Elastic Storage of Time Series Data in Apache IoTDB

[Industry]

Yongzao Dan
Tsinghua University
crz22@mails.tsinghua.edu.cn

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

Xiangdong Huang*
Timecho Ltd
hxd@timecho.com

Chen Wang
Timecho Ltd
wangchen@timecho.com

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

Jianmin Wang
Tsinghua University
jimwang@tsinghua.edu.cn

ABSTRACT

The time series data model commonly used in IoT scenarios usually partitions the data in two dimensions: time series and time. And the historical data is periodically cleared by setting Time to Live (TTL). The time series database cluster is gradually expanded as the time series required by the business increases. Due to the uninterrupted work of the sampling time series, the IoT scenario has a large intensive write load. The relational data model rarely considers the time partition, which makes it difficult to carry out the automatic data deletion mechanism based on TTL in contrast. As the cluster starts from scratch, the replica placement algorithm to maintain load balance and disaster recovery capability during the gradual growth process is barely discussed in previous research. The large amount of intensive write load in the IoT scenario makes the selection of the primary replica a significant influence on the write load balance of the cluster. Our solution is to come up with a data allocation table that can automatically balance the storage load using TTL. Using heuristic search in replica placement to provide load balance and disaster recovery capability for the cluster. The cost flow model is established to balance the write load of the cluster. Our algorithms has been implemented and deployed in Apache IoTDB [27], an open source time series database system. Our algorithm achieves the best computing load balance and disaster recovery capability in the evaluation and ensures that the cluster can realize storage load balance after TTL.

PVLDB Reference Format:
Yongzao Dan, Xiangpeng Hu, Xiangdong Huang, Chen Wang, Shaoxu Song, and Jianmin Wang. Elastic Storage of Time Series Data in Apache IoTDB. PVLDB, 17(X): XXX-XXX, 2024.
doi:XX.XX/XXX.XX

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

*Xiangdong Huang is the PMC Chair of Apache IoTDB Committee (https://iotdb.apache.org/).
†Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. X ISSN 2150-8097.
doi:XX.XX/XXX.XX

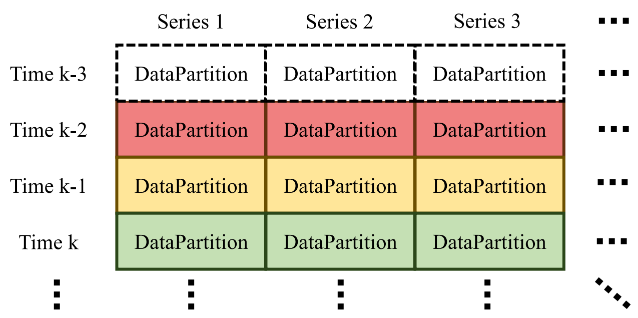


Figure 1: Data partition model in IoT scenarios

1 INTRODUCTION

what prob-
lem is stud
ied in this
paper

The Internet of Things (IoT) [21] scenario has brought new business opportunities and nurtured many new enterprises [24]. The high write load and data growth in IoT scenarios make it necessary for enterprises to deploy distributed databases. Most distributed database systems are equipped with data partition and replica placement algorithms. The data written by the user is first divided into reasonable data partitions by the data partition algorithm and then handed over to a specific replica group for unified management. The node to which each replica in the replica group belongs is determined by the replica placement algorithm. Proper replica placement can improve the load balance and disaster recovery capability of the cluster. Based on the cluster replica placement, selecting the appropriate primary replica can effectively balance the write load. In this paper, we propose a set of data partition, replica placement, and primary replica selection algorithms suitable for IoT scenarios.

1.1 Motivation

use examp
from IoTD
customers.

