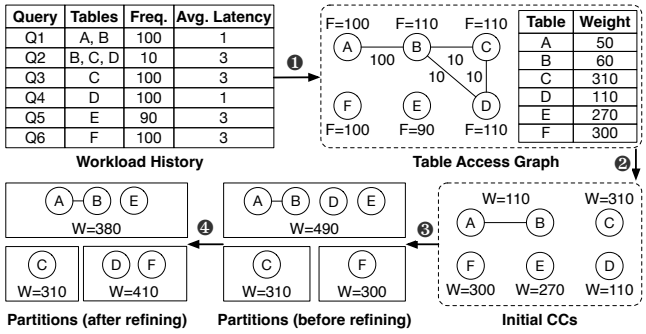


**Figure 4: Table-level rule generation.**

initial partitions. Second, we greedily find and combine the two least-weighted partitions until the number of remaining partitions matches that of the compute nodes. Third, we refine the partitioning result by iteratively examining whether there is an initial CC that can be migrated from the largest partition to the smallest partition to narrow their weight difference.

Figure 4 shows an example of the table-level rule generation process for three compute nodes. Note that the edges between B, C, and D are removed before constructing the initial CCs due to their relatively low dependency scores. During the partitioning phase, Seron greedily combines (A, B) with D, and then with E, to form an intermediate partitioning result, which is later refined into the final partitions. These partitions can serve as routing rules by directing each query to the compute node whose corresponding partition overlaps the most with the query's tables.

## 4.3 Row-Level Rule Generation

Besides table-level rules, Seron also supports finer-grained row-level rule generation, which splits the workload based on predicate values. Specifically, Seron maintains a weighted histogram for each index column, with the horizontal axis spanning the index value range and the area of each bin representing the accumulated query latency within that interval. Note that for discrete index columns, we convert its values to continuous intervals to maintain representation consistency. The number of bins is a configurable hyperparameter and defaulted to 64 in Seron, which produces sufficiently good partition results. While a larger setting enables finer-grained rules, the performance gains are marginal and at the cost of increased generation overhead. Given a workload history, Seron extracts for each query its predicates and adds the query's execution latency to the corresponding histogram bins. For instance, query Q1 in Figure 5 intersects with both $[1, 3)$ and $[3, 5)$. Therefore, Seron distributes its latency to both bins proportionally to their overlapping lengths. Similarly, query Q2 covers three intervals, and its latency is divided equally among the corresponding bins. As for queries Q3 and Q4, which fall entirely within a single interval, Seron directly assigns their latency to the respective bins.

After histogram construction, the partitioning process is straightforward: Seron splits the index value range into equi-area intervals, where the number of intervals matches the number of compute nodes. For instance, in Figure 5, Seron partitions the histogram into three intervals: $[1, 3.5)$, $[3.5, 5)$, and $[5, 9)$. By assigning each interval to a specific node and applying the whole process to every index column, Seron generates a collective row-level rule set. Note

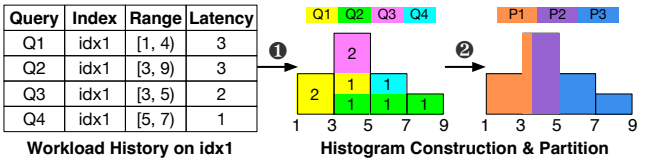


**Figure 5: Row-level rule generation.**

**Algorithm 1:** Unified Partitioning Framework

**Parameters:** workload history $W$; number of partitions $n$; imbalance threshold $r$;

```
1  function UnifiedPartition(W, n, r)
2      OutlierCCs ← [];
3      P ← TablePartition(W, n);
4      while P.maxWeight() > r * P.minWeight() do
5          M ← P.maxPartition();
6          # Select outlier CC from M whose weight is
7          # closest to P.maxWeight() − P.minWeight()
8          CC ← SelectOutlier(M, P);
9          OutlierCCs.append(M.pop(CC));
10     W ← FilterWorkload(W, OutlierCCs);
11     Q ← RowPartition(W, n);
12     return (P, Q); # The result is a hybrid rule set
```

that, unlike table-level rules, a single query's predicate set may match multiple row-level rules pointing to different destinations. We will elaborate on how Seron prioritizes these rules based on index statistics in Section 5.

## 4.4 Unified Partitioning Framework

Both table-level and row-level rule generation serve the purpose of load balancing and conflict reduction. Table-level rules offer strong isolation guarantees, i.e., interferences between different compute nodes are minimized as they operate on distinct tables. However, achieving perfect table-level partitioning can be difficult as real-world tables are usually highly related within the schema. Row-level rules, on the other hand, are generally applicable as long as appropriate index columns exist, but they cannot fully eliminate cross-node conflicts because different compute nodes may still access the same table (further discussion in Section 5). Therefore, we propose a unified partitioning framework that combines the benefits of both approaches. The framework starts by partitioning the workload history at the table level. If this initial partitioning results in skewed partitions (e.g., the weight ratio of any two partitions exceeds a threshold $r$), the largest partition is repeatedly examined to identify outlier CCs that can be removed to make the partition weights more balanced. These extracted CCs are then used to filter the original workload, forming a workload subset involving only relative tables, which is subsequently partitioned based on index values. In this way, Seron maximizes the utilization of strong-isolation guarantees provided by table-level rules while also ensuring overall balance by handling imbalanced parts through row-level rule generation. The aforementioned unified partitioning framework is summarized in Algorithm 1.

The proposed partitioning framework differs from existing distributed database solutions [11, 34, 37, 52] in four aspects. First, the

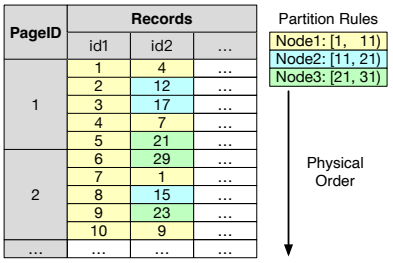


**Figure 6: Example of intra-index conflicts.**

partitioning process is performed at multiple granularities, providing greater flexibility and stronger isolation guarantees compared to single-level partitioning schemes. Second, the framework is implemented logically rather than physically, which is lightweight and can be directly integrated into the query router, maintaining decoupling from backend databases. Third, leveraging the shared-storage architecture of multi-primary cloud-native databases, the overhead of partition migration is negligible (only in terms of transient cache misses), which allows Seron to be more aggressive in adapting to workload drifts. Finally, Seron accommodates multiple partition rules simultaneously in the form of a candidate rule set, in contrast to traditional solutions that support only a single active rule. Seron also allows users to specify a customized rule subset that remains fixed throughout the partitioning process.

