Migration-Free Elastic Storage of Time Series in Apache IoTDB

Rongzhao Chen
Tsinghua University
crz22@mails.tsinghua.edu.cn

Xiangpeng Hu
Tsinghua University
hxp23@mails.tsinghua.edu.cn

Xiangdong Huang
Tsinghua University
huangxdong@tsinghua.edu.cn

Chen Wang
Tsinghua University
wang_chen@tsinghua.edu.cn

Shaoxu Song*
Tsinghua University
sxsong@tsinghua.edu.cn

Jianmin Wang
Tsinghua University
jimwang@tsinghua.edu.cn

ABSTRACT

In distributed time series databases (TSDBs), time series data are typically partitioned by both series and time. These partitions are then allocated to shards, whose replicas determine the storage location, with the leader managing the write load. In Internet of Things (IoT) scenarios, clusters expand as the number of sensors continues to grow. A common approach to re-balancing storage is migrating existing partitions, yet it incurs additional overhead. Fortunately, Time to Live (TTL) is often implemented in time series databases to automatically unload expired data. As a result, dynamically expanding shards rather than migrating existing partitions can also restore storage balance. In addition, the cluster's fault tolerance depends on replica placement schemes, and an expanding cluster complicates this issue. Finally, the intensive write load in IoT scenarios requires balanced leader selection, which becomes difficult due to fault-tolerant placement schemes. To address these IoT challenges, this paper presents the migration-free data partitioning and allocation strategies, a storage-balanced replica placement algorithm with proven fault tolerance, and a write-balanced leader selection algorithm. Our proposals have been deployed in Apache IoTDB since version 1.3. Extensive evaluation of the system demonstrates its superiority in availability and performance.

PVLDB Reference Format:
Rongzhao Chen, Xiangpeng Hu, Xiangdong Huang, Chen Wang, Shaoxu Song, and Jianmin Wang. Migration-Free Elastic Storage of Time Series in Apache IoTDB. PVLDB, 18(6): 1784 - 1797, 2025.
doi:10.14778/3725688.3725706

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/apache/iotdb/.

1 INTRODUCTION

Apache IoTDB [44] is an open-source time series database [29, 33, 42] designed to manage numerous IoT sensors. Given the rapid growth of sensors, the distributed IoTDB is built with scalability. In this paper, we describe how our solutions enhance this elasticity.

*Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725706

1.1 Terminology

We use Figure 1 to illustrate how time series data are typically stored in a TSDB cluster. Each *sensor* is mapped to a *series*, with the time series data partitioned by both series and time. The reason is that (i) customers often deploy over ten million sensors, and (ii) time series data are associated with timestamps. Combining a *series partition* with a *time partition* creates a *data partition*.

To reduce management costs, many distributed systems group data partitions into *shards* [10, 11, 18] as the unit of replication. A shard is a logical set of data partitions. For example, in Figure 1, the shard r_6 consists of data partitions $d_{4,3}$ and $d_{4,4}$. The *replicas* of a shard are its physical copies. In this case, the replicas of shard r_6 are placed at nodes n_7 and n_8. After cluster expansion, the replica placement algorithm determines which nodes host the physical replicas of the dynamically expanded logical shards. When new shards are placed or nodes fail, the leader selection algorithm chooses a *leader* to manage write loads.

1.2 Background

1.2.1 Time to Live. TTL effectively disposes of the massive time series data generated in IoT scenarios. For example, in one of our user's production environments, even with the deployment of the MQTT protocol [28] and a compression algorithm tailored for IoT scenarios [46], the stored data still accumulates to tens of terabytes daily. TTL is common in TSDBs because queries and analytics in IoT contexts typically focus on recent data, leading to the archiving of low-value historical data. As illustrated in Figure 1, data partitions within time partition t_1 expire and are automatically unloaded at the end of time partition t_4.

1.2.2 Gradual Cluster Expansion. As the number of sensors grows significantly, the cluster needs to be expanded accordingly. For instance, a new energy power company may increase the number of sensors in its production environment by 20% over the course of a year. As shown in Figure 1, the growth in sensors leads to series expansion, thus the load on each node increases, which eventually requires cluster expansion by time partition t_3.

1.2.3 Intensive Write Load. It is unsurprising that the hotspots of time series data are the data partitions within the most recent time partition, as numerous sensors continuously collect data. For instance, a steel manufacturer has deployed over 3 million sensors, with 70% of them sampling every second, resulting in an average of 2 million data points written per second. As illustrated in Figure 1, the current time partition is t_4, and the write loads are concentrated on the data partitions within it.

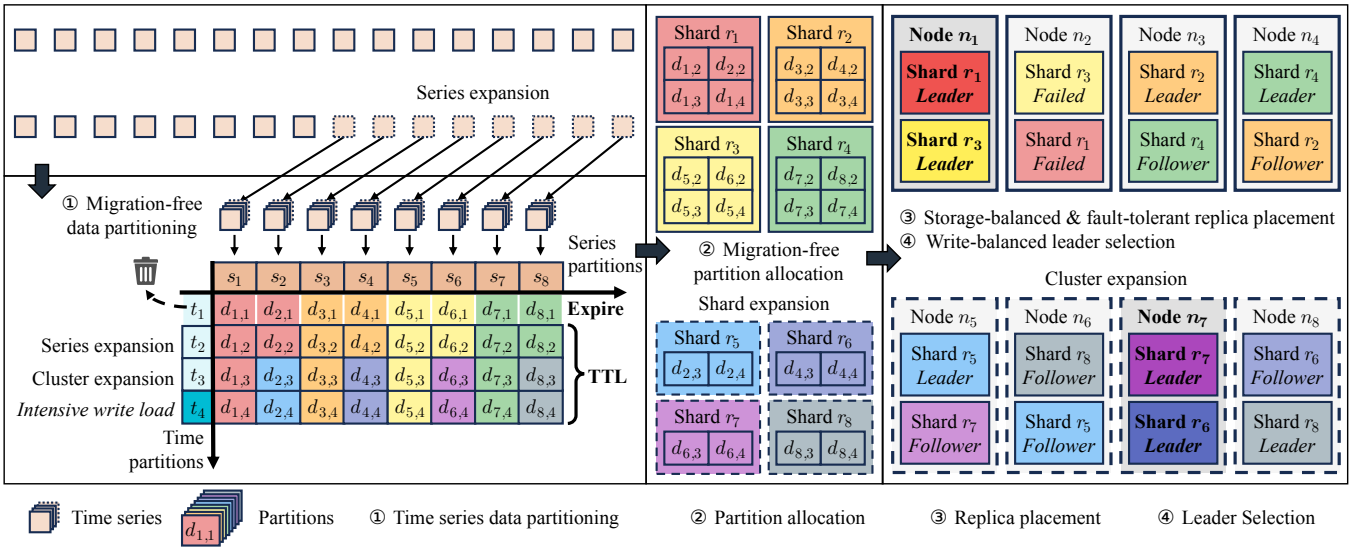

Figure 1: Migration-free elastic storage of ever evolving time series in the IoT scenarios. Dynamic shards match the variously expending series and cluster.

1.3 Challenge

1.3.1 Migration-Free Data Partitioning and Allocation. Data migration is closely integrated with partitioning and allocation strategies. OpenTSDB [32], built upon HBase [19], adjusts its range partitioning as the series expand. Since new series modify the load among partitions, these partitions are either split or merged, then migrated to other shards. Another instance is the Cassandra [23], which employs a dynamic allocation strategy through the use of the consistent hash algorithm [22]. When cluster has expanded and new shards have been added, part of the previous mapping between partitions and shards changes, triggering migration. Hence, the relatively stable partitioning and allocation strategies are required for a migration-free mechanism.

1.3.2 Storage-Balanced and Fault-Tolerant Replica Placement. The shards' replica placement schemes are fundamental to maintaining storage balance and fault tolerance. While a Weighted-Round-Robin (WRR) can achieve storage balance, it compromises single-node fault tolerance. For instance, in the case of nodes n_{1-4} in Figure 1, if node n_2 fails, the load initially handled by n_2 would be transferred solely to n_1, potentially overloading as experimented in Section 5.3.2. COPYSET [9] effectively models and addresses single-node and multi-node fault tolerance, yet it requires a sufficient number of nodes to maintain storage balance. Since gradually expanding a cluster is more cost-effective, demanding a large number of nodes at the initial deployment is impractical. Thus, a sophisticated replica placement algorithm that produces better placement schemes is required. For example, with nodes n_{5-8}, each node would not only hold an equal number of replicas but also share its load to multiple other nodes in the event of failure.

1.3.3 Write-Balanced Leader Selection. The intensive write load necessitates balanced leader distribution. Many consensus protocols define rules for leader election [31, 43], and TimescaleDB [42] implements this through Patroni [35]. However, a replica that initiates a vote request early is more likely to become the leader. This results in random distribution, and can be unbalanced when the number of nodes is initially small. Conversely, deliberate selection achieves a more balanced distribution, but fault-tolerant placement schemes invalidate simple methods like greedy. For example, greedily selecting leaders for shards r_{5-8} in Figure 1 might result in node n_7 holds two leaders. This occurs because each step considers only a subset of nodes, as detailed in Section 4. Thus, a selection algorithm that consistently balances leader distribution is required.

1.4 Contribution

- We implement data partitioning and allocation strategies that leverage TTL. To illustrate the performance of our strategies in storage balance, we prove the linear decrease of the cluster's disk usage standard deviation (std) in Proposition 1.
- We formalize the placement problem and prove its NP-completeness (Theorem 1). Our storage-balanced and fault-tolerant method (Algorithm 1) provides adequate single-node fault tolerance with a replication factor of $\rho = 2$ (Proposition 2) and adapts to $\rho > 2$ (Proposition 3). Additionally, we introduce a probabilistic model to evaluate the cluster's multi-node fault tolerance (Proposition 4).
- We design the write-balanced leader selection approach (Algorithm 2) with guaranteed optimal solutions (Proposition 5).
- Our proposals have been implemented in Apache IoTDB since version 1.3 [3], and deployed in the production environment.
- In comparison to our algorithms (Section 5), the state-of-the-art solution exhibits a 248.4% higher disk usage std in expansion scenario and a 70.0% higher std in disaster scenario. Additionally, the alternative solutions' write throughput std is 322.9% higher in expansion scenario and 60.3% higher in disaster scenario.

Table 1 lists the frequently used notations in this paper.

Table 1: Notations

Symbol	Description
$\mathcal{N}$	The set of cluster nodes.
$\mathcal{R}$	The set of cluster shards.
$\mathcal{H}$	The set of shards' leaders.
s_i	The i-th series partition.
t_j	The j-th time partition.
$d_{i,j}$	The partition consisting of s_i and t_j.
n	The number of nodes in the cluster.
n_i	The i-th node in the cluster.
σ_i	The scatter width of n_i.
ρ	The replication factor, which represents the number of replicas of each shard.
r	The number of shards in the cluster.
r_j	The j-th shard in the cluster, a set composed of ρ nodes where its replicas are located.
ω	The load factor, which represents the number of replicas expected to be held by each node.
ω_i	The actual number of replicas held by n_i.
p_i	The node where the leader of r_i located.
η_i	The number of leaders in n_i.