1.1.1 Time to Live. The growth of data in IoT scenarios is explosive. For instance, the vehicle companies we serve have dozens of sampling time series for each commercial vehicle, and millions of commercial cars have been sold, generating terabytes of new data every day. Queries and analytics in IoT scenarios tend to focus on recent data, making historical data less valuable than recent data. Time to Live (TTL) is a feature contained in many time series databases, such as InfluxDB [16]. The time series data would be automatically deleted by the database system through user setting TTL so that the storage pressure on the cluster can be reduced. Assuming that the TTL configured by the user is equal to 3, as presented in Figure 1, the data written at time $k-3$ should be automatically deleted when time k arrives. The data written at time $k-2$ will be the next batch to be deleted due to TTL expiration.

1.1.2 Gradual Data Expansion. Planning cluster size according to the number of existing time series is an effective method to control storage cost. For example, with the development of the new energy industry, wind power enterprises and photovoltaic enterprises will introduce more power generation equipment, bringing about a gradually increasing time series. Therefore, many IoT enterprises will have the requirement to gradually expand their cluster. As shown in Figure 1, an increase in the time series increases the amount of data in the same period, which requires the cluster to have more nodes to manage this additional load.

1.1.3 Intensive Write Workload. Time series continuously produces time series data with the latest time. For instance, the data collected by the meteorological observation station will be marked with the current time, the typhoon and rainstorm that have ended will no longer produce meteorological data. This makes the time series data being written to the time series database almost carry the latest time. As shown in Figure 1, the time series database receives almost no time series data from time $k-1$ or earlier when processing time data written at time k.

1.2 Challenge

y existing
utions do
t work for
e above
T work-
d

1.2.1 Partition Allocation. In the relational database architecture, the importance of data is often related to its key, which means not only the recent but also the historical data have the potential to become hotspots. As a result, it is generally believed that it's necessary to process data migration for transferring load after cluster expansion. Data partitions in Cassandra [11] are always automatically migrated with the distribution of new replica groups (virtual nodes) introduced by new nodes as designed by the consistent hash algorithm [10]. In the HDFS [23], the balancer process in NameNode automatically evaluates the load of each DataNode in the background and initiates migration to keep the load between DataNodes within a reasonable range. This architecture is also adopted by OpenTSDB [18], which uses HBase [8] to manage the storage of time series data. However, data migration is always accompanied by additional network traffic costs in the cluster. And then in turn reduces the quality of the cluster's write service, which is not desired in IoT scenarios with high write load.

y exist-
g meth-
s such as
pyset does
t work,
e to the
adual data
pansion,
d TTL

1.2.2 Replica Placement. Both load balance and disaster recovery capability are related to the cluster's replica replacement algorithm. The Copyset [4] paper modeled the probability of cluster data unavailability based on combinatorial mathematics and probability theory, proposed an algorithm with both load balance and disaster recovery capability and carried out implementation and analysis in HDFS cluster. However, the Copyset algorithm requires the number of nodes in the cluster to be relatively fixed. The author team subsequently proposed the Tiered Replication [3] algorithm, which is more friendly to cluster expansion. But both Copyset and Tiered Replication algorithms are randomness, ultimately their load balance is guaranteed only when the cluster has enough nodes. As shown in Figure 2, this replica placement scheme improves the disaster recovery capability of the cluster. For example, when Node 1

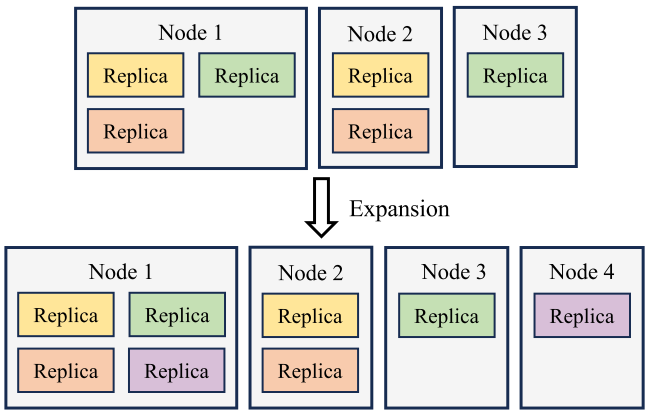


Figure 2: Random gives better disaster recovery capability but results in unbalanced load distribution