## 5 CONFLICT HANDLING

The conflict handling component is another critical part of our Seron system, which actively detects and prevents physical data collisions in the foreground and addresses the **when** question, complementing the background logical rule generation framework. In this section, we describe how Seron models and handles two predominant types of conflicts observed in multi-primary shared-storage databases, which arise from either physical-logical misalignment within a single index (*intra-index conflicts*, § 5.1) or logical-logical misalignment across different indexes (*inter-index conflicts*, § 5.2).

### 5.1 Intra-Index Conflict

As mentioned earlier, row-level rules can still suffer from cross-node conflicts despite their ability to logically partition the records within a table. This phenomenon is often due to the misalignment between logical and physical data layouts. Consider the example in Figure 6 where column *id1* is the clustered index that perfectly aligns with the physical order, whereas column *id2* is a non-clustered index that is poorly correlated with the physical layout. Suppose that both indexes' value ranges are split into the same set of intervals $\{[1, 11), [11, 21), [21, 31)\}$, assigned sequentially to three backend compute nodes. Then, as we can see in the figure, while the rule on *id1* is able to guarantee physical isolation between different nodes (e.g., all records in Page 1 and Page 2 correspond to Node 1, with no overlaps or intersections with other pages or nodes), conflicts may arise with respect to *id2* as different nodes can still access the same page (e.g., records in Page 1 are assigned to different nodes according to their *id2* attribute values and the rule intervals). Such conflicts, which we refer to as *intra-index conflicts*, need to be modeled and resolved properly for better query performance.

**Conflict Modeling.** The root cause of intra-index conflicts lies in the fluctuation of index values, which results in the mapping of physically adjacent records to different row-level rule intervals and, eventually, across distinct compute nodes. To measure the degree of fluctuation for each column, an intuitive metric is the *total variation*, which is defined as the sum of absolute differences in column values between all adjacent record pairs. Formally, suppose the records within a table follow the physical order $(r_1, r_2, \ldots, r_n)$, then for each column $c$, its total variation is calculated as

$$\mathrm{TV}(c) := \sum_{k=1}^{n-1} \left| r_{k+1}[c] - r_k[c] \right|, \tag{3}$$

where $r_k[c]$ denotes the value of record $r_k$ in column $c$. A larger TV indicates a more scattered column value distribution, which increases the likelihood of intra-index conflicts.

While TV provides a qualitative measure of fluctuation, further refinement is required to make it comparable across different indexes, as the metric also depends on the column value range and the number of records. To bridge this gap, Seron normalizes the TV of each column $c$ against its value range before dividing by the record count to derive the mean difference between adjacent record pairs (obtainable in most database systems via a Sequential Scan), denoted as $d$. Seron also calculates the average number of records $m$ in each page based on backend database statistics. Given a row-level rule on $c$, Seron simulates the column value fluctuation within a page as an $m$-step random walk of step size $d$, with its starting point sampled from the value distribution of $c$. Seron then counts the number of all simulated record pairs in the page that are in conflict (i.e., belonging to different rule intervals). By repeating this Monte Carlo process multiple times, Seron derives an estimation of the average conflict rate within each page, which is a directly comparable metric across different columns.

The calculation of TV requires a full sequential scan through the table, which, despite being affordable in most cases, can be time-consuming for exceptionally large tables. Fortunately, as we have observed in practice, the estimated conflict rate remains stable across data updates. Therefore, for high-cardinality tables, Seron estimates the mean difference $d$ using only a fraction of data samples, bypassing the calculation of the actual TV for higher efficiency. The Monte Carlo process, on the other hand, typically converges within $10^4$ simulations, whose computational cost is negligible. In our experiments, the whole conflict modeling process completes within seconds in the background, imposing little overhead on the foreground query router.

**Conflict Resolution.** Following the conflict modeling process, Seron ranks all row-level rules based on their estimated conflict rates to prioritize those that incur minimal intra-index conflicts. When an incoming query's predicates match multiple row-level rules, Seron selects the highest-ranking one to determine its routing destination. In cases where a query can only match with high-conflict rules, we employ an additional conflict table to actively detect and avoid concurrent accesses across different compute nodes, as explained in the next section.

### 5.2 Inter-Index Conflict

Besides intra-index conflicts, another source of data contention arises from the weak correlation between distinct indexes. Consider the example in Figure 7 where indexes *id1* and *id2* represent different logical data dimensions within the table. When queries

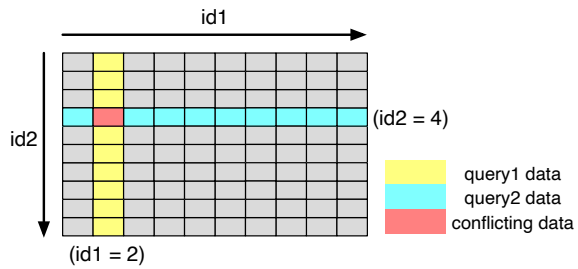


**Figure 7: Example of inter-index conflicts.**

on both indexes are executed simultaneously, contention naturally occurs within the overlapping region, as highlighted in red in the figure. In contrast to intra-index conflicts, this type of conflicts, which we refer to as *inter-index conflicts*, is purely on the logical level and stems from inherent index correlations rather than alignment with physical data layout. Ideally, queries on such conflicting indexes should avoid concurrent write operations to mitigate frequent page ownership transfer within the contested region, which is a major cause of performance drop in multi-primary cloud-native databases as pointed out earlier.

**Conflict Modeling.** To identify potential inter-index conflicts, SERON clusters all index columns in the database into correlated groups. Specifically, SERON first constructs for each table a pairwise Pearson correlation matrix of its index columns. Then, SERON regards each index column as an independent initial group and iteratively combines these groups by identifying index pairs with high correlation coefficients (e.g., exceeding 0.8). The resulting table-level groups are further merged via primary-foreign key relationships specified in the schema to form the database-level groups. Queries whose index predicates belong to the same group are considered suitable for concurrent execution, as their correlated nature implies a minimized contention footprint. Rules based on indexes in distinct groups are therefore considered as conflicting.

Analogous to the calculation of intra-index conflict rates, the clustering results also appear to be robust to data updates. Therefore, for large tables with multiple indexes, SERON estimates the correlation matrix using a similar sampling-based approach to reduce overhead. The entire clustering process is carried out periodically in the background during off-peak intervals to minimize interference with foreground query routing.