2 DATA PARTITIONING AND ALLOCATION

In Apache IoTDB, the relationship between "series" and "sensors" is 1-to-1. As illustrated in Figures 1 and 2, all the time series are partitioned in two dimensions. (1) Each time series is assigned to one *series partition* s_i, by hashing the series name. (2) Each time series is further partitioned by time range, as *time partitions* t_j. Although we tend to access the series in timestamp sequence, the partitions on time enable the parallel analysis of a time series, e.g., compute the average in each time partition and aggregate it for the whole series. One may assign all the time series to just one series partition, which also decreases parallelism.

2.1 Data Partitioning

Existing data partitioning strategies are often too mutable, thus we propose a more stable series partitioning operator (Section 2.1.1) to avoid migration. Additionally, we define a time partitioning operator (Section 2.1.2) given the intensive write load of IoT scenarios.

2.1.1 Series Partitioning Operator. As presented in Figures 1 and 2, Apache IoTDB records the mapping between data partitions and shards. It is frequently accessed in read and write workload, and thus needs to be concise with a limited number of series partitions. Meanwhile, limiting the number of series partitions also reduces the cost of allocating data partitions to shards in Section 2.2.2. Therefore, the number of series partitions ϕ is often configured significantly smaller than the number of deployed sensors. The series partitioning operator $o_s(series)$ maps a series to a series partition s_i, where $0 \le i < \phi$, and employs a string hash by default.

Note that each series is hashed to exactly one series partition. In Figure 2, there are 8 series expansion, assigned to 8 series partitions, respectively. As a result, series partitions have similar load, while the mapping of series and series partitions remains stable.

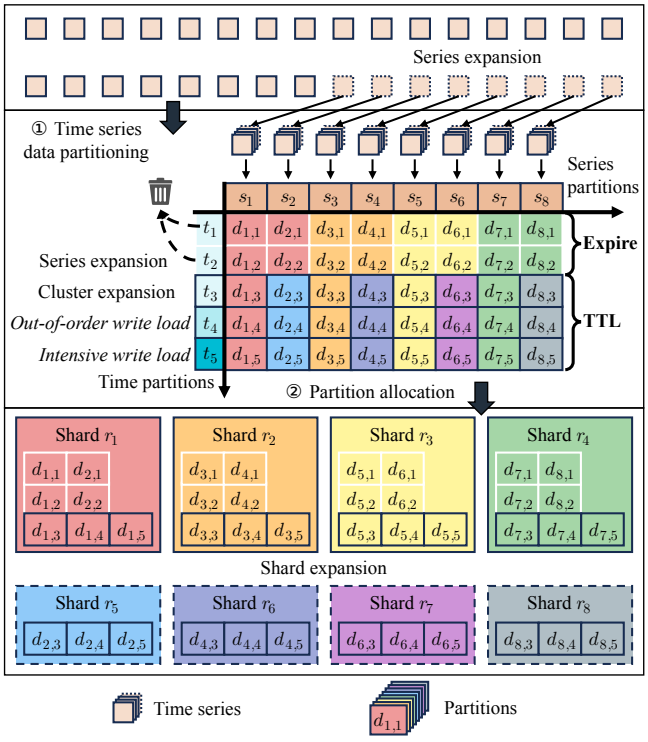

Figure 2: Example of data partitioning and allocation.

2.1.2 Time Partitioning Operator. In the current Apache IoTDB, the system requires that time partitions are cut along the same time for different sensors (series), as shown in Figures 1 and 2. By (randomly) assigning each time series to a series partition by hashing, each series partition tends to have a similar proportion of time series with different reading intervals, i.e., balanced load. Although tailoring each time partition for time series with different reading intervals would help in further load balance, it incurs huge extra cost in allocating and managing the mapping between data partitions and shards. Network fluctuations may cause concurrent writes across different time partitions. However, previous research shows that about 7% of IoT data are out-of-order, with delays of only a few seconds [21]. Since our time partitioning interval spans days, out-of-order data result in brief concurrent writes across time partitions, which minimally affect the overall balance.

2.2 Partition Allocation

Our partitioning strategy divides the time series data into horizontally balanced and vertically distinct hot and cold partitions. Subsequently, when cluster expansion introduces new shards (Section 2.2.1), the partition allocation strategy (Section 2.2.2) is responsible for evenly allocating all series partitions across all shards.

2.2.1 Shard Expansion. New shards' replicas are placed after cluster expansion. We assume that the load factor ω is a constant, and Equation 1 gives the number of shards. The $\omega = 2$ in Figure 1.

$$r = \left\lfloor \frac{n\omega}{\rho} \right\rfloor \tag{1}$$

2.2.2 Allocation Strategy. We group data partitions to shards by the partition allocation strategy. In short, as illustrated in Figure 2, it typically inherits the previous allocation results, and reallocates some data partitions to the newly expanded shards.

During the scaling out process, the replicas and leaders of certain series partitions are relocated to rebalance the storage and the write loads. This relocation is accomplished by sequentially executing the allocation strategy (Section 2.2.2), replica placement (Algorithm 1), and leader selection (Algorithm 2). For example, in Figure 1, (i) when nodes n_{5-8} are added to the cluster at time partition t_3, the allocation strategy creates logical shards r_{5-8} and evenly reallocates all series partitions across the shards r_{1-8}, resulting in the relocation of some series partitions. For example, shard r_2 initially manages the replica and leader of series partition s_4, which is relocated to shard r_6 during the scaling out process. (ii) The replicas of shards r_{5-8} are subsequently placed to nodes n_{5-8} using the placement algorithm. (iii) Finally, the leaders for shards r_{5-8} are selected by the leader selection algorithm.

This strategy trades short-term storage load imbalance to conserve cluster resources, with the duration of the storage imbalance is determined by the TTL. For example, the cluster experiences storage imbalance during time partitions t_{3-4} and regains storage balance by the end of time partition t_5 in Figure 2.

2.3 Load Balance Analysis

Note that unlike the computing-oriented stream processing systems [4, 47, 49], time series databases (TSDBs) are storage-oriented. In TSDBs, a node becomes "read-only" when its disk is full. Given that larger time windows in TSDBs lead to greater data accumulation, it is crucial to monitor the disk usage standard deviation (std) across all nodes to ensure that no node becomes the bottleneck of storage. Therefore, in this section, we provide a detailed analysis of how this standard deviation, denoted as $\mathcal{D}(v)$, evolves over time v in our mechanism that utilizes TTL.

Let a TSDB cluster be ideally load-balanced where data partitions are evenly allocated to all shards, and the replicas of shards are balanced placed across all nodes. Considering that in typical IoT scenarios, sensors sample data at fixed intervals, we can assume that the increment in disk usage per unit time for each node is constant, denoted as C. With TTL in place, the disk usage of all nodes will increase linearly to $\text{TTL} \times C$. Subsequently, the std of disk usage across all nodes is demonstrated in Proposition 1.

Proposition 1. *Assume that the increment in disk usage per unit time for each node is a constant C. Let μ represent the timestamp when the cluster is expanded, having α nodes before μ and $\alpha+\beta$ nodes after μ. With TTL in place, the std of disk usage across all nodes is:*

$$\mathcal{D}(v) = \begin{cases} 0, & v < \mu \vee \mu + TTL < v \\ \dfrac{\sqrt{\alpha\beta}C}{\alpha+\beta} \times (\mu + TTL - v), & \mu \le v \le \mu + TTL \end{cases} \tag{2}$$

Proof. Please see the proof in Section A.1 [1]. □

The replica placement in Section 3 specializes in storage balance, ensuring that each node has enough disk space to store new data. The leader selection in Section 4 focuses on processing balance, optimizing write throughput utilizing CPU and memory resources.

3 REPLICA PLACEMENT

The simple Weighted-Round-Robin (WRR) algorithm lacks fault tolerance as it results in fixed node pairings. While the COPYSET [9] algorithm addresses this issue, it cannot ensure storage balance without sufficient nodes. Let us first introduce some preliminaries that model fault tolerance (Section 3.1). We then propose a more appropriate problem definition (Section 3.2) and the corresponding choreographed placement algorithm (Section 3.3) for IoT scenarios. Finally, Section 3.4 provides comprehensive proofs regarding the fault tolerance of our placement algorithm.

3.1 Preliminaries

An existing study [9] introduces scatter width and Copyset as key measures of fault tolerance. We reference them as follows:

Definition 1 (scatter width). *Let $\mathcal{N}$ denote nodes set, $\mathcal{R}$ denote shards set, scatter width σ_i of node n_i is given by Equation 3.*

$$\sigma_i = \left| \bigcup_{n_i \in r_j} r_j \right| - 1 \tag{3}$$

For the notations used to define the scatter width in Equation 3, n_i represents the i-th node in the cluster, and r_j denotes the placement scheme for the j-th shard, which refers to the set of nodes holding the replicas of the j-th shard. Thus, in Equation 3, $\cup_{n_i \in r_j} r_j$ refers to the union of the node sets r_j that contain node n_i. In Figure 1, the scatter width σ_1 of node n_1 is 1 since $n_1 \in r_1, r_3$ and $|r_1 \cup r_3| - 1 = 1$. Similarly, the scatter width σ_5 of node n_5 is 2 since $n_5 \in r_5, r_7$ and $|r_5 \cup r_7| - 1 = 2$. The Copyset paper does not consider the load-sharing property of scatter width in the event of single-node failures, i.e., the scatter width of a node can represent the number of other nodes that take on its write load during failure.

A Copyset is an independent replica placement scheme. In Figure 1, nodes n_{1-4} have two Copysets: $\{n_1, n_2\}, \{n_3, n_4\}$, while nodes n_{5-8} have four Copysets: $\{n_5, n_6\}, \{n_5, n_7\}, \{n_6, n_8\}, \{n_7, n_8\}$. Using more Copysets increases the scatter width; however, this does not imply that all possible Copysets should be used. In a cluster with n nodes and a replication factor ρ, exhausting all $\binom{n}{\rho}$ Copysets would mean that any concurrent failure of ρ nodes would render a shard unavailable, reducing the cluster's multi-node fault tolerance.

3.2 Replica Placement Problem

As the cluster expands, more replicas will be placed. In this process, our goal is to maximize the scatter width sum of cluster nodes to improve their load-sharing property, while ensuring that the number of replicas held by each node does not exceed the load factor ω, thereby maintaining storage balance. This problem is modeled in Section 3.2.1, and its hardness is analyzed in Section 3.2.2.

3.2.1 Problem Definition. An undirected graph that excludes multiple edges represents the relation between nodes and shards. Figure 3 illustrates a cluster with 8 nodes and a replication factor $\rho = 3$. Each node n_i is represented by a vertex n_i, and each shard r_j is denoted by a group of edges (n_ι, n_κ) where $n_\iota, n_\kappa \in r_j \wedge n_\iota \neq n_\kappa$. The degree of vertex n_i is the scatter width σ_i of node n_i, as multiple edges are excluded. If nodes n_{1-4} and shards r_{1-4} in Figure 1 are converted similarly, only 2 edges remain: (n_1, n_2) and (n_3, n_4).