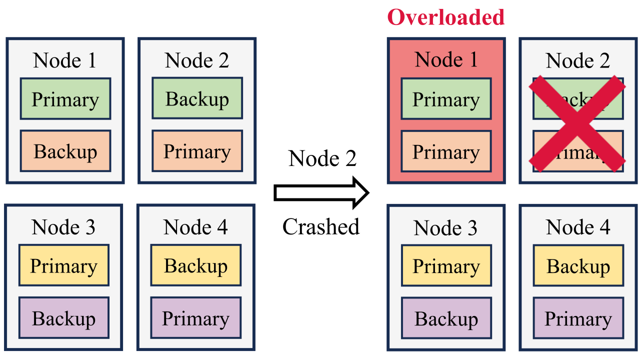


Figure 3: Intuitive greedy results in fixed pair and thus poor in disaster recovery capability

shutdown, its load can be distributed to Node 2 and Node 3. The randomized replica placement algorithm makes it easy for each node to hold a different number of replicas, which makes the load difference of each node in the newly established cluster obvious. This problem cannot be solved by gradual expansion, because both existing and new nodes have the probability of obtaining new replicas. For example, when Node 4 is expanded and the cluster thus needs to add a new replica group, both Nodes 1, 2, and 3 have the probability of obtaining a replica from the new replica group.

To achieve load balance feature, an intuitive solution is to ensure that all nodes in the cluster hold the same number of data replicas. However, the disaster recovery capability of the cluster is then compromised. A greedily Round-Robin-like algorithm is sufficient to satisfy the load balance requirement but greatly reduces the disaster recovery capability in the meantime, as shown in Figure 3. The Round-Robin-like algorithm will inevitably form a fixed node pairing. One node in this pair is likely to be overloaded as long as the other node shutdown unexpectedly due to all load from the shutdown node has to be carried by the alive one.

1.2.3 Primary Replica Selection. Write requests for data partitions are generally handled by the primary replica of the corresponding

how exist-
ing strateg
such as Ra
fails, owing
to intensiv
writes

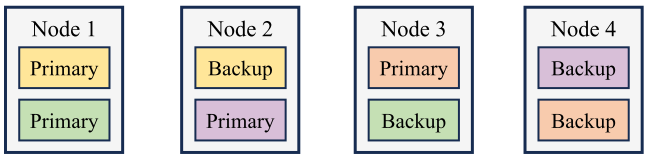


Figure 4: Uneven primary replica distribution

replica group. If the replica group elects its primary replica, such as Raft [17] and the implementation in TimescaleDB [25] through Patroni [20], may result in the primary replica distribution shown in Figure 4. Because the earlier the vote request is made, the more likely the replica is to be elected. This uneven primary replica distribution will make it difficult for the cluster to withstand the high intensive write load in an IoT scenario due to a single node of computing bottleneck. To balance the write load across the cluster, a straightforward approach is for each replica group, select the node with the fewest number of primary replicas as its primary replica. The following selection process conforms to this greedy method, still resulting in the distribution shown in Figure 4:

- **Step 1.** Let the primary replica of the purple replica group be at Node 2.
- **Step 2.** Let the primary replica of the pink replica group be at Node 3.
- **Step 3.** Let the primary replica of the green replica group be at Node 1.
- **Step 4.** Regardless of whether the primary replica of the yellow replica group is located on node 1 or node 2, the distribution of primary replicas is uneven.