**Conflict Resolution.** To detect potential data collisions caused by inter-index conflicts, SERON maintains a global conflict table (stored in the backend databases) to monitor the data access patterns of active queries. The conflict table records each query's type (read or write), involved tables, and matched routing rule before its execution, and invalidates the corresponding entry upon its completion. When a new query arrives, SERON performs a pre-routing validation against the conflict table to check the following conflict conditions:

- **Table-Level Conflicts:** The incoming query's involved tables overlap with those of an active query routed via table-level rules;
- **Inter-Index Conflicts:** The incoming query's routing rule conflicts with that of an active query routed via row-level rules;
- **Intra-Index Conflicts:** The incoming query's routing rule exhibits a high conflict rate (as introduced in the previous section) and there exists an active query routed via the same rule but executing on a different target node.

If any conflict condition is met and at least one of the queries involves a write operation, SERON identifies a potential data collision and postpones the incoming query until there are no conflicts or a timeout is reached. We allow users to specify a maximum timeout (i.e., the longest allowed query delay) to balance between query performance and result timeliness. Essentially, SERON utilizes the conflict table to reorder queries under timeout constraints, aiming to minimize the concurrent execution of conflicting queries.

For index predicates within the same correlated group, an extra rule alignment step is necessary to facilitate conflict-less execution. For instance, a primary-foreign key pair should follow the same routing rules to maintain logical consistency, whereas the rule generation component might produce different sets of intervals for them due to divergent workload histories. To address this issue, SERON designates a representative index for each group and maps the rule intervals of other indexes onto this pivot. The mapping is derived via direct linear regression performed alongside the computation of index correlation coefficients, leveraging the fact that group members are already highly correlated by construction. When an incoming query passes the validation, SERON routes it using the transformed predicates and row-level rules to ensure maximum logical separation across compute nodes.

Since the conflict detection component is situated on the critical path of query routing, a major concern is the additional latency it adds to query response. As described earlier, SERON minimizes this overhead by offloading the costly statistics-gathering processes to the background, leaving only conflict table lookups and updates in the foreground. These operations typically complete within milliseconds and are negligible compared to the overall query execution time. For extremely short-running queries with millisecond-level latencies, SERON bypasses their conflict detection process to ensure minimal performance degradation.

It is worth mentioning that the mechanisms introduced in this section can also be applied to non-index columns. However, due to the high overhead of collecting statistics for all column pairs and the predominance of index predicates in real-world workloads, SERON mainly focuses on index-related statistics and conflicts. In rare cases where no indexes exist, it is easy to extend SERON to support conflict handling of non-index columns.

## 6 ONLINE ADAPTATION

During system operation, it is common for query workloads to shift over time, rendering the routing rules calculated from historical workloads imbalanced. To efficiently re-balance the workload without the high overhead of frequent re-partitioning, we implement online-adaptation mechanisms inside SERON. In this section, we first introduce how we detect runtime workload drifts (§ 6.1), and then present adaptation techniques corresponding to both mild workload drifts (§ 6.2) and highly skewed query ranges (§ 6.3).

### 6.1 Workload Drift Detection

In real-world scenarios, the ability to adapt quickly to workload drifts is vital for query routers to achieve robust performance. While the background rule generation framework can continuously update the routing rules based on recent workloads, it often suffers from relatively high computational overhead, which makes it difficult to respond in real time. To enable more timely adaptation,

SERON incorporates mechanisms to detect potential workload drifts based on runtime feedbacks. Specifically, SERON periodically collects recent throughput and the number of backlogged queries of each compute node, based on which it estimates the expected latency of executing all queued queries. These latencies serve as qualitative measurements of the current load distribution, and in order to promote load balancing, it is necessary to offload queries from hotspot compute nodes. According to different levels of workload skewness, SERON re-balances the workload through either routing rule adjustment or connection configuration, which we elaborate in the following sections.

## 6.2 Routing Rule Adjustment

Upon workload drifts, SERON first tries to adjust the routing policies by shrinking the partition ranges corresponding to overloaded compute nodes within active rules (i.e., rules that have matched at least one query since the previous adjustment). Specifically, for a table-level rule, SERON greedily selects and migrates the largest CC to the least loaded compute node. For a row-level rule, SERON adjusts the interval boundaries by a predefined step to shift the burden from hotspot compute nodes to underutilized destinations, analogous to ideas used in consistent hashing [21] which enjoys the advantage of minimal data movement. Finally, SERON checks the waiting queue and redirects those backlogged queries whose destinations have been modified according to the adjusted rules.

In contrast to an entire re-partition, the routing rule adjustment process primarily aims to rapidly mitigate the burden on saturated compute nodes rather than achieve perfect load balance all at once. Such a design allows SERON to continuously focus on the reduction of long-tail query latency, which is a critical factor that impacts end-user experiences. Moreover, when compute nodes are dynamically added or removed, the same process can be employed to quickly re-distribute the workloads among the updated node set. In our implementation, we further enhance the responsiveness of rule adjustments by employing lazy-update techniques, that is, a rule is only modified when matched with an incoming query.

## 6.3 Connection Reconfiguration

While rule adjustments are able to handle common workload drifts, there are still scenarios where queries are too concentrated within a specific range to be redirected via rule changes alone. For instance, a TPC-C workload may temporarily consist of queries exclusively targeting a single warehouse, which, as the warehouse ID itself usually serves as the routing index, cannot be distributed across different compute nodes through interval boundary adjustments. In such cases, further assessment is required before determining whether the workload can be offloaded to multiple compute nodes.

Intuitively, read-intensive queries benefit from multi-node execution as they do not require exclusive page ownership, thereby achieving higher resource utilization across compute nodes. In contrast, write-intensive queries may experience performance degradation due to increased data conflicts, which often outweighs the improvements gained through load balancing. This intuition is empirically validated by our mini-experiment targeting a single warehouse in TPC-C as illustrated in Figure 8. From the figures, we can see that for read-intensive transactions such as *OrderStatus*, the overall TPS (Transactions Per Second) consistently scales

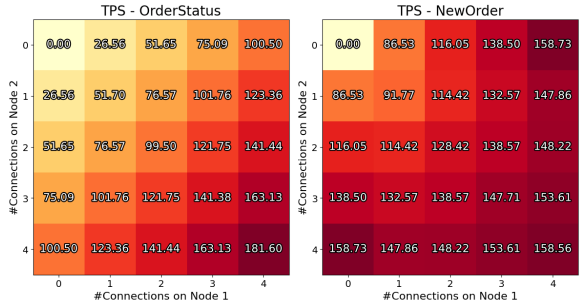


**Figure 8: Mini-experiment on multi-node execution TPS of TPC-C transactions targeting a single warehouse.**

with the total number of connections (i.e., parallel query execution slots), regardless of whether these connections reside on a single node or multiple nodes. For write-intensive transactions such as *NewOrder*, however, offloading the workload to multiple nodes brings no improvement to, and in fact often harms, the overall TPS. These findings agree with our previous analysis.