Formally, we convert the shard set $\mathcal{R}$ into an adjacency matric $\mathcal{A} = (a_{ij})_{n\times n}$ and introduce the indicator function [], which is 1 iff. the condition inside holds true; otherwise is 0. Let $x_{ik} \in \{0, 1\}$, $x_{ik} = 1$ iff. $n_i \in r_k$. Thus, $a_{ij} = [i \neq j \wedge \sum_{k=1}^{r} x_{ik} \cdot x_{jk} \geq 1]$, meaning that there exists at least one shard r_k s.t. $n_i, n_j \in r_k$. Let $\mathcal{W} = (\omega_i)_{1\times n}$, and assume that m shards need to be placed. We define the replica placement problem $RPP(\mathcal{N}, \mathcal{A}, \mathcal{W}, \omega, \rho, m)$ as follows:

Problem 1 (replica placement problem). *Given $\mathcal{N}, \mathcal{A}, \mathcal{W}, \omega, \rho, m$, find replica placement schemes $r'_1, r'_2, \cdots, r'_m$ that*

$$\textit{maximize} \quad \Sigma = \sum_{i=1}^{n} \sigma_i = \sum_{i=1}^{n} \sum_{j=1\wedge j\neq i}^{n} [(a_{ij} + \sum_{k=1}^{m} x'_{ik} \cdot x'_{jk}) \geq 1],$$

$$\textit{subject to} \quad \omega_i + \sum_{j=1}^{m} x'_{ij} \leq \omega; \forall n_i \in \mathcal{N},$$

$$\textit{where} \quad x'_{ij} = 1 \textit{ iff. } n_i \in r'_j; x'_{ij} \in \{0, 1\}.$$

In Problem 1, m is the number of shards that need to be placed. In the constraint inequation $\omega_i + \sum_{j=1}^{m} x'_{ij} \leq \omega$, the + represents the sum of the number of replicas already placed on node n_i, i.e., ω_i, and the number of replicas to be placed on node n_i, i.e., $\sum_{j=1}^{m} x'_{ij}$. Since this summation should be bounded by the load factor ω, we further enforced the $\leq$ condition. Therefore, m does not need to be sufficiently large, as long as the constraint is feasible.

3.2.2 Hardness Analysis. To analyze NP complexity, define the decision problem as $RPDP(\mathcal{N}, \mathcal{A}, \mathcal{W}, \omega, \rho, m, \Delta)$, which asks whether there exist m schemes s.t., after placing them, the scatter width sum Σ' satisfies $\Sigma' - \Sigma \geq \Delta$. Unfortunately, this is NP-complete.

Theorem 1. *$RPDP(\mathcal{N}, \mathcal{A}, \mathcal{W}, \omega, \rho, m, \Delta)$ is NP-complete.*

Proof. Please see the proof in Section A.2 [1]. □

3.3 Partite Graph Placement

The complexity arises from the unpredictability in the number of nodes introduced by gradually expanding clusters, and different users require varying replication factors to ensure diverse levels of fault tolerance. Thus, we propose a valuation function in Section 3.3.1 and a coordinated heuristic search algorithm in Section 3.3.2.

3.3.1 Valuation Function. If a pair of vertices is already adjacent, including them in the new placement scheme does not increase their scatter width. In conjunction with the storage balance constraint, we propose the valuation function as in Equation 4.

$$f(\mathcal{N}, \mathcal{A}, \mathcal{W}) = \left(\sum_{n_\iota, n_\kappa \in \mathcal{N}} a_{\iota\kappa}, \sum_{n_\iota \in \mathcal{N}} \omega_\iota \right) \tag{4}$$

The input is a node set $\mathcal{N}$, an adjacency matrix $\mathcal{A}$ and a load vector $\mathcal{W}$. The output is a tuple (μ, ν) composed of two nonnegative integers. Here, μ is twice the number of edges that are covered by nodes in $\mathcal{N}$, and ν is the number of replicas held by nodes in $\mathcal{N}$. Since each covered edge indicates that the corresponding scatter width has contributed to this pair of nodes, we expect μ to be as small as possible. To ensure that each node owns an equal number of replicas, we also aim for ν to be as small as possible. Thus, we define $(\mu_1, \nu_1) < (\mu_2, \nu_2)$ iff. $\mu_1 < \nu_2 \vee (\mu_1 = \mu_2 \wedge \nu_1 < \nu_2)$.

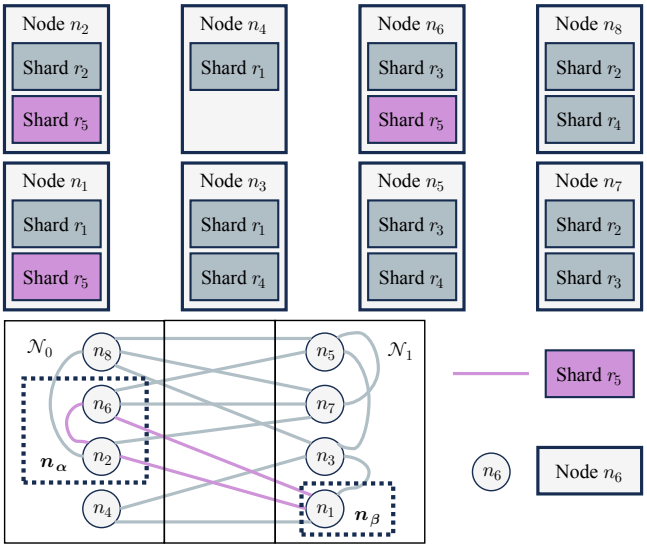

Figure 3: Example of Algorithm 1 for the new shard r_5.

Algorithm 1: Partite Graph Placement (PGP)

Input: Node set $\mathcal{N}$, adjacency matrix $\mathcal{A}$, load vector $\mathcal{W}$, load factor ω and replication factor ρ
Output: The next shard's replica placement scheme r'

1 $\mathrm{N} \leftarrow PartiteGraph(\mathcal{N}); \mathbf{n}_\alpha, \mathbf{n}_\beta, \mathcal{N}_\alpha \leftarrow \emptyset$;
2 **foreach** $\mathcal{N}_i \in \mathrm{N}$ **do**
3 $\mathbf{n}^*_\alpha \leftarrow \lfloor \frac{\rho}{2} \rfloor$ nodes n_j in $\mathcal{N}_i$ with minimum ω_j;
4 $n_\iota \leftarrow$ a node in $\mathcal{N}_i$ s.t. $f(\mathbf{n}^*_\alpha \cup n_\iota, \mathcal{A}, \mathcal{W})$ is minimum;
5 $\mathbf{n}^*_\alpha \leftarrow \mathbf{n}^*_\alpha \cup n_\iota$;
6 **if** $f(\mathbf{n}^*_\alpha, \mathcal{A}, \mathcal{W}) < f(\mathbf{n}_\alpha, \mathcal{A}, \mathcal{W})$ **then**
7 $\mathbf{n}_\alpha \leftarrow \mathbf{n}^*_\alpha$;
8 $\mathcal{N}_\alpha \leftarrow \mathcal{N}_i$;
9 **foreach** $\mathcal{N}_j \in \mathrm{N} \setminus \{\mathcal{N}_\alpha\}$ **do**
10 $n_\kappa \leftarrow$ a node in $\mathcal{N}_j$ s.t. $f(\mathbf{n}_\alpha \cup n_\kappa, \mathcal{A}, \mathcal{W})$ is minimum;
11 $\mathbf{n}_\beta \leftarrow \mathbf{n}_\beta \cup n_\kappa$;
12 $r' \leftarrow \mathbf{n}_\alpha \cup \mathbf{n}_\beta$;
13 **return** r';

3.3.2 Placement Algorithm. To pre-prune and achieve storage balance constraint, the line 1 of Algorithm 1 eliminates those $\omega_i = \omega$ nodes, so that $\forall n_i \in \mathcal{N}, \omega - 1 \leq \omega_i \leq \omega$ is ultimately satisfied after placing $r = \left\lfloor \frac{n\omega}{\rho} \right\rfloor$ shards. The graph is then divided to efficiently enumerate potential schemes, and separately estimate the scatter width increment. For instance, in Figure 3, 8 nodes are divided into subsets of nodes $\mathcal{N}_0, \mathcal{N}_1$. Specifically, the i-th subsets of nodes can be obtained by $\mathcal{N}_i = \{n_j : n_j \in \mathcal{N} \wedge \omega_j < \omega \wedge j \bmod \lceil \frac{\rho}{2} \rceil = i\}$, and we limit $0 \leq i < \lceil \frac{\rho}{2} \rceil$ to construct $\lceil \frac{\rho}{2} \rceil$ subsets of nodes.

Algorithm 1 searches for next placement scheme r' in two steps. In lines 2-8, for each subsets, it selects $\lfloor \frac{\rho}{2} \rfloor$ nodes that hold the fewest number of replicas into $\mathbf{n}^*_\alpha$ to balance storage load. Then, it picks a n_ι that minimizes $f(\mathbf{n}^*_\alpha \cup n_\iota, \mathcal{A}, \mathcal{W})$ and adds it to $\mathbf{n}^*_\alpha$,

expecting that n_ι increases the scatter width by 1 for others in $\mathbf{n}^*_\alpha$. Among all these $\mathbf{n}^*_\alpha$, the one with the lowest valuation function is chosen as $\mathbf{n}_\alpha$, which comprises $\lfloor \frac{\rho}{2} \rfloor + 1$ nodes for the result. In lines 9-11, it enumerates other subsets except $\mathcal{N}_\alpha$. From each subsets, it selects a n_κ that minimizes $f(\mathbf{n}_\alpha \cup n_\kappa, \mathcal{A}, \mathcal{W})$, expecting that n_κ increases the scatter width by 1 for all nodes inside $\mathbf{n}_\alpha$. Finally, it obtains $\mathbf{n}_\beta$ that includes $\lceil \frac{\rho}{2} \rceil - 1$ nodes, and returns $r' = \mathbf{n}_\alpha \cup \mathbf{n}_\beta$ as the scheme for next shard. Expectedly, each node in r' increases by at least $\lceil \frac{\rho}{2} \rceil$ scatter width, since each node in $\mathbf{n}_\alpha$ gains 1 from n_ι and gains $\lceil \frac{\rho}{2} \rceil - 1$ from $\mathbf{n}_\beta$, while each node in $\mathbf{n}_\beta$ gains $\lfloor \frac{\rho}{2} \rfloor + 1 \geq \lceil \frac{\rho}{2} \rceil$ from $\mathbf{n}_\alpha$. Algorithm 1 runs efficiently, since sorting costs $O(n \log n)$ and invoking the valuation function costs $O(\rho^2)$.