1.3 Contribution

w the pro- sal ad- esses the ove issues, e by one

Our target is to develop a set of data partition, replica placement, and primary replica selection algorithms that are better suited to IoT scenarios. Our contribution is thus shown as follows:

(1) **Data partition algorithm:** Based on the TTL of time series data, we propose the data allocation table to ensure load balance while avoiding data migration as described in section 3.
(2) **Replica placement algorithm:** Based on the modeling method proposed in the Copyset paper, we propose the GCR replica placement algorithm that can accompany the growth process of enterprise clusters from scratch, and establish a disaster model and a recovery model for this algorithm as shown in section 4.
(3) **Primary replica selection algorithm:** Considering the binding relationship between primary replica and intensive write load, we design the CFD algorithm that can always find the best primary replica distribution, and presented in section 5.

vance in periments

(4) **Evaluaton:** Through the combination of the above algorithms, the tested IoTDB cluster is guaranteed to achieve storage balance when the expansion is finished and after TTL of time. The cluster has strong load balance and disaster recovery capability during the whole evaluation, as shown in section 6.

2 THE ARCHITECTURE OF IOTDB

d an erview agraph

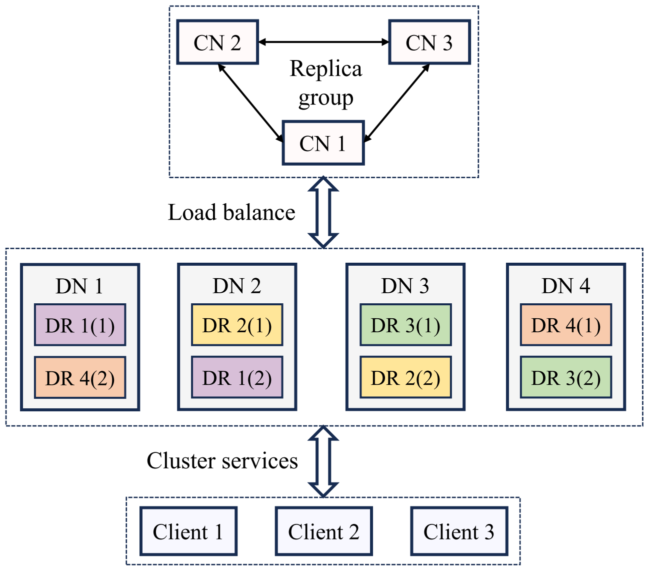


Figure 5: IoTDB cluster architecture

This section presents the cluster architecture and the partition process of IoTDB [9]. The main roles that make up the IoTDB cluster will be listed briefly. The whole process of data partition from generation to allocation is then shown.

2.1 IoTDB Cluster Architecture

add an overview paragraph

The IoTDB is a master-slave architecture inside the cluster. The ConfigNode guides data distribution among DataNodes. From an external perspective, it is a peer-to-peer architecture, where a client can connect to any DataNode to get services.

these are definitions not table

again and again, ask ruilei and jiangtian to advise in writing, don't spen my time o these writi issues

Definition 1 (cluster roles). *The IoTDB cluster consists of the following roles:*

- **ConfigNode (CN):** Manage cluster partition tables and load balance scheduling.
- **DataNoe (DN):** Store time series data and provide read/write services to clients.
- **DataRegion (DR):** Manage a portion of the cluster's data partitions.
- **DataRegionGroup (DRG):** Replica group composed by DataRegions.

As presented in Figure 5, the IoTDB cluster consists of ConfigNode and DataNode. The ConfigNode is the manager of the cluster schedule. All ConfigNodes in the cluster form a replica group, and the primary ConfigNode maintains cluster load balance. It is responsible for allocating DataRegion to DataNodes, thereby indicating how data is distributed within the cluster.

DataRegions across different DataNodes constitute DataRegionGroups, signifying the replica group of data replicas. DataRegions denoted by the same color compose a DataRegionGroup in Figure 5. Each DataRegionGroup serves as a hub for a set of time series, handling their respective read and write requests.