Based on the observations, we conclude that the offloading performance of queries within the same range is impacted by two major factors, namely the access type (read or write) and connection-level concurrency. Especially for complex transactions that exhibit varying read-write characteristics, SERON needs to determine and apply the optimal connection configuration to maximize their throughput. To this end, SERON integrates a set of lightweight regression models to predict the parallel behaviors of different transactions to assist in the configuration process, which attempts to offload the dominant transaction template if a rule adjustment fails.

**Model Structure.** We adopt XGBoost [7] for regression, which is widely recognized for its efficiency and accuracy on tabular data. To capture the distinct parallel behaviors of different transactions, we construct a dedicated model for each transaction template. The input to these models is a vector composed of the number of concurrent connections on each compute node, and the output is the normalized average latency (measured against single-node serialized execution) for that transaction under the specified connection configuration. To ensure generalizability to unseen transaction templates when encountering new workloads, we establish a global model that incorporates a transaction's read-write ratio as an additional feature, which is a critical influencing factor as discussed earlier. We derive this ratio directly by counting the number of read and write queries within the transaction logic. Leveraging these models, SERON can estimate the expected latency of complex transactions across varying concurrency levels, providing useful guidance for finding the optimal connection configurations.

**Data Collection.** SERON collects training data through a hybrid approach involving both online monitoring and pseudo-executions. When the system is busy, SERON records the execution latency of active transactions and captures their corresponding connection-level concurrency from the conflict table. During idle periods, SERON proactively explores unseen connection configurations and pseudo-executes the transactions (i.e., aborting instead of committing at the end of the transaction) to obtain their average latency without introducing side effects. Additionally, if the workload is known a priori, SERON can also collect training data offline to completely eliminate interference with online user requests.

**Algorithm 2:** Online Adaptation Process

```
function OnlineAdaptation(MaxIter)
    L ← ExpectedLatencies();
    if not DriftDetected(L) then return;
    R ← ActiveRuleSet();
    if AdjustRoutingRules(R, L) then return;
    # Rule adjust fails due to concentrated ranges
    visit, queue ← Dict(), Queue();
    config ← GetConnectionConfig(DominantTxn());
    model ← SelectModel(DominantTxn());
    for iter from 1 to MaxIter do
        visit[config] ← true;
        UpdateBestConfig(model, config);
        for c in config.unPrunedNeighbors() do
            if not visit[c] then queue.append(c);
        if queue.empty() then break;
        config ← queue.front();
    ApplyConfig(DominantTxn(), GetBestConfig());
```

**Configuration Search.** When Seron detects a load imbalance caused by concentrated query ranges that cannot be resolved via rule adjustments, it initiates a configuration search to tune the connection concurrency. Specifically, Seron aims to find a configuration that minimizes the estimated completion time $T$ of the current workload, defined as

$$T := \frac{\mathrm{L}(p_1, \ldots, p_n)}{p_1 + \ldots + p_n}, \tag{4}$$

where $n$ denotes the number of compute nodes, $p_i$ represents the number of active connections on node $i$, and $\mathrm{L}(p_1, \ldots, p_n)$ is the model-predicted average transaction latency under configuration $(p_1, \ldots, p_n)$. Starting from the current state, Seron iteratively explores and evaluates neighboring configurations (i.e., configurations where any $p_i$ is adjusted by $\pm 1$) to find the optimal setting. To accelerate searching, Seron employs memoization techniques and prunes the search paths where the completion time $T$ increases. Once the process finishes or an iteration limit is reached, Seron reconfigures the connection pool according to the best found connection configuration to maximize database throughput.

The main ideas of the online adaptation process described in this section are summarized in Algorithm 2. Periodically, Seron detects workload drifts based on expected latencies of each compute node. Once hotspot nodes are detected, Seron first tries to adjust the routing rules, and if that process fails due to concentrated query ranges, Seron further checks whether connection reconfiguration can be applied to alleviate load imbalance. Note that for queries offloaded to multiple compute nodes via connection reconfiguration, their expected latencies are still attributed to the original destination so as to maintain consistency of the rule adjustment process.

# 7 EXPERIMENTS

We evaluate Seron and compare it against baseline methods. We first describe the experimental settings (§ 7.1) and present the overall performance evaluation results across various datasets (§ 7.2). We then provide a detailed analysis of the overhead introduced by Seron (§ 7.3), followed by discussions of its adaptability (§ 7.4), scalability (§ 7.5), as well as ablation studies (§ 7.6).

## 7.1 Experimental Settings

**Environments.** We use GaussDB [24] as the backend cloud-native database in our experiments. The system follows a disaggregated architecture consisting of three compute nodes, along with shared memory and storage layers. All experiments are conducted on a dedicated server cluster running EulerOS 2.0 (SP5). Each compute node is provisioned with 10 Intel(R) Xeon(R) E5-2697 v3 CPU cores (2.60 GHz) and 10 GB of DDR4 RAM.

**Implementation.** We implement our Seron system using Python 3.10. The foreground layer wraps the official GaussDB connector library to provide core functionalities, including rule matching, conflict handling, and the user interface. The background layer consists of independent worker processes that communicate with the foreground layer via IPC. We initialize a connection pool with eight connections per compute node for all evaluated routing strategies.

**Datasets.** We use the following benchmarks in our experiments.

- **TPC-C** [2] is a well-known OLTP benchmark consisting of nine relations and five transaction templates. We set the number of warehouses to 100 across all our experiments.
- **YCSB** [9] is a widely adopted benchmark for evaluating the basic CRUD efficiency of database engines. We initialize 10 tables with 1 million tuples each and assess the performance of each router using a customized write-heavy workload with wide-range skewed predicates targeting different tables.
- **Sysbench** [38] is a customizable benchmarking tool that allows users to generate different types of workloads. We use this tool to simulate two diverse scenarios: (i) **Sysbench-CI**, where query predicates utilize only the clustered index that aligns with the physical data layout; and (ii) **Sysbench-MIX**, where query predicates can utilize either clustered or non-clustered indexes. In both scenarios, we initialize a table with 5 million tuples.