When searching the placement scheme for shard r_5 in Figure 3, the algorithm first selects n_2 in $\mathcal{N}_0$ because ω_2 is the smallest. Then, it appends n_6 into $\mathbf{n}_\alpha$ because $f(\{n_2, n_6\}, \mathcal{A}, \mathcal{W}) = (0, 2)$ is the smallest tuple. Finally, it selects n_1 from $\mathcal{N}_1$ and adds it to $\mathbf{n}_\beta$ because $f(\{n_2, n_6\} \cup \{n_1\}, \mathcal{A}, \mathcal{W}) = (0, 3)$ is also the smallest tuple. The placement scheme for shard r_5 is thus $r' = \{n_2, n_6, n_1\}$.

3.4 Fault Tolerance Analysis

We show in Section 3.4.1 that Algorithm 1 can provide sufficient scatter width, thus improve the fault tolerance when a single-node fails. Nevertheless, Algorithm 1 uses different placement schemes to fulfill this objective, which may compromise the fault tolerance when multi-node fail as mentioned in Section 3.1. Thus, Section 3.4.2 analyzes multi-node fault tolerance of Algorithm 1.

3.4.1 Single-Node Fault Tolerance. To quantify the scatter width provided by Algorithm 1, we define the lower bound of scatter width ratio as $\breve{\sigma}_n = \frac{\sum_{i=1}^{n} \sigma_i}{\sum_{i=1}^{n} \sigma_i^*} \geq \min_{n_i \in N} \left(\frac{\sigma_i}{\sigma_i^*} \right)$. The σ_i^* represents the maximum possible scatter width of node n_i. When the replication factor $\rho = 2$, Algorithm 1 ensures adequate scatter width, as shown in Proposition 2. Furthermore, Proposition 3 shows that Algorithm 1 also provides enough scatter width for $\rho > 2$ scenarios.

If all r shards are optimally placed, the scatter width of any node n_i reaches its maximum value—$\sigma_i^* = (\rho - 1)\omega$. Instead of painstakingly finding optimal schemes, Algorithm 1 aims to construct more "good" schemes that increase $\lceil \frac{\rho}{2} \rceil$ scatter width for each involved node to the greatest extent possible. Fortunately, the sufficient condition for the existence of a good scheme is looser than that for an optimal scheme, as given by Lemmas 2, 3 and 4. Assuming that any node is within at least $\breve{\omega}$ good schemes, the lower bound of $\breve{\sigma}_n$ can be calculated as $\breve{\sigma}_n \geq \frac{\lceil \frac{\rho}{2} \rceil \breve{\omega}}{(\rho-1)\omega}$. Lemma 1 provides $\breve{\omega}$'s lower bound.

Lemma 1. *Assuming that at least $\breve{r}$ of all r placement schemes generated by Algorithm 1 are good, and for each node, assuming that it is included in at least $\breve{\omega}$ good placement schemes, then $\breve{\omega} \geq \breve{r} \times \frac{\lfloor \frac{\rho}{2} \rfloor}{n}$.*

Proof. Please see the proof in Section A.3 [1]. □

The more good schemes leads to the larger $\breve{\omega}$. While the degree of a node indicates its current scatter width, the complement degree—the number of nodes not adjacent to the specified node—implies if this node is capable of being involved in a good scheme.

$$g(n_\gamma, \mathcal{N}, \mathcal{A}) = \sum_{n_\theta \in \mathcal{N}} (1 - a_{\gamma\theta}) \tag{5}$$

Define the complement degree in Equation 5. The input is a node n_γ, a node set $\mathcal{N}$ and an adjacency matrix $\mathcal{A}$. The output is the number of nodes in $\mathcal{N}$ that are non-adjacent to n_γ. Lemmas 2, 3 are respectively the sufficient conditions for the existence of $\mathbf{n}_\alpha, \mathbf{n}_\beta$.

Lemma 2. *After the Algorithm 1 selects $\mathbf{n}^*_\alpha$ in line 3, there must $\exists n_\iota \in \mathcal{N}_i$ s.t. $g(n_\iota, \mathbf{n}^*_\alpha, \mathcal{A}) = \lfloor \frac{\rho}{2} \rfloor$ when $|\mathcal{N}_i| > \lfloor \frac{\rho}{2} \rfloor^2 \omega - \lfloor \frac{\rho}{2} \rfloor$.*

Proof. Please see the proof in Section A.4 [1]. □

Lemma 3. *After the Algorithm 1 selects $\mathbf{n}_\alpha$, there must $\exists n_\kappa \in \mathcal{N}_j$ s.t. $g(n_\kappa, \mathbf{n}_\alpha, \mathcal{A}) = \lfloor \frac{\rho}{2} \rfloor + 1$ when $|\mathcal{N}_j| > \left(\lfloor \frac{\rho}{2} \rfloor + 1 \right)^2 \omega$.*

Proof. Please see the proof in Section A.5 [1]. □

According to Lemmas 2, 3, the more nodes within each subset, the higher the likelihood of a good scheme existing. As "filled" nodes that own ω replicas are removed to meet the storage balance constraint, Lemma 4 shows the growth rate of filled nodes.

Lemma 4. *After Algorithm 1 placed r shards, the upper bound of the number of filled nodes is $\hat{n} \leq r \times \frac{\rho}{\omega}$.*

Proof. Please see the proof in Section A.6 [1]. □

In the case of $\rho = 2$, after the graph division process of Algorithm 1, there is only a subset of nodes $\mathcal{N}_0 = \mathcal{N}$. Thus, the Lemma 3 can be ignored, and the corresponding proposition is:

Proposition 2. *Let $\mathcal{N}$ denote node set, ω denote load factor. For replication factor $\rho = 2$, after Algorithm 1 places $r = \left\lfloor \frac{n\omega}{\rho} \right\rfloor$ shards, the limit of the lower bound of $\breve{\sigma}_n$ is given by*

$$\lim_{n \to \infty} \breve{\sigma}_n \geq \frac{1}{2}. \tag{6}$$

Proof. Please see the proof in Section A.7 [1]. □

When $\rho = 2$, the limit of $\breve{\sigma}_n$ fulfills single-node fault tolerance. As the failed node's load is carried by leaders, and Proposition 5 shows that the number of leaders held by each node when $\rho = 2$ is $\frac{\omega}{2}$; although the possible $\max(\sigma_i^*) = \omega$, Algorithm 1 establishes a lower bound of $\frac{\omega}{2}$, adequate for sharing the failed node's leaders. Next, we present that Algorithm 1 also adapts to $\rho > 2$ scenarios:

Proposition 3. *Let $\mathcal{N}$ denote node set, ω denote load factor. For replication factor $\rho > 2$, after Algorithm 1 places $r = \left\lfloor \frac{n\omega}{\rho} \right\rfloor$ shards, the limit of the lower bound of $\breve{\sigma}_n$ is given by*

$$\lim_{n \to \infty} \breve{\sigma}_n \geq \frac{1}{4 \times \lceil \frac{\rho}{2} \rceil}. \tag{7}$$

Proof. Please see the proof in Section A.8 [1]. □

Given that the conditions outlined in Propositions 2 and 3 require deploying a sufficient number of nodes, we conducted a simulation of Algorithm 1 under the limitation of $n \in [1, 100]$, $\rho \in [2, 5]$, and $\omega \in [1, 10]$. As illustrated in Figure 4, Algorithm 1 regularly attains at least 50% of the theoretically optimal solution, and the average ratio surpasses 87%. Consequently, the minimum scatter width guaranteed by Algorithm 1 provides adequate support for single-node fault tolerance, and empirical evaluations will be conducted in Section 5.3.2 to validate these findings.

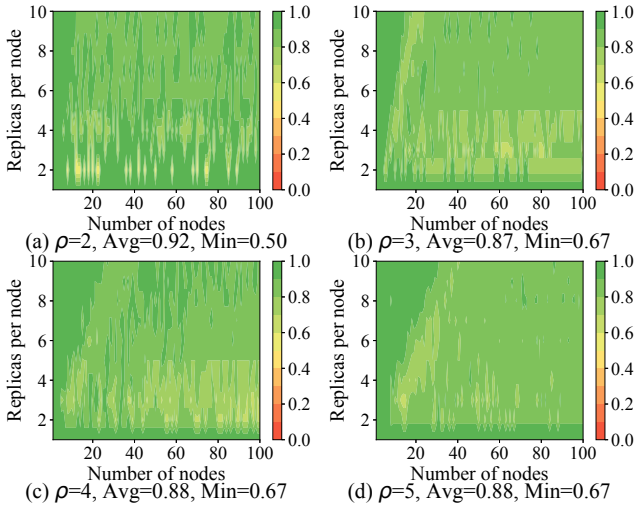

Figure 4: The lower bound of the scatter width ratio $\breve{\sigma}_n$ produced by Algorithm 1.

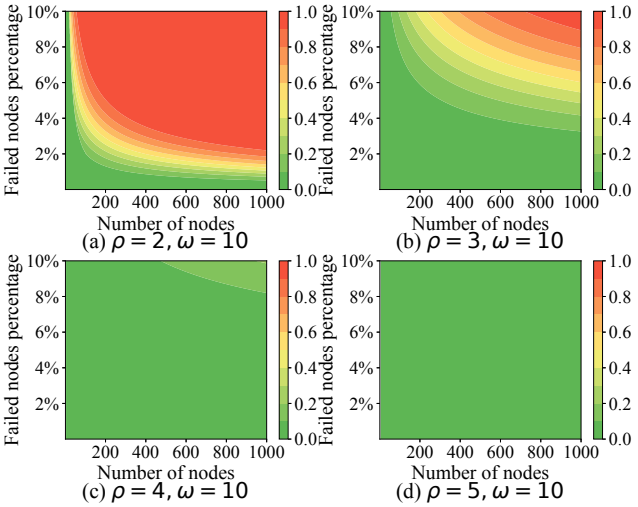

Figure 5: The probability distribution that the unavailable shards occur $P(X(n,m) \geq 1)$ produced by Algorithm 1.

3.4.2 Multi-Node Fault Tolerance. To the best of our knowledge, no probabilistic model measures the likelihood of shard unavailability when multiple nodes fail simultaneously. If ρ nodes fail and match the placement of a shard, the shard becomes "unavailable." Previous research shows that large-scale disasters, involving the simultaneous failure of about 1% of cluster nodes, occur annually in clusters of a certain scale [14, 20]. Thus, we assess this probability and show the multi-node fault tolerance of Algorithm 1.