2.2 IoTDB Partition Process

add an overview paragraph

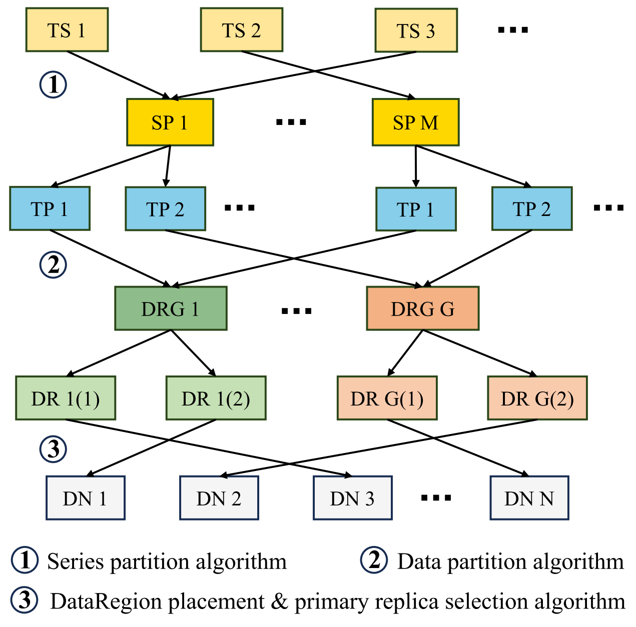

TS 1
TS 2
TS 3
①
SP 1
SP M
TP 1
TP 2
TP 1
TP 2
②
DRG 1
DRG G
DR 1(1)
DR 1(2)
DR G(1)
DR G(2)
③
DN 1
DN 2
DN 3
DN N
① Series partition algorithm
② Data partition algorithm
③ DataRegion placement & primary replica selection algorithm

Figure 6: IoTDB partition process

IoTDB first partitions the time series dimension to reduce the metadata storage burden of massive time series in production environments. Then partition is performed in time dimension based on the property of intensive write load of time series data. The data partition composed of series partition and time partition is thereby generated and assigned to the cluster replica groups.

Definition 2 (partition concepts). *The concepts used in the IoTDB partition process are listed below:*

- **Time series (TS):** Corresponds to the actual sampling time series deployed in the production environment.
- **Series partition slot (SP):** Dividing a batch of time series into a series partition slot according to the series partition algorithm.
- **Time partition slot (TP):** Dividing a time partition slot every fixed time interval.

IoT time series generate time series data through continuous sampling. To manage the load from these time series, an initial step is to distribute them across several virtual containers. IoTDB introduces the concept of series partition slot, ensuring that time series are uniformly distributed across these slots via a series partition algorithm, as depicted in Figure 6 ①. By default, IoTDB employs a hash algorithm for allocating time series to series partition slots. Moreover, to enhance adaptability in various production environments, IoTDB provides an open interface for the series partition algorithm. This flexibility allows users, with a nuanced understanding of their time series' load, to implement custom series partition algorithms like interval or list partition, thereby optimizing control over their database's series partition slots.

Considering that time series data inherently uses timestamps as the primary key, and data timestamps are typically sequential and clustered around recent times, IoTDB introduces the concept

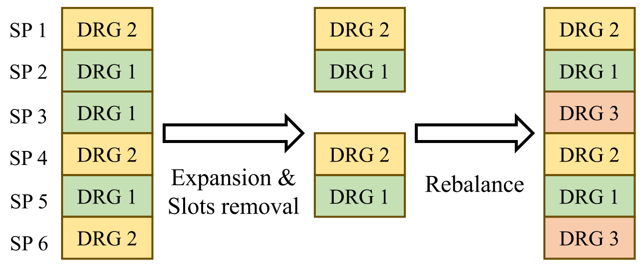


Figure 7: Data allocation table

of time partition slots. Eventually, a data partition will be generated through a series partition slot paired with a time partition slot. Figure 6 ② illustrates how, post the combination of series and time partition slots, the data partitions allocate to DataRegionGroups through our data partition algorithm described in section 3.