**Baselines.** We compare Seron with the following baselines.

- **Round-Robin (RR)**, which is a straightforward strategy that routes the queries in cyclic order to backend compute nodes.
- **Minimal-Load (MIN)**, which is a dynamic heuristic-based strategy that routes each incoming query to the compute node with the minimal number of queueing queries at that time.
- **Fixed-Rule (FIX)**, which utilizes a fixed rule set generated by splitting all index ranges into intervals of equal length. We use this baseline to simulate scenarios where the routing strategies can be pre-defined by the user.
- **Gauss-Router (GR)**, which is the built-in smart router implementation of GaussDB as described in [24]. It constructs page-affinity graphs based on historical workloads and applies graph partitioning algorithms to derive the routing rules.

**Metrics.** We employ the following metrics to evaluate the performance of each routing method in our experiments.

- **Average Execution Time**, which is the actual query execution latency on backend compute nodes and directly quantifies the optimization ability of each routing method.
- **Average End-to-End Time**, which includes the queuing duration (if there are backlogged queries at the router), routing

overhead, and actual database response time (from query submitted to result received).
- **Database Throughput**, measured in Transactions Per Minute (TPM), which reflects the system's overall processing capability.

## 7.2 Performance Evaluation

We first present the performance evaluation results of all tested routing methods. For TPC-C, we follow the standard transaction mix percentages as defined in the benchmark specification. For YCSB, we generate update queries with skewed predicate ranges and targeting different tables to assess cross-table routing efficiency. For Sysbench-CI and Sysbench-MIX, we employ a workload with 50/50 read-write ratio and a 30% hotspot range to simulate more complex data access patterns. We assume queries arrive according to a Poisson process with a rate parameter of $\lambda = 2 \times 10^3$ and report the statistics of the first 30, 000 submitted queries. The overall experiment results are summarized in Figure 9, based on which we make the following observations.

First, Seron consistently achieves superior performance across diverse datasets and workloads, validating the effectiveness of our proposed system. These performance improvements are primarily attributed to better rule generation and conflict handling mechanisms within Seron. Specifically, on the YCSB dataset, Seron accurately identifies relation-based access patterns and generates table-level rules accordingly, achieving maximal physical data isolation while baselines like FIX and GR remain limited to row-level partitioning based on index predicates. The benefits of conflict handling, on the other hand, become more pronounced as the complexity of data access patterns increases. For instance, on TPC-C and Sysbench-CI, where the data is inherently partitionable via clustered indexes, the reduction in average execution time compared to the strongest baseline (21.9% and 2.6%, respectively) is less significant than on Sysbench-MIX (61.7%), where both intra-index and inter-index conflicts arise. This acceleration in query execution, combined with the enhanced load balancing capabilities provided by online adaptation, allows Seron to achieve the best average end-to-end time and overall throughput.

Second, FIX and GR generally outperform RR and MIN as their underlying logical data partitioning effectively reduces cross-node data contention compared to the naive methods. On well-partitioned datasets like TPC-C and Sysbench-CI, GR performs better than FIX due to its enhanced load balancing abilities. However, on datasets with diverse access patterns, GR often fails to identify the optimal routing strategies for the ongoing workload and suffers from consequent data conflicts. For instance, on YCSB, where the predicate ranges are highly skewed, GR does not recognize table access patterns and inevitably distributes hotspot queries across multiple compute nodes. Similarly, on Sysbench-MIX, where non-clustered index predicates incur both intra-index and inter-index conflicts, GR lacks the detection capability, allowing multiple conflicting queries to execute simultaneously. Conversely, the static and imbalanced workload distribution strategy of FIX unintentionally reduces the frequency of cross-node page transfers, causing it to outperform GR in such scenarios. These findings collectively emphasize the necessity of the rule generation and conflict handling components in our Seron system.

Third, note that the mean query end-to-end time is plotted in log-scale, where the queuing time accounts for the vast majority. This is attributed to the high query arrival rate ($\lambda = 2 \times 10^3$) which exceeds the database's processing capability (around $10^3$ average TPS with Seron, and even lower with other baselines). We employ such a high arrival rate in the main experiments to simulate heavy-load scenarios with frequent query conflicts, which inevitably leads to backlogging. We include the routing overhead of Seron in the plot (for baselines, the routing latency is negligible since they directly apply pre-defined rules). As shown in the figure, even with the routing overhead, Seron's response time remains the lowest across all methods. More importantly, most routing procedures are performed while the query is backlogged, indicating that this overhead has the minimal impact on overall throughput as the database is still fully utilized processing other queries. We will provide a detailed breakdown of the routing overhead in § 7.3.

To further investigate Seron's performance under different query arrival rates, we conduct supplementary experiments on TPC-C by varying $\lambda$, as shown in Figure 10. As expected, when the arrival rate drops below the capacity, the end-to-end time quickly approaches the execution time due to less queuing, and Seron maintains a performance edge over GR in most cases. Although the P99 end-to-end time of Seron is slightly higher under low-intensity workloads, this is primarily attributed to the deliberate delays of specific queries for conflict resolution, which can be mitigated by adjusting the maximum timeout limit described in § 5.2.

In summary, Seron significantly enhances end-to-end query performance by generating high-quality routing rules and resolving runtime data conflicts. For Seron's online adaptation capabilities, we will provide more discussions in § 7.4.