Proposition 4. *Let $\mathcal{N}$ denote nodes set, ρ denote replication factor, r denote the number of employed disparate placement schemes. Let $X(n,m)$ denote the number of unavailable shards when m nodes among n cluster nodes fail simultaneously. The probability that at least one unavailable shard occurs is given by*

$$\lim_{n\to\infty} P(X(n,m) \geq 1) = 1 - e^{-\binom{m}{\rho} \times \frac{r}{\binom{n}{\rho}}}. \tag{8}$$

Proof. Please see the proof in Section A.9 [1]. □

We use Proposition 4 to illustrate the multi-node fault tolerance of Algorithm 1, with $\omega = 10, \rho \in [2,5], n \in [1,1000]$, and the number of failed nodes ranging from $[0, 10\%]$, as shown in Figure 5. When $\rho = 2$, the cluster can endure 1% of its nodes fail, and this tolerance improves significantly as the ρ increases. In Figures 5(b), (c), and (d), the probability of unavailable shards is minimal, due to the significantly lower number of employed placement schemes compared to the total number of possible schemes.

3.4.3 Fault Tolerance and Storage Balance Comparison. Our PGP algorithm demonstrates exceptional performance in single-node fault tolerance (Section 3.4.1), as its lower bound of scatter width, $\breve{\sigma}_n$, is rigorously proven by Propositions 2 and 3. In addition, the upper bound on the number of pairwise distinct placement schemes, $\dot{r}$, generated by PGP is constrained by Equation 1. This ensures the multi-node fault tolerance (Section 3.4.2) of PGP, as derived from Proposition 4. With the storage balance constraint in place (Algorithm 1), PGP guarantees that the upper bound on the difference in the number of replicas held by each node, Δ_ω, is minimal even within an ever-expanding cluster.

Table 2: Overview of replica placement algorithms, with PGP being the most fault-tolerant and storage-balanced.

Algorithm	$\breve{\sigma}_n$	$\dot{r}$	Δ_ω
PGP (ours)	$\geq 1/\left(4 \times \left\lceil \frac{\rho}{2} \right\rceil\right)$	$\leq n\omega/\rho$	≤ 1
WRR	$\geq 1/\omega$	$\leq n/\rho$	≤ 1
COPYSET [9]	-	$\leq n\omega/\rho$	-
TIERED [8]	-	$\leq n\omega/\rho$	-
GEMINI [45]	$\geq 1/\omega$	$\leq n/\rho$	-
HYDRA [24]	-	$\leq n\omega/\rho$	-

Table 2 compares different replica placement algorithms, where n is the number of nodes, r is the replication factor, and ω is the load factor, as defined in Table 1. Given that $\rho \ll \omega$, a very common scenario in real deployments, the $\breve{\sigma}_n$ of PGP is significantly larger than that of Weighted-Round-Robin (WRR) and GEMINI. In contrast, the $\breve{\sigma}_n$ of COPYSET, TIERED, and HYDRA is not derivable due to their inherent randomness. Since we adopt the design of $\omega \ll n$ from COPYSET, the $\dot{r}$ of PGP is already sufficiently small. Lastly, our PGP algorithm is among the few that ensure storage balance by guaranteeing $\Delta_\omega \leq 1$.

4 LEADER SELECTION

The following selection process for shards r_{5-8} in Figure 1 shows GREEDY's myopia: (i) leader of r_5 on n_5, (ii) leader of r_6 on n_7, (iii) leader of r_7 on n_7, (iv) leader of r_8 on n_8. At (iii), selecting the leader of r_7 on either n_5 or n_7 leads to an unbalanced distribution. To address this, we formalize the leader selection problem (4.1), design a selection algorithm (4.2), and prove its optimality (4.3).

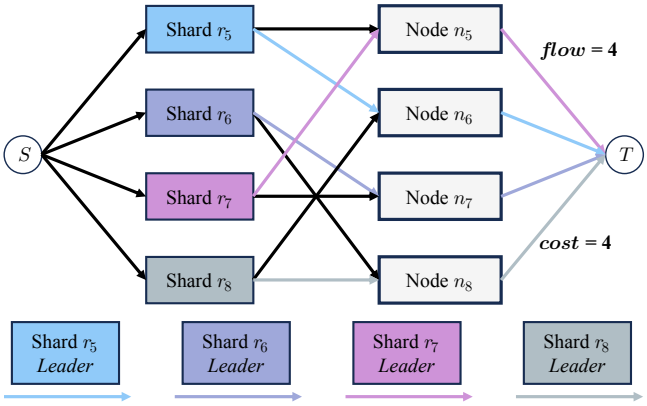

Figure 6: Example of Algorithm 2 for leader selection.

4.1 Leader Selection Problem

To embody the selection process, for shards r_{5-8} and nodes n_{5-8} in Figure 1, we use directed edges to connect each shard with nodes that owns its replicas in Figure 6. Formally, let η_i denote the number of leaders held by n_i, p_j is the leader of r_j. Let $y_{ij} \in \{0, 1\}$, $y_{ij} = 1$ iff. $p_j = n_i$, meaning that the leader of r_j is the replica that resides on node n_i. The leader selection problem is as follows:

Problem 2 (leader selection problem). *Given $\mathcal{N}, \mathcal{R}$, select the leader replicas set $\mathcal{H} = \{p_1, p_2, \cdots, p_r\}$ that*

$$\begin{aligned} &\textit{minimize} && \Sigma = \sum_{i=1}^{n} \eta_i^2 = \sum_{i=1}^{n} (\sum_{j=1}^{r} y_{ij})^2, \\ &\textit{subject to} && p_i \in r_i; \forall r_i \in \mathcal{R}, \\ &\textit{where} && y_{ij} = 1 \textit{ iff. } p_j = n_i; y_{ij} \in \{0, 1\}. \end{aligned}$$

4.2 Cost Flow Selection

Adding a source S and a sink T in Figure 6 reduces Problem 2 to the cost flow problem [15] that can be solved by combining Dinic [12] and SPFA [27] algorithms. Next, we give the corresponding edge modeling (Section 4.2.1) and selection algorithm (Section 4.2.2).

4.2.1 Edge Modeling. Employing the cost flow algorithm requires to design the capacity and cost of each edge. In Figure 6, for edges between the source and shard vertices, and between shard vertices and node vertices, assigning a capacity of 1 and a cost of 0 is sufficient, as each shard requires only 1 leader. For edges between node vertices and the sink, we need a cost function whose minimum value ensures balanced distribution. The definitions are:

- $\widetilde{Q} = \{q_{S,r_i} : r_i \in \mathcal{R}\}$: Connects source vertex S with shard vertices in R to limit the number of leaders in each shard.
- $\widetilde{B} = \{b_{r_i,n_j} : r_i \in \mathcal{R} \wedge n_j \in r_i\}$: Connects shard vertices with node vertices to denote the leader selection of each shard.
- $\widetilde{L} = \{l_{n_j,T,k} : n_j \in \mathcal{N} \wedge 1 \le k \le \omega\}$: Connects node vertices with sink vertex T, indicating η_j in the final distribution, where the k-th edge that connects n_j to T has a cost of $\delta(k) = 2k - 1$.

4.2.2 Selection Algorithm. Algorithm 2 constructs the above edges in lines 2-9, where the function $MakeEdge(v_i, v_j, \delta)$ creates an edge that connects v_i and v_j with a capacity of 1 and a cost of δ. Next, it uses the cost flow algorithm to obtain the min-cost max-flow of $\widetilde{G}$ in line 10. Finally, it collects the leader distribution from $\widetilde{B}$ in lines 11-15, where $flow[e]$ is the actual flow of an edge $e \in \widetilde{B}$.

Algorithm 2: Cost Flow Selection (CFS)

Input: A node vertices set $\mathcal{N}$ and a shard vertices set $\mathcal{R}$.
Output: The optimal shards' leaders set $\mathcal{H}$.

1 $\widetilde{V} \leftarrow \{S\} \cup \mathcal{N} \cup \mathcal{R} \cup \{T\}, \widetilde{Q}, \widetilde{B}, \widetilde{L} \leftarrow \emptyset$;
2 **foreach** $r_i \in \mathcal{R}$ **do**
3 $q_{S,r_i} \leftarrow MakeEdge(S, r_i, 0), \widetilde{Q} \leftarrow \widetilde{Q} \cup q_{S,r_i}$;
4 **foreach** $r_i \in \mathcal{R}$ **do**
5 **foreach** $n_j \in r_i$ **do**
6 $b_{r_i,n_j} \leftarrow MakeEdge(r_i, n_j, 0), \widetilde{B} \leftarrow \widetilde{B} \cup b_{r_i,n_j}$;
7 **foreach** $n_j \in \mathcal{N}$ **do**
8 **for** $k = 1 \rightarrow \omega$ **do**
9 $l_{n_j,T,k} \leftarrow MakeEdge(n_j, T, 2k-1), \widetilde{L} \leftarrow \widetilde{L} \cup l_{n_j,T,k}$;
10 $\widetilde{E} \leftarrow \widetilde{Q} \cup \widetilde{B} \cup \widetilde{L}, \widetilde{G} \leftarrow (\widetilde{V}, \widetilde{E}), flow \leftarrow CostFlow(\widetilde{G})$;
11 **foreach** $b_{r_i,n_j} \in \widetilde{B}$ **do**
12 **if** $flow[b_{r_i,n_j}] = 1$ **then**
13 $p_i \leftarrow n_j, \mathcal{H} \leftarrow \mathcal{H} \cup p_i$;
14 **return** $\mathcal{H}$;

For instance, Figure 6 is the flow network of nodes n_{5-8} and shards r_{5-8} in Figure 1. The max-flow is 4 since each shard is designated 1 leader. The min-cost should be 4, as any unbalanced distribution that includes a node that possesses more than 1 leader results in a higher cost. Therefore, by constructing a min-cost solution, Algorithm 2 generates the most balanced leader distribution.

4.3 Optimality Analysis

In this section, Lemma 5 guarantees that Algorithm 2 always generates valid solutions, and Proposition 5 demonstrates that these solutions lead to the optimally balanced leader distribution.

Lemma 5. *The leader distribution generated by Algorithm 2 guarantees that each shard will be designated exactly one leader.*

Proof. Please see the proof in Section A.10 [1]. □

Because the cost of edges $l_{n_j,T,k} \in \widetilde{L}$ is an additive convex function, any valid solution generated by Algorithm 2 tends to be a balanced leader distribution, as demonstrated by Proposition 5.

Proposition 5. *Let $\mathcal{N}$ denote the nodes set, $\mathcal{R}$ denote the shards set placed by Algorithm 1. Algorithm 2 generates the optimal leader distribution since $\forall n_i, n_j \in \mathcal{N}, |\eta_i - \eta_j| \le 1 \wedge \forall r_k \in \mathcal{R}, p_k \in r_k$.*

Proof. Please see the proof in Section A.11 [1]. □

As demonstrated by Proposition 5, our CFS algorithm can generate the optimal balanced leader distribution based on the placement schemes produced by the PGP algorithm, distinguishing CFS from other algorithms. In this context, the GREEDY and RANDOM algorithms remain trapped in suboptimal solutions, as discussed at the beginning of Section 4, while dynamic balancing algorithms such as LOGSTORE [5] and ESDB [48] do not offer this feature.