Last but not least, as presented in Figure 6 ③ the IoTDB cluster needs to carefully decide on the DataRegion placement schemes and cautiously select the primary DataRegion for each DataRegionGroup. The DataRegion placement schemes affect the load balance and disaster capability as shown in section 4. The primary DataRegion selection influences the computing load balance ability, which is explained in section 5.

3 DATA PARTITION ALGORITHM

This section will first present our data structure to assist the data partition table inside the IoTDB. And elucidates how our data partition algorithm achieves balanced data partition distribution while providing the possibility that avoid data migration.

3.1 Data Allocation Table

To facilitate data integration or mining, many time series data queries are aggregate queries across continuous time partitions, with some not suitable for parallel processing across multiple DataNodes. To address this, we introduce the data allocation table which balances equalization and inheritance. As illustrated in Figure 7. The data allocation table will remain static post the uniform distribution of series partition slots to DataRegionGroups until there's a change in the number of DataRegionGroups, which will happen after cluster expansion. At that juncture, the data allocation table is updated to evenly and randomly redistribute several series partition slots from each DataRegionGroup that existed before the expansion, then reallocating them to the newly formed DataRegionGroup.

3.2 Data Partition Table

The data partition table reflects the actual allocation outcomes, guided by the data allocation table, ensuring both balance and inheritance as Figure 8 demonstrates. Assuming that the time partition k is currently written, the data partitions paired with time partitions that are greater than k will be allocated based on the data allocation table. This setup ensures that data partitions are balanced and maintain a lineage. If the data allocation table is altered due to cluster expansion. Notably, it's unnecessary for the IoTDB cluster to initiate active data partition migrations if the user has

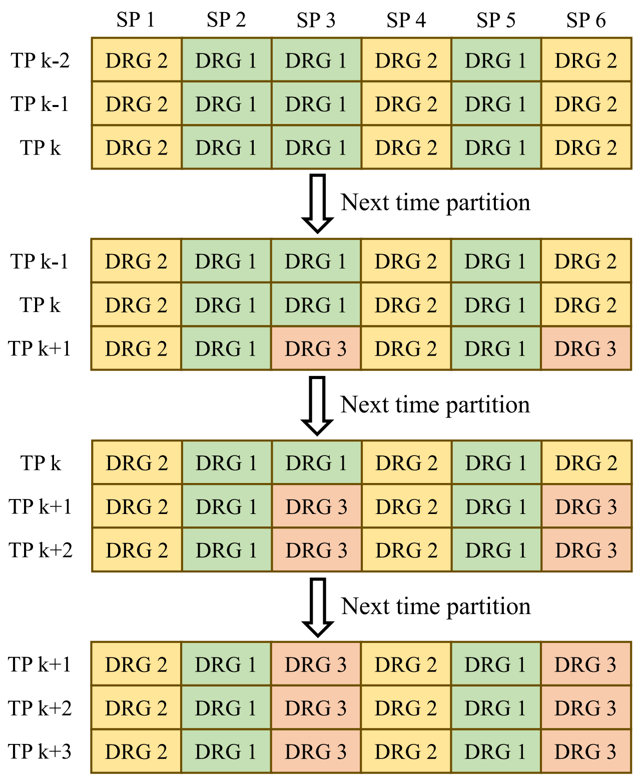


Figure 8: Data partition table

set the TTL when deploying the IoTDB cluster. Assuming that the TTL set by the user is 3 time partitions, then the data partitions corresponding to the time partition less than or equal to $k-3$ have been deleted, and the data partitions within time partition $k-2$ will be deleted as soon as the time partition $k+1$ arrives. Eventually, the cluster achieves a balanced storage load distribution post-expansion and TTL expiration.

4 REPLICA PLACEMENT ALGORITHM

This section delves into our replica placement algorithm. The cluster will assign DataRegions to DataNodes through the DataRegion placement algorithm, taking into account the current resource landscape while ensuring load balance and disaster recovery capability. Table 1 shows the concepts used in this section.