## 7.3 Overhead Analysis

As Seron focuses on write-intensive OLTP scenarios, a major concern is the extra overhead it introduces to the end-to-end query latency. As discussed earlier, Seron minimizes this impact by only retaining necessary routing procedures in the foreground, which include query parsing, rule matching, conflict table lookup, and conflict table update. There are delays of certain queries for conflict resolution that add to the end-to-end latency. Table 1 provides a detailed overhead breakdown of all these procedures in milliseconds for both well-partitioned workloads (TPC-C, YCSB, and Sysbench-CI) and conflicting workloads (Sysbench-MIX), where the induced delays are amortized values. As we can see from the table, for well-partitioned workloads, all these procedures combined cost less than 2 ms per query, which is quite affordable as individual clients typically do not exceed this request rate. For conflicting workloads like Sysbench-MIX, the overhead is dominated by the induced delay due to intense query contention, which exceeds 10 ms. However, as we have discussed in § 7.2, this overhead is largely absorbed by the queuing duration with limited impact on overall throughput, and would be vastly reduced under less intense workloads or lower conflict rates. In addition, for extremely short-running queries, Seron leverages runtime statistics to identify their specific patterns and bypasses the conflict table operations entirely to further accelerate the routing process. To conclude, the overhead of Seron contributes to a small portion of the end-to-end latency, and is usually not on the critical path. Regarding spatial overhead, the

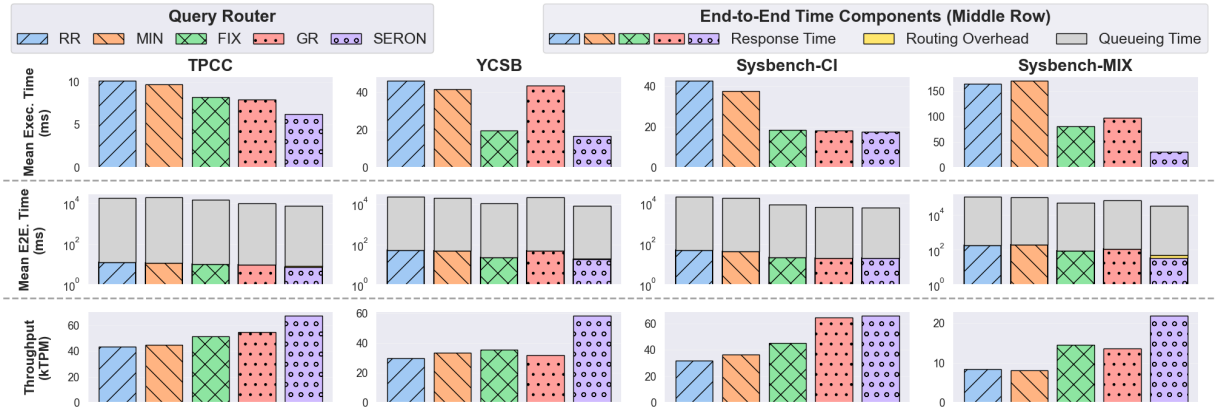


**Figure 9: Performance evaluation results of all routing methods on different datasets.**

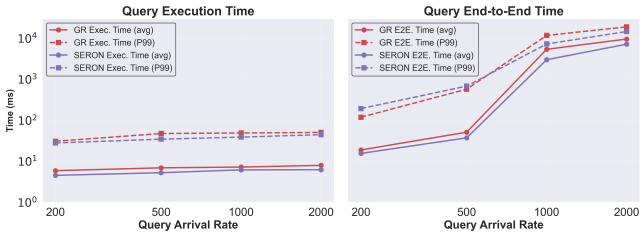


**Figure 10: Impact of arrival rates on Exec./E2E. time.**

additional data structures introduced by SERON (routing rules, the conflict table, and the XGBoost model) typically take less than 1 MB in memory, which is also quite minimal.

## 7.4 Adaptability Evaluation

In this subsection, we evaluate SERON's adaptability to various types of workload drifts. Specifically, we use the TPC-C benchmark and start with the standard transaction mix. At 30 seconds, we restrict the range of warehouse IDs to a small subset to simulate a skewed predicate shift. At 60 seconds, we further constrain the workload to a single warehouse with a 50/50 mix of *NewOrder* and *OrderStatus* transactions. We include GR as a performance reference. Figure 11 illustrates the throughput (TPS) trends for both methods. As we can see, upon the initial drift at 30 seconds, both routers suffer a performance decline. However, SERON successfully adjusts its rules within 10 seconds to re-balance the shifted workload and increase its throughput. Similarly, at the 60-second mark, when the concentrated range renders rule adjustments ineffective, SERON is able to offload the read-only *OrderStatus* transactions to multiple compute nodes through connection reconfiguration. In summary, with the online adaptation component, our SERON system demonstrates superior adaptability to complex workload drifts compared to GR.

We also investigate the generalization ability of the global model in § 6.3. Specifically, we train the model using execution statistics from *NewOrder*, *OrderStatus*, and *Delivery* transactions and assess its Q-Error on unseen transaction patterns across ~100 different connection configurations. As shown in Table 2, the prediction Q-Error remains close to 1 even at the 99th percentile, validating the model's accuracy for unobserved transaction patterns.

## 7.5 Scalability Evaluation

In this subsection, we evaluate the performance of SERON across different scales of the compute layer. Specifically, we use the TPC-C benchmark and compare the overall throughput of GR and SERON as the number of compute nodes ranges from 2 to 5. We follow

**Table 1: Overhead breakdown of SERON foreground router.**

| Procedure | Well-Part. Workloads | Conflicting Workloads |
|---|---|---|
| Query Parsing | ~ 0.3 ms | |
| Rule Matching | ~ 0.2 ms | |
| Conflict Table Lookup | ~ 0.1 ms | |
| Conflict Table Update | ~ 0.2 ms | |
| *Amortized Induced Delay* | < 1 *ms* | < 15 *ms* |
| **Total Overhead** | < 2 **ms** | < 16 **ms** |

**Table 2: Model prediction Q-Error for unseen transactions.**

| Transaction | 50% Q-Error | 90% Q-Error | 99% Q-Error |
|---|---|---|---|
| Payment | 1.0242 | 1.0625 | 1.0878 |
| StockLevel | 1.0407 | 1.0796 | 1.1045 |

the standard transaction mix percentages as in § 7.2. Figure 12 illustrates the evaluation results, which highlight the performance gap between the two routing strategies.

As we can see, SERON maintains a consistent and considerable performance edge over GR across all configurations. Notably, GR reaches its throughput peak at 3 compute nodes and experiences a slight performance degradation with more compute nodes added. This trend occurs because with more compute nodes and concurrent queries, the cross-node communication overhead as well as data conflict penalties also rises, offsetting the benefits of increased execution slots. These impacts can be more pronounced for workloads with complex access patterns such as Sysbench-MIX. In contrast, SERON's throughput continues to scale despite such disadvantages. The reason lies in that, with better rule generation and conflict handling, SERON reduces the frequency of page ownership transfer during query execution, thereby fully leveraging the concurrency gains provided by the expanded compute layer. While the throughput growth rate tapers beyond 3 compute nodes, this is primarily due to the performance bottleneck shifting from data conflicts at the memory layer to disk I/O bandwidth saturation at the storage layer. In summary, SERON demonstrates superior routing efficiency compared to the built-in Gauss Router as the number of backend compute nodes increases.