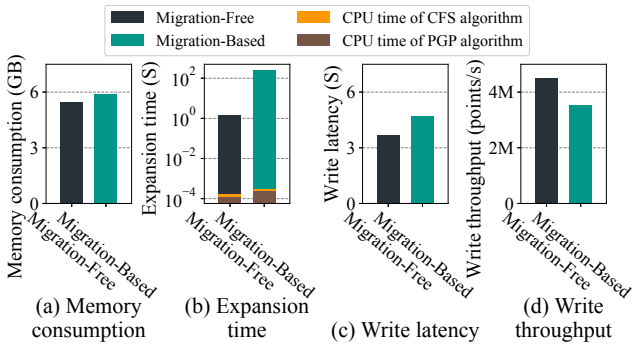

Figure 7: Comparison of write efficiency between migration-free and migration-based mechanisms in the expansion scenario. The runtime of our algorithms is negligible, and the migration-free mechanism achieves better performance.

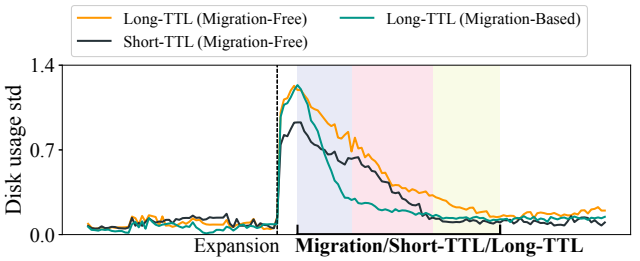

Figure 8: Comparison of disk usage std across different lengths of TTL settings in the expansion scenario.

5 EVALUATION

Section 5.1 presents the experimental setup, and Section 5.2 compares a migration-based mechanism. We evaluate the replica placement algorithms in Section 5.3 and compare the leader selection algorithms in Section 5.4. The evaluation codes are available at [2].

5.1 Experimental Setup

5.1.1 Hardware. All experiments are performed on an IoTDB cluster consisting of 21 ECS virtual machines on Tencent Cloud, where each machine has 4 CPU cores, 8GB of RAM, a 100 GB SSD, and a 1.5 Gbps ethernet. The cluster includes 1 primary node to deploy the algorithms, 4 write clients, and 16 nodes for data storage.

5.1.2 Dataset. The Write-Ahead Logging (WAL) files provided by AUTOAI are used as our dataset. The dataset includes sampling records from approximately 20 million sensors, each mapped to a unique series. Furthermore, since the WAL files preserve the original sequence of data arrival, the write clients we implemented more accurately reproduce the load of the production environment.

5.1.3 Evaluation Scenario. We design two evaluation scenarios to correspond to the IoT challenges mentioned in Section 1: (i) Expansion scenario: deploying an 8-node cluster initially and then expanding it to 16 nodes. After the cluster expansion, we double the write load and switch the time partition of write requests. (ii) Disaster scenario: deploying a 16-node cluster, shutting down a node after running for a while, and restarting it after a fixed time.

5.1.4 Key Metrics. Disk usage std to reflect the storage balance and fault tolerance of our PGP (Algorithm 1). Write throughput std to present the write load balance of our CFS (Algorithm 2).

5.2 Migration Evaluation

We monitor the memory and CPU usage after expansion. As shown in Figure 7(a), the memory usage of the migration-based approach is similar to our migration-free proposal. The expansion time, as depicted in Figure 7(b), differs significantly. Due to the cumbersome data migration, the expansion time of the migration-based method is 200× slower than that of the migration-free mechanism. Additionally, the overhead of both our PGP placement and CFS selection algorithms is negligible among the total expansion time, less than 1/1000. Next, we compare the latency and throughput to leverage the performance gains. Compared to the migration-based mechanism, the migration-free mechanism achieves 21.6% lower write latency and 27.5% higher write throughput. The results demonstrate that the trade-off of our algorithms is worthwhile, and clarify their practicality in time-sensitive applications.

Figure 8 presents a further analysis on how TTL affect storage balance. In addition to the normal short-TTL (10-minute), we consider another workload pattern of long-TTL (15-minute). As shown, the unusual data inflow/outflow pattern leads to higher temporary imbalance after expansion, owing to cold data accumulation. Consequently, the migration-free approach takes a longer time to rebalance storage. In this case, the alternative mechanism of migration could be triggered. It eliminates imbalance quickly with extra costs as analyzed in Figure 7. This adjustment enables best practices for deploying our strategy in different IoT settings.

5.3 Replica Placement Evaluation

In this section, we compare our PGP with baseline replica placement algorithms: (i) WRR: places replicas to each node in turn. (ii) COPYSET [9]: generates random permutations and splits them into multiple placement schemes. (iii) TIERED [8]: enumerates every possible placement scheme and returns the one with the fewest overlaps with employed schemes. We further compare PGP with two advanced fault-tolerant replica placement algorithms: (i) GEMINI [45], an advanced multi-node fault-tolerant (Section 3.4.2) algorithm that divides nodes into groups and places replicas in turn, and (ii) HYDRA [24], an advanced variant of the COPYSET algorithm that randomly generates Copysets as placement schemes.

5.3.1 Expansion Scenario. As shown in Figure 9, fault-tolerant algorithms, including COPYSET, TIERED, GEMINI, and HYDRA, perform poorly after cluster expansion due to their insufficient consideration of this context. Consequently, they fail to maintain storage balance, as depicted in Figure 10. The detailed disk usage distributions generated by different placement algorithms are illustrated in Figure 19 in the full technical report [1].

Figure 10 shows the changes in disk usage std over time, where Weighted-Round-Robin (WRR) and our PGP achieve the best storage balance by minimizing the std. On the contrary, fault-tolerant algorithms, including COPYSET, TIERED, HYDRA and GEMINI,

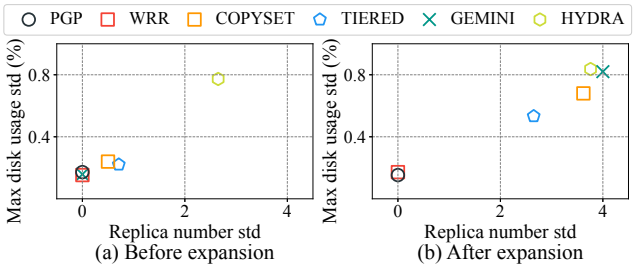

Figure 9: Replica distribution in expansion scenario.

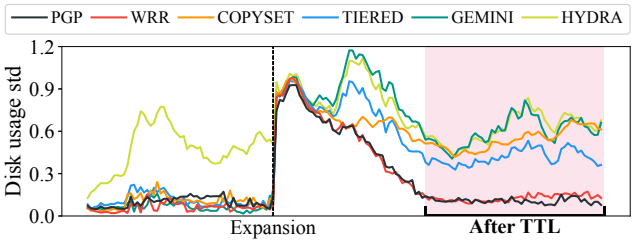

Figure 10: Disk usage std in expansion scenario.

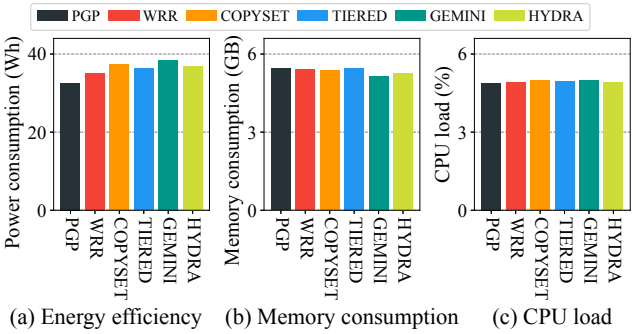

Figure 11: Cluster energy efficiency and resource usage produced by different placement algorithms.

perform poorly after expansion due to their insufficient consideration of this context. As a result, in Figure 10, the maximum std after TTL eliminates expired data (represented by the pink range) of our PGP is the lowest, while the std produced by the state-of-the-art fault-tolerant algorithms is at least 248.4% higher. A detailed comparison is provided in Table 3 of our full technical report [1].

With a high frequency, TTL is triggered to eliminate expired data. As shown in Figure 10, the disk usage std generated by our PGP algorithm decreases in a near-linear manner.

Given the same amount of data to write, Figure 11 monitors the electricity and memory, CPU usage of IoTDB. Figure 11(a) shows the electricity consumed for the same number of write requests by different placement algorithms. As shown, the electricity overhead of our PGP is at least 7.4% lower than that of the other alternatives. Meanwhile, the average memory and CPU overhead across all 16 storage nodes for different placement algorithms remain similar, as

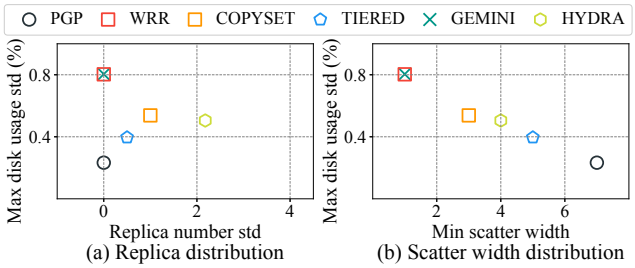

Figure 12: Replica distribution in disaster scenario.

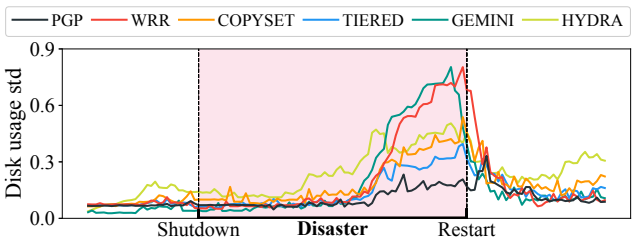

Figure 13: Disk usage std in disaster scenario.

shown in Figures 11(b) and 11(c). This is because the same amount of workload is processed under the same hardware configurations, and our PGP completes all write tasks more efficiently.

5.3.2 Disaster Scenario. As in Figure 12, WRR and GEMINI ensure an even replica distribution with minimal scatter width. The COPYSET, TIERED, and HYDRA obtain higher scatter width by an unbalanced replica distribution. In Figure 13, the disk usage std for GEMINI closely resembles that of WRR, while COPYSET, TIERED, and HYDRA exhibit similar results. The detailed variation trend of disk usage during this evaluation is illustrated in Figure 20 in [1].

Figure 13 illustrates the changes in disk usage std during the disaster scenario. Similar to WRR, the GEMINI for multi-node fault-tolerant only (Section 6.2) causes the disk usage of another node to proliferate when a node fails. Meanwhile, COPYSET and its advanced variants, including TIERED and HYDRA, retain some degree of tolerance. In Figure 13, the maximum std during disaster (represented by the pink range) of our PGP is the lowest, while the std produced by the state-of-the-art fault-tolerant algorithms is at least 70.0% higher. We present this std in Table 3 [1].

5.4 Leader Selection Evaluation

In this section, we compare our CFS with baseline selection algorithms: (i) GREEDY: selects as leader the replica on the node that owns the fewest leaders. (ii) RANDOM: selects a random leader. To better showcase the ability of our CFS algorithm in achieving write load balance, we also compare with more recent and advanced baseline algorithms, in addition to the GREEDY and RANDOM strategies. Two recent dynamic leader selection algorithms are considered: (i) LOGSTORE [5], which periodically invokes the max-flow algorithm [17], and (ii) ESDB [48], which inserts replicas into a hash ring and periodically selects leaders based on hashing time.