4.1 DataRegion placement Algorithm

This section defines the DataRegion placement problem. Our Greedy Copyset Replication (GCR) algorithm used to solve this problem is then illustrated.

4.1.1 Problem Definition. We model the DataRegion placement problem as an optimization problem, as shown below:

Problem 1 (DataRegion placement problem). *Given the set of N DataNodes $\{d_1, d_2, \dots, d_N\}$ and the placed G DataRegionGroup set $\{g_1, g_2, \dots, g_G\}$ currently in the cluster. The DataRegion placement problem is to find the best placement scheme while satisfying the constraints shown in Formula 1.*

Table 1: Concepts used in replica placement algorithm

Definition	Description
N	The number of DataNodes in the cluster.
G	The number of DataRegionGroups in the cluster.
R	The replication factor, which represents the number of DataRegions held by each DataRegionGroup.
d_i	The i-th DataNode in the cluster.
g_i	A set of d_i where the DataRegions in DataRegionGroup i located, $\forall i, \lvert g_i \rvert = R$.
$g_{i,j}$	The j-th DataNode in g_i.
W_i	The load factor of d_i. Represents the expected maximum number of DataRegions held by d_i.
w_i	The number of actual placed DataRegions in d_i.
S_i	The scatter width of d_i.
$comb_i$	The 2-combination set generated by g_i.

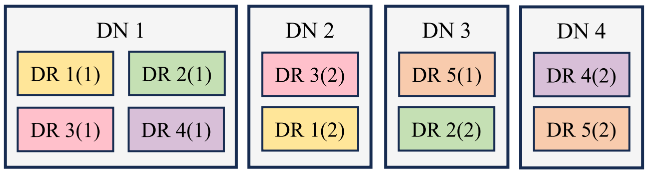


Figure 9: DataRegion placement example

$$\begin{array}{llll} max & \min_i \{S_i\} & i = 1, \dots, N & \\ s.t. & w_i \le W_i; & i = 1, \dots, N & (1) \\ & |w_i - w_j| \le 1; & 1 \le i < j \le N & \end{array}$$

- **Optimize:** $\max\min_i \{S_i\}$. Maximize the minimum S_i in the cluster.
- **Constraint 1:** $\forall i, w_i \le W_i$. As shown in Table 1, the number of DataRegions w_i held by d_i is expected to be less than W_i.
- **Constraint 2:** $\forall i \ne j, |w_i - w_j| \le 1$. Each DataRegion maintains a batch of data partitions. Therefore, if the difference in the number of DataRegions held by different DataNodes is too large, the storage load of DataNodes varies significantly.

Definition 3 (scatter width [4]). *Given the set of N DataNodes $\{d_1, d_2, \dots, d_N\}$ and the placed G DataRegionGroup set $\{g_1, g_2, \dots, g_G\}$ currently in the cluster. The scatter width S_i of d_i is equals to $|\cup_{j, d_i \in g_j} g_j| - 1$.*

For instance, the scatter width of d_1 in Figure 9 is 3. Because $d_1 \in g_1, g_2, g_3, g_4$ and $|g_1 \cup g_2 \cup g_3 \cup g_4| - 1 = 3$. Furthermore, d_2 can share the load from g_1, g_3, d_3 can share the load from g_2 and d_4 can share the load from g_4 when d_1 is down. Therefore the larger the scatter width of each DataNode the better.

Definition 4 (2-combination set). *Given the allocated G DataRegionGroup set* $\{g_1, g_2, \ldots, g_G\}$ *currently in the cluster. The 2-combination set generated by* g_i *is* $\{\forall_{j\neq k}\{g_{i,j}, g_{i,k}\}\}$.