## 7.6 Ablation Study

We study the individual contributions of SERON's core components. We construct a complex Sysbench workload consisting of 50%


Figure 11: Adaptability evaluation.

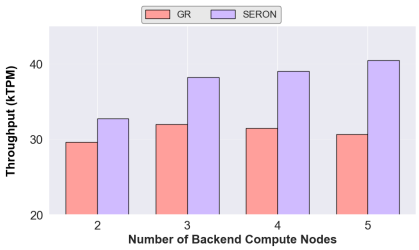


Figure 12: Scalability evaluation.

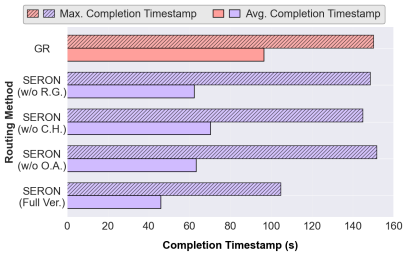


Figure 13: Ablation study of Seron.

skewed update queries utilizing both clustered and non-clustered index predicates, and 50% select queries targeting a concentrated data range, with predicate range hotspots shifting over time. The query arrival rate is kept at $\lambda = 2 \times 10^3$. We compare Seron against three variants: (i) Seron without Rule Generation (*w/o R.G.*); (ii) Seron without Conflict Handling (*w/o C.H.*); and (iii) Seron without Online Adaptation (*w/o O.A.*). We also include GR as a performance reference. Figure 13 illustrates both the maximum and average completion timestamps of the queries. As we can see, removing the three components increases the maximum completion timestamp by 41.7%, 38.3%, and 44.8%, respectively, while the average completion timestamp rises by 35.8%, 52.9%, and 37.7%. These results validate the effectiveness of the individual components in Seron. Specifically, the Rule Generation component enables Seron to handle skewed predicate ranges, while Conflict Handling mitigates cross-node data contention. The Online Adaptation component further optimizes performance by adjusting routing rules and connection configurations dynamically. Notably, even when a component is disabled, Seron's performance remains on par with, if not superior to, that of GR, further underscoring the robustness of our design.

## 8 RELATED WORK

**Multi-primary cloud-native databases.** Multi-primary cloud-native databases have received significant attention in recent years, mainly due to their ability to overcome the write throughput bottleneck inherent in single-primary architectures [1, 4, 13, 41]. Early works focusing on multi-primary shared-storage databases include Oracle RAC [6] and IBM Db2 pureScale [46], which require dedicated networking hardware and lack the scaling ability critical for modern-day cloud deployments. Recent studies design compute-storage disaggregation in cloud-native databases to enhance scalability. For instance, Amazon Aurora-MM [3] adopts Optimistic Concurrency Control protocols to achieve high performance in low-contention scenarios. On the other hand, Taurus-MM [12], PolarDB-MP [48], and GaussDB [24] rely on explicit locking mechanisms and cross-node synchronization techniques to manage data conflicts. *While Seron is currently implemented with GaussDB as the backend database, the core idea of mitigating query conflicts at the router module can be seamlessly integrated into other multi-primary cloud-native architectures to improve overall performance.*

**Database partitioning.** Database partitioning is a fundamental technique for traditional distributed databases that divides large datasets into smaller, more manageable units [5]. In a typical shared-nothing architecture, data is physically partitioned into shards located at different nodes [25], requiring sophisticated designs to handle cross-shard transactions and reduce data migration overhead [11, 34, 37, 40, 52]. In contrast, by leveraging the shared-storage architecture of multi-primary cloud-native databases, Seron is able to perform logical partitioning at the router module, thereby minimizing performance impacts on active queries and achieving greater flexibility. The decoupling of the logical partitioning process from the backend databases also allows Seron to be a non-intrusive solution suitable for various architectures.

**Query scheduling.** Query scheduling is a long-standing research topic in the database community, with its scope ranging from intra-query scheduling [14, 15, 26, 27] to inter-query scheduling [8, 19, 47, 53], and single-server scheduling [36, 43, 51] to multi-server scheduling [20, 28, 30, 33]. Our Seron system focuses on inter-query scheduling for multi-primary cloud-native databases with shared storage. While existing works have addressed similar problems under single-primary settings, using either heuristic-based strategies [33, 35, 42] or learning-based approaches [19], they primarily optimize for read-heavy scenarios and fail to account for the query conflicts associated with write-intensive workloads. Furthermore, these works lack the ability to explicitly model and monitor ongoing data contention. *To our best knowledge, Seron is the first dedicated query router for multi-primary cloud-native databases to address cross-node query conflicts and load balancing simultaneously.*

## 9 CONCLUSION

We study the problem of efficient query routing in multi-primary cloud-native databases with shared storage. We identify the core design challenges and propose Seron, a smart query router that can minimize cross-node conflicts while achieving load balance. Seron incorporates a two-stage rule generation framework that produces candidate routing rules at different granularities, a set of conflict handling mechanisms that actively detect and handle physical data collisions on the fly, and online adaptation techniques that optimize towards load balance. We have integrated Seron into GaussDB and deployed it in production environments to handle real-world workloads. Experiments on multiple datasets verify the efficiency and effectiveness of Seron against existing routing strategies.

## ACKNOWLEDGMENT

Guoliang Li is the corresponding author. This paper was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (62525202, 62232009), Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China(JYB2025XDXM509), Shenzhen (CJGJZD20230724093403007), China Railway Science Research Institute Group Co., Ltd, Zhongguancun Lab and Huawei.

# REFERENCES


[1] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, et al. 2019. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*. 1743–1756.

[2] TPC-C Benchmark. 2010. https://www.tpc.org/tpcc/. [Accessed 2026-01].

[3] Eric Boutin and Steve Abraham. 2019. Amazon Aurora Multi-Master: Scaling Out Database Write Performance. *Amazon ReInvent* (2019).

[4] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, et al. 2021. Polardb serverless: A cloud native database for disaggregated data centers. In *Proceedings of the 2021 International Conference on Management of Data*. 2477–2489.

[5] S. Ceri, M. Negri, and G. Pelagatti. 1982. Horizontal data partitioning in database design. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data* (Orlando, Florida) *(SIGMOD '82)*. Association for Computing Machinery, New York, NY, USA, 128–136. https://doi.org/10.1145/582353.582376

[6] Sashikanth Chandrasekaran and Roger Bamford. 2003. Shared cache-the future of parallel databases. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE Computer Society, 840–840.