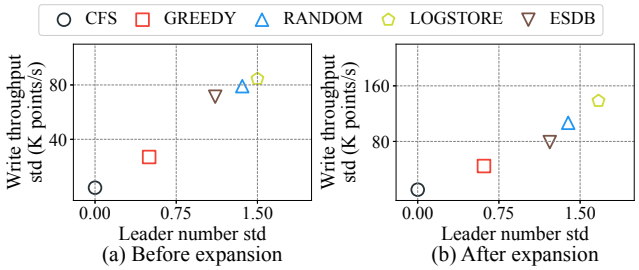

Figure 14: Leader distribution in expansion scenario.

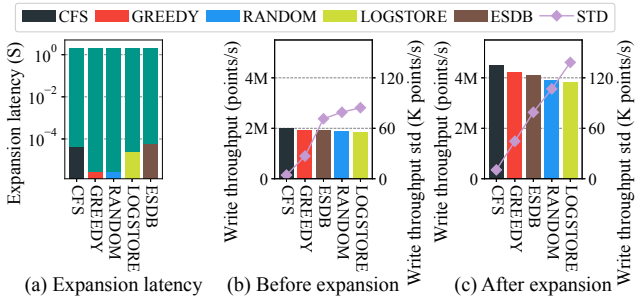

Figure 15: Cluster write throughput in expansion scenario.

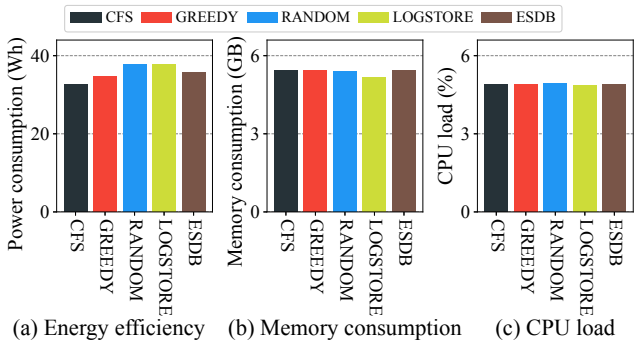

Figure 16: Cluster energy efficiency and resource usage produced by different selection algorithms.

5.4.1 Expansion Scenario. Figure 14 illustrates the leader distribution. Both before and after expansion, the dynamic algorithms LOGSTORE and ESDB fall behind CFS, resulting in a higher average write throughput std and, consequently, lower cluster write throughput, as shown in Figures 15(b) and 15(c). In Figure 15(a), although the expansion latency of our CFS is slightly higher, the trade-off is justified by the improved write performance.

Figure 15 shows the write performance in the expansion scenario. Benefiting from the most balanced leader distribution (Figure 14), our CFS achieves the best write throughput both before and after expansion. The LOGSTORE and ESDB do not perform as well, as their primary objective is to prevent any node from overloading, rather than generating a balanced leader distribution. The average write throughput std after expansion produced by the alternatives is

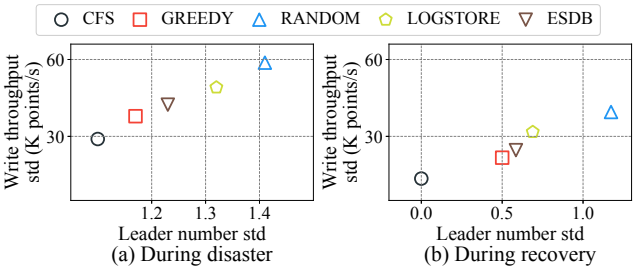

Figure 17: Leader distribution in disaster scenario.

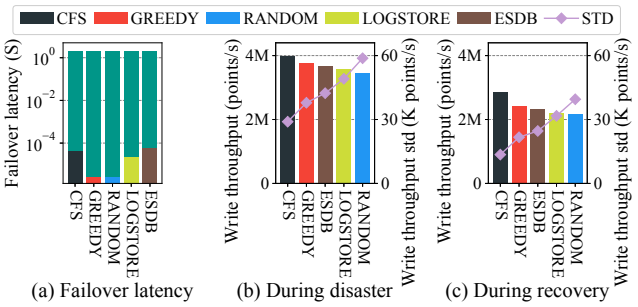

Figure 18: CFS improves cluster write throughput without compromising failover latency in the disaster scenario.

at least 322.9% higher than that of CFS, while the cluster throughput of CFS is at least 6.9% greater. A detailed record of the average write throughput std is provided in Table 4 [1].

As depicted in Figure 16(a), the electricity consumption of our CFS is at least 6.5% lower than that of other baseline algorithms under the same hardware and workload configurations in Figures 16(b) and (c). This improvement is due to our CFS completing all write tasks more efficiently, delivering the best write throughput, which is enabled by the most balanced leader distribution.

5.4.2 Disaster Scenario. As shown in Figure 17, the leader distribution of the dynamic balancing algorithms LOGSTORE and HYDRA is worse than that of CFS. This leads to inferior throughput performance during both the disaster and recovery phases, in Figures 18(b) and 18(c). Analogously, our CFS produces slightly higher failover latency, as seen in Figure 18(a), but this is still justified by the improved cluster write throughput performance.

Figure 18 shows the write performance in the disaster scenario. Similar to Section 5.4.1, the most balanced leader distribution (Figure 17) achieves the highest write throughput for CFS during disaster and recovery phases. The average write throughput std during recovery produced by the alternatives is at least 60.3% higher than that of CFS, while CFS's cluster throughput is at least 18.4% greater. Detailed average write throughput std are listed in Table 5 [1]. In Figure 18(a), although our CFS is relatively time-consuming, it accounts for only $1/10^4$ of the failover latency. Meanwhile, CFS improves cluster write throughput by at least 5.1% during disaster (Figure 18(b)) and at least 18.4% during recovery (Figure 18(c)).

6 RELATED WORK

In this section, we first introduce existing data partitioning and allocation architectures in Section 6.1. Subsequently, in Section 6.2, we elaborate on research directions similar to our fault tolerance modeling, as well as replica placement algorithms from related communities. Finally, Section 6.3 demonstrates the potential of our selection algorithm for serving different types of consensus protocols and compares our approach with other dynamic alternatives.

6.1 Decentralized Partitioning and Allocation

The Distributed Hash Table (DHT) architecture, exemplified by Chord[41], CAN [37], Pastry [38], and Tapestry [50], offers a balanced, decentralized, and scalable approach to data partitioning and allocation and has been deployed in many distributed systems, such as ZHT [25] and DART [26]. However, we chose not to adopt the DHT architecture primarily due to its reliance on a dynamic hash map, which is difficult to leverage TTL. After cluster expansion, the unexpired data will be migrated to new nodes.

6.2 Replica Placement

As introduced by the paper of COPYSET [9], employing fewer number of Copysets improves the cluster's multi-node fault tolerance but reduces the scatter width of each node, thereby compromising single-node fault tolerance. Two directions in replica placement research follow this trade-off: (i) The algorithm for multi-node failures only, such as GEMINI [45], which minimize the number of employed Copysets; and (ii) COPYSET and its advanced variants, including TIERED [8] and HYDRA [24], which aim to achieve adequate scatter width with as few Copysets as possible. However, these approaches do not account for storage balance in an ever-expanding cluster, making them unsuitable for our scenario. The symmetric counterpart of (ii), maximizing scatter width given a fixed number of Copysets, is more appropriate for our work. Unfortunately, it presents greater challenges, as we have proven in Theorem 1. While the Balanced Incomplete Block Design (BIBD) [13] can also model our replica placement challenge, it only constructs optimal schemes under several restrictions [39]. Thus, research leveraging BIBD [16] can only be applied to a limited set of fixed cluster configurations, restricting the possible combinations of the number of nodes n, replication factor ρ, and load factor ω.

Unlike storage balance considered in our distributed database scenarios, the distributed data stream processing systems need to study the placement of inter-correlated operators. (i) The heuristics methods proposed by Nardelli et al. [30] only serve workloads with the replication factor $\rho = 1$. However, in our context, a replication factor of $\rho \geq 2$ is always required to ensure fault tolerance. (ii) The EDRP presented by Cardellini et al. [6] considers the data transition costs between stream processing operators as part of its optimization objective. We do not leverage this feature, since different shards do not exchange data when handling write requests. (iii) NEMO by Chatziliadis et al. [7] places replicas to reduce the latency introduced by geo-distributed clusters. However, the current version of Apache IoTDB does not yet optimize for the communication latency between geo-distributed nodes. Nevertheless, NEMO's practical geographical modeling approach offers a promising avenue for further refinement of our system in future work.

6.3 Leader Selection

It is true that leader selection protocols [31, 40] tend to be indeterminate in who would eventually be elected, but can be configured to "prefer" certain leaders without compromising integrity of the protocol. For instance, Ratis [36], an open-source implementation of Raft [31], supports relatively deterministic leader elections by assigning different priorities to replicas. IoTDB employs Ratis for strong consistency, i.e., the leader distribution in Ratis within our system is deliberately directed by giving higher weight to the selected replica. On the other hand, controlling the leader distribution of eventually consistent protocols is comparatively easier, as some of these protocols, such as CURP [34], do not define any leader election principle. Thereby, in the eventual consistency implementation of Apache IoTDB, our CFS algorithm is also applied. In this sense, the "directed" approach of our CFS algorithm can be applied to both eventual and strong consistency.

Dynamic selection algorithms are commonly used to address unbalanced load distribution, such as the LOGSTORE [5] and ESDB [48] for the tenant workload. Since it is difficult to predict whether a tenant and its corresponding replicas will become a hotspot, the primary objective of these algorithms is to prevent the load on each node from exceeding a predefined threshold. Fortunately, the characteristic of hot data partitions shifting over time in our context, as introduced in Section 1.2.3, makes a globally balanced leader selection algorithm both feasible and practical.

7 CONCLUSION

This study is inspired by the TTL feature of time series data, the need to gradually expand cluster nodes in response to the number of growing sensors, and the intensive write load in IoT scenarios. We propose data partitioning and allocation strategies that avoid data migration by utilizing TTL. Our evaluation demonstrates that these strategies automatically obtain eventual storage balance after TTL expiration. To approximate the NP-complete replica placement problem, we design the PGP placement algorithm through a systematic division of cluster nodes, and provide rigorous proofs to its fault tolerance. The PGP algorithm achieves the most balanced storage load distribution in our evaluations, which improves the cluster's availability, thus making it highly suitable for dynamically expanding series and clusters in IoT scenarios. To handle intensive write loads in IoT scenarios, we propose the CFS leader selection algorithm, which consistently provides an optimally balanced solution. Its benefits for cluster performance are presented in our evaluations. Our proposal has become the replica placement and leader selection solution of Apache IoTDB since version 1.3.

ACKNOWLEDGMENTS

This work is supported in part by the Chongqing Technology Innovation and Application Development Project (CSTB2023TIAD-STX0034), the National Natural Science Foundation of China (92267203, 62021002, 62072265, 62232005), the National Key Research and Development Plan (2021YFB3300500), and Beijing Key Laboratory of Industrial Big Data System and Application. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

[1] 2025. *Appendix.* (2025, April). [Online]. Available: https://crzbulabula.github.io/migration-free_appendix.pdf.

[2] 2025. *Evaluation codes.* (2025, April). [Online]. Available: https://github.com/CRZbulabula/iotdb/tree/migration-free-elastic-storage.

[3] 2025. *IoTDB implementations.* (2025, April). [Online]. Available: https://iotdb.apache.org/UserGuide/V1.3.x/Technical-Insider/Cluster-data-partitioning.html.

[4] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere platform for big data analytics. *VLDB J.* 23, 6 (2014), 939–964. https://doi.org/10.1007/S00778-014-0357-Y

[5] Wei Cao, Xiaojie Feng, Boyuan Liang, Tianyu Zhang, Yusong Gao, Yunyang Zhang, and Feifei Li. 2021. LogStore: A Cloud-Native and Multi-Tenant Log Database. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2464–2476. https://doi.org/10.1145/3448016.3457565

[6] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Optimal operator deployment and replication for elastic distributed data stream processing. *Concurr. Comput. Pract. Exp.* 30, 9 (2018). https://doi.org/10.1002/CPE.4334

[7] Xenofon Chatziliadis, Eleni Tzirita Zacharatou, Alphan Eracar, Steffen Zeuch, and Volker Markl. 2024. Efficient Placement of Decomposable Aggregation Functions for Stream Processing over Large Geo-Distributed Topologies. *Proc. VLDB Endow.* 17, 6 (2024), 1501–1514. https://www.vldb.org/pvldb/vol17/p1501-chatziliadis.pdf

[8] Asaf Cidon, Robert Escriva, Sachin Katti, Mendel Rosenblum, and Emin Gün Sirer. 2015. Tiered Replication: A Cost-effective Alternative to Full Cluster Geo-replication. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, Shan Lu and Erik Riedel (Eds.). USENIX Association, 31–43. https://www.usenix.org/conference/atc15/technical-session/presentation/cidon

[9] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John K. Ousterhout, and Mendel Rosenblum. 2013. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, Andrew Birrell and Emin Gün Sirer (Eds.). USENIX Association, 37–48. www.usenix.org/conference/atc13/technical-sessions/presentation/cidon

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 8. https://doi.org/10.1145/2491245

[11] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert van Renesse. 2020. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 325–338. https://www.usenix.org/conference/nsdi20/presentation/ding

[12] Efim A Dinic. 1970. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, Vol. 11. 1277–1280.

[13] Ronald Aylmer Fisher et al. 1940. 174: An Examination of the Different Possible Solutions of a Problem in IncompleteBlocks. (1940).

[14] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in Globally Distributed Storage Systems. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). USENIX Association, 61–74. http://www.usenix.org/events/osdi10/tech/full_papers/Ford.pdf

[15] Lester Randolph Ford and Delbert R Fulkerson. 1957. A simple algorithm for finding maximal network flows and an application to the Hitchcock problem. *Canadian journal of Mathematics* 9 (1957), 210–218.

[16] Z. Gao, S. Lin, and N. Yu. 2020. Deterministic Schemes of Copyset Replication. In *2020 International Conference on Computer Engineering and Application (ICCEA)*. IEEE Computer Society, Los Alamitos, CA, USA, 622–626. https://doi.org/10.1109/ICCEA50009.2020.00136

[17] Andrew V. Goldberg and Robert Endre Tarjan. 1988. A new approach to the maximum-flow problem. *J. ACM* 35, 4 (1988), 921–940. https://doi.org/10.1145/48014.61051

[18] Fusheng Han, Hao Liu, Bin Chen, Debin Jia, Jianfeng Zhou, Xuwang Teng, Chuanhui Yang, Huafeng Xi, Wei Tian, Shuning Tao, Sen Wang, Quanqing Xu, and Zhenkun Yang. 2024. PALF: Replicated Write-ahead Logging for Distributed Databases. *Proc. VLDB Endow.* 17, 12 (2024), 3745–3758. https://www.vldb.org/pvldb/vol17/p3745-xu.pdf

[19] HBase. 2025. *HBase.* https://hbase.apache.org/ (2025, April). [Online]. Available: https://hbase.apache.org/.

[20] Robert J. Chansler Jr. 2012. Data Availability and Durability with the Hadoop Distributed File System. *login Usenix Mag.* 37, 1 (2012). https://www.usenix.org/publications/login/february-2012/data-availability-and-durability-hadoop-distributed-file-system.

[21] Yuyuan Kang, Xiangdong Huang, Shaoxu Song, Lingzhe Zhang, Jialin Qiao, Chen Wang, Jianmin Wang, and Julian Feinauer. 2022. Separation or Not: On Handing Out-of-Order Time-Series Data in Leveled LSM-Tree. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 3340–3352. https://doi.org/10.1109/ICDE53745.2022.00315

[22] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. 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 the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, Frank Thomson Leighton and Peter W. Shor (Eds.). ACM, 654–663. https://doi.org/10.1145/258533.258660

[23] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.* 44, 2 (2010), 35–40. https://doi.org/10.1145/1773912.1773922

[24] Youngmoon Lee, Hasan Al Maruf, Mosharaf Chowdhury, Asaf Cidon, and Kang G. Shin. 2022. Hydra : Resilient and Highly Available Remote Memory. In *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*, Dean Hildebrand and Donald E. Porter (Eds.). USENIX Association, 181–198. https://www.usenix.org/conference/fast22/presentation/lee

[25] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, Dongfang Zhao, Ke Wang, Anupam Rajendran, Zhao Zhang, and Ioan Raicu. 2013. ZHT: A Light-Weight Reliable Persistent Dynamic Scalable Zero-Hop Distributed Hash Table. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*. IEEE Computer Society, 775–787. https://doi.org/10.1109/IPDPS.2013.110

[26] Pinchao Liu, Dilma Da Silva, and Liting Hu. 2021. DART: A Scalable and Adaptive Edge Stream Processing Engine. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 239–252. https://www.usenix.org/conference/atc21/presentation/liu

[27] Edward F Moore. 1959. The shortest path through a maze. In *Proc. of the International Symposium on the Theory of Switching*. Harvard University Press, 285–292.

[28] MQTT. 2025. *MQTT.* https://mqtt.org/ (2025, April). [Online]. Available: https://mqtt.org/.

[29] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. 2017. Time series databases and influxdb. *Studienarbeit, Université Libre de Bruxelles* 12 (2017), 1–44.

[30] Matteo Nardelli, Valeria Cardellini, Vincenzo Grassi, and Francesco Lo Presti. 2019. Efficient Operator Placement for Distributed Data Stream Processing Applications. *IEEE Trans. Parallel Distributed Syst.* 30, 8 (2019), 1753–1767. https://doi.org/10.1109/TPDS.2019.2896115

[31] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, Garth Gibson and Nickolai Zeldovich (Eds.). USENIX Association, 305–319. www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[32] OpenTSDB. 2025. *Hbase in OpenTSDB.* http://opentsdb.net/ (2025, April). [Online]. Available: http://opentsdb.net/docs/build/html/user_guide/backends/hbase.html.

[33] OpenTSDB. 2025. *OpenTSDB.* http://opentsdb.net/ (2025, April). [Online]. Available: http://opentsdb.net/.

[34] Seo Jin Park and John K. Ousterhout. 2019. Exploiting Commutativity For Practical Fast Replication. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 47–64. https://www.usenix.org/conference/nsdi19/presentation/park

[35] Patroni. 2025. *Patroni.* https://github.com/zalando/patroni (2025, April). [Online]. Available: https://github.com/zalando/patroni.

[36] Apache Ratis. 2025. *Ratis.* https://github.com/apache/ratis (2025, April). [Online]. Available: https://github.com/apache/ratis.

[37] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. 2001. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA*, Rene L. Cruz and George Varghese (Eds.). ACM, 161–172. https://doi.org/10.1145/383059.383072

[38] Antony I. T. Rowstron and Peter Druschel. 2001. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In

Middleware 2001, IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, Germany, November 12-16, 2001, Proceedings (Lecture Notes in Computer Science), Rachid Guerraoui (Ed.), Vol. 2218. Springer, 329–350. https://doi.org/10.1007/3-540-45518-3_18

[39] Kenneth FN Scott. 1973. On the Construction of Bibd With λ= 1. *Canad. Math. Bull.* 16, 3 (1973), 329–335.

[40] Buti Sello, Jianming Yong, and Xiaohui Tao. 2024. Erdos: A Novel Blockchain Consensus Algorithm with Equitable Node Selection and Deterministic Block Finalization. *Data Sci. Eng.* 9, 4 (2024), 361–377. https://doi.org/10.1007/S41019-024-00251-0

[41] Ion Stoica, Robert Tappan Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.* 11, 1 (2003), 17–32. https://doi.org/10.1109/TNET.2002.808407

[42] TimescaleDB. 2025. *TimescaleDB.* https://www.timescale.com/ (2025, April). [Online]. Available: https://www.timescale.com/.

[43] Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, Eric A. Brewer and Peter Chen (Eds.). USENIX Association, 91–104. http://www.usenix.org/events/osdi04/tech/renesse.html

[44] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. 2023. Apache IoTDB: A Time Series Database for IoT Applications. *Proc. ACM Manag. Data* 1, 2 (2023), 195:1–195:27. https://doi.org/10.1145/3589775

[45] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (Eds.). ACM, 364–381. https://doi.org/10.1145/3600006.3613145

[46] Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jianmin Wang. 2022. Time Series Data Encoding for Efficient Storage: A Comparative Analysis in Apache IoTDB. *Proc. VLDB Endow.* 15, 10 (2022), 2148–2160. https://doi.org/10.14778/3547305.3547319

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[48] Jiachi Zhang, Shi Cheng, Zhihui Xue, Jianjun Deng, Cuiyun Fu, Wenchao Zhou, Sheng Wang, Changcheng Chen, and Feifei Li. 2022. ESDB: Processing Extremely Skewed Workloads in Real-time. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2286–2298. https://doi.org/10.1145/3514221.3526051

[49] Zhou Zhang, Peiquan Jin, Xike Xie, Xiao-Liang Wang, Ruicheng Liu, and Shouhong Wan. 2024. Online Nonstop Task Management for Storm-Based Distributed Stream Processing Engines. *J. Comput. Sci. Technol.* 39, 2 (2024), 116–138. https://doi.org/10.1007/S11390-021-1629-9

[50] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. 2004. Tapestry: a resilient global-scale overlay for service deployment. *IEEE J. Sel. Areas Commun.* 22, 1 (2004), 41–53. https://doi.org/10.1109/JSAC.2003.818784