[7] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[8] Yun Chi, Hakan Hacígümüş, Wang-Pin Hsiung, and Jeffrey F Naughton. 2013. Distribution-based query scheduling. *Proceedings of the VLDB Endowment* 6, 9 (2013), 673–684.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[10] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[11] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 48–57.

[12] Alex Depoutovitch, Chong Chen, Per-Ake Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, et al. 2023. Taurus MM: bringing multi-master to the cloud. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3488–3500.

[13] Haowen Dong, Chao Zhang, Guoliang Li, and Huanchen Zhang. 2024. Cloud-native databases: A survey. *IEEE Transactions on Knowledge and Data Engineering* 36, 12 (2024), 7772–7791.

[14] Minos N Garofalakis and Yannis E Ioannidis. 1996. Multi-dimensional resource scheduling for parallel queries. *ACM SIGMOD Record* 25, 2 (1996), 365–376.

[15] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. 2014. Deployment of query plans on multicores. *Proceedings of the VLDB Endowment* 8, 3 (2014), 233–244.

[16] Andi Gutmans. 2022. Introducing AlloyDB for PostgreSQL: Free yourself from expensive, legacy databases. https://cloud.google.com/blog/products/databases/introducing-alloydb-for-postgresql.

[17] Chunyue Huang, Shuang Liu, Xinyi Zhang, Wenhao Li, Wei Lu, and Xiaoyong Du. 2025. Chimera: Mitigating Ownership Transfers in Multi-Primary Shared-Storage Cloud-Native Databases. *Proceedings of the VLDB Endowment* 18, 10 (2025), 3368–3381.

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

[19] Yuwei Huang and Guoliang Li. 2024. Laser: Buffer-Aware Learned Query Scheduling in Master-Standby Databases. *Proceedings of the VLDB Endowment* 18, 3 (2024), 743–755.

[20] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 261–276.

[21] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.

[22] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.

[23] Guoliang Li, Haowen Dong, and Chao Zhang. 2022. Cloud databases: New techniques, challenges, and opportunities. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3758–3761.

[24] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. 2024. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3786–3798.

[25] Peng-Ju Liu, Cui-Ping Li, and Hong Chen. 2024. Enhancing storage efficiency and performance: A survey of data partitioning techniques. *Journal of Computer Science and Technology* 39, 2 (2024), 346–368.

[26] Weizheng Lu, Chao Hui, Yunhai Wang, Feng Zhang, Yueguo Chen, Bao Liu, Chengjie Li, Zhaoxin Wu, and Xuye Qin. 2025. Decentralized Actor Scheduling and Reference-based Storage in Xorbits: a Native Scalable Data Science Engine. *Proceedings of the VLDB Endowment* 18, 9 (2025), 2955–2963.

[27] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: a hybrid database for transactional and analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 2530–2542.

[28] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*. 270–288.

[29] Microsoft. 2023. Azure Cloud. https://azure.microsoft.com/en-us.

[30] Beomseok Nam, Minho Shin, Henrique Andrade, and Alan Sussman. 2010. Multiple query scheduling for distributed semantic caches. *J. Parallel and Distrib. Comput.* 70, 5 (2010), 598–611.

[31] Oracle. 2021. Oracle RAC. https://www.oracle.com/database/real-application-clusters/.

[32] Xi Pang and Jianguo Wang. 2024. Understanding the performance implications of the design principles in storage-disaggregated databases. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[33] Thomas Phan and Wen-Syan Li. 2008. Load distribution of analytical query workloads for database cluster architectures. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. 169–180.

[34] Abdul Quamar, K Ashwin Kumar, and Amol Deshpande. 2013. SWORD: scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th international conference on extending database technology*. 430–441.

[35] Uwe Rohm, Klemens Bohm, and H-J Schek. 2001. Cache-aware query routing in a cluster of databases. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 641–650.

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

[37] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.

[38] Sysbench. 2020. https://github.com/akopytov/sysbench. [Accessed 2026-01].

[39] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1493–1509.

[40] Junichi Tatemura, Tao Zou, Jagan Sankaranarayanan, Yanlai Huang, Jim Chen, Yupu Zhang, Kevin Lai, Hao Zhang, Gokul Nath Babu Manoharan, Goetz Graefe, et al. 2023. Progressive Partitioning for Parallelized Query Execution in Google's Napa. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3475–3487.

[41] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[42] Florian Waas and Martin L Kersten. 2000. *Memory aware query scheduling in a database cluster*. CWI (Centre for Mathematics and Computer Science).

[43] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-tuning query scheduling for analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 1879–1891.

[44] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated database systems. In *Companion of the 2023 International Conference on Management of Data*. 37–44.

[45] Ruihong Wang, Jianguo Wang, Stratos Idreos, M Tamer Özsu, and Walid G Aref. 2022. The case for distributed shared-memory databases with RDMA-enabled memory disaggregation. *Proceedings of the VLDB Endowment* 16, 1 (2022), 15–22.

[46] IBM white paper. 2009. Transparent application scaling with IBM DB2 pureScale. https://public.dhe.ibm.com/software/data/sw-library/db2/papers/db2-pure-scale-wp.pdf. Accessed on January 16, 2026.

[47] Ziniu Wu, Markos Markakis, Chunwei Liu, Peter Baile Chen, Balakrishnan Narayanaswamy, Tim Kraska, and Samuel Madden. 2025. Improving DBMS Scheduling Decisions with Accurate Performance Prediction on Concurrent Queries. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4185–4198.

[48] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: a multi-primary cloud-native database via disaggregated shared memory. In *Companion of the 2024 International*

*Conference on Management of Data*. 295–308.

[49] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3385–3397.

[50] Xiangyao Yu. 2025. Disaggregation: A New Architecture for Cloud Databases. *Proceedings of the VLDB Endowment* 18, 12 (2025), 5527–5530.

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

[52] Qiushi Zheng, Zhanhao Zhao, Wei Lu, Chang Yao, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2024. Lion: Minimizing distributed transactions through adaptive replica provision. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2012–2025.

[53] Qiyu Zhuang, Wei Lu, Shuang Liu, Yuxing Chen, Xinyue Shi, Zhanhao Zhao, Yipeng Sun, Anqun Pan, and Xiaoyong Du. 2025. TxnSails: Achieving Serializable Transaction Scheduling with Self-Adaptive Isolation Level Selection. *arXiv preprint arXiv:2502.00991* (2025).