For example, the 2-combination sets generated by g_1, g_2 in Figure 9 are:

DataRegionGroup 1:$\{\{1,2\}\}$, DataRegionGroup 2:$\{\{1,3\}\}$

4.1.2 Greedy Copyset Replication. The IoTDB cluster adaptively increases its DataRegionGroups number based on the prevailing load, with each placement cycle contemplating the scheme for all DataRegions in the upcoming DataRegionGroup. The GCR algorithm firstly involves a multi-criteria sorting mechanism for all DataNodes within the cluster:

- **First Criterion:** w_i in ascending order. This approach aims to equalize the number of DataRegions in each DataNode, thereby ensuring load balance.
- **Second Criterion:** S_i in ascending order. Prioritizing DataNodes with lower scatter width allow for an enhancement in their scatter width, contributing to cluster disaster recovery capability.
- **Third Criterion:** A random weight assigned to each DataNode, sorted from lowest to highest. This randomization step is crucial when the first two criteria result in a tie among DataNodes, therefore preventing the algorithm from settling into a local optimal solution and ensuring a more dynamic and effective distribution strategy.

For instance, the sorting result of Figure 9 might be $[d_2, d_4, d_3, d_1]$. Because d_2's current scatter width and the number of DataRegions it holds are both the lowest ($w_2 = 2, S_2 = 1$). Since $w_3 = w_4 = 2, S_3 = S_4 = 2$, the relative ordering of d_3, d_4 is thus decided based on randomly assigned weights. Since d_1 accommodates 4 DataRegions with a scatter width of 3, it is assigned the lowest priority in the sorting result.

Following the multi-criteria sorting result of DataNodes, the GCR algorithm then employs a depth-first search (DFS) to explore different placement possibilities, as detailed in Algorithm 1. The DFS algorithm also applies a multi-criteria sorting logic to evaluate and compare potential placement schemes. Let the scheme currently being retrieved as *currentScheme*. The schemes' priority is governed by the following criteria:

Algorithm 1 Greedy Copyset Replication

```
Output: An optimal DataRegionGroup allocation scheme
1: sort(cluster.dataNodes)
2: $optimalSchemes \leftarrow []$
3: $currentScheme \leftarrow []$
4: GCRDfs($optimalSchemes, currentScheme$)
5: return Random.choice($optimalSchemes$)
```

- **First Criterion:** $\sum_{d_i \in currentScheme.dataNodes} w_i$ in ascending order. With a preference for lower totals to promote uniform distribution of DataRegions across all DataNodes.
- **Second Criterion:** $\sum_{i=1}^{G} |currentScheme.comb \cap comb_i|$ also in ascending order. The intersection size of the 2-combination set generated by the current placement scheme with the existing 2-combination sets in the cluster, with a smaller intersection being advantageous. The scatter width for the involved DataNodes will be maximized when the 2-combinations generated by different DataRegionGroups are non-overlapping.

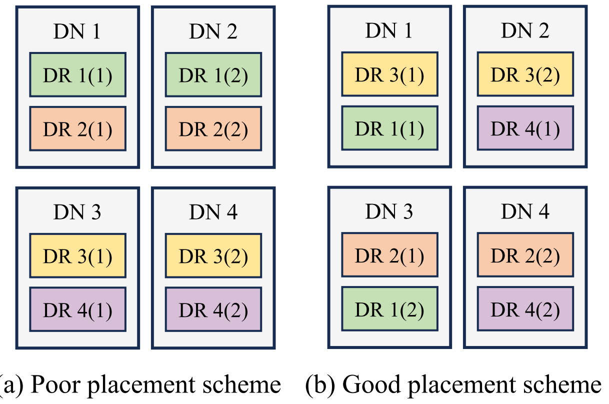


Figure 10: Possible placement schemes

Figure 10 (a) is an example of a poor DataRegion placement scheme. Although each DataNode holds the same number of DataRegions, the DataRegionGroups correspond to the following 2-combination sets:

$$g_1 : \{\{1,2\}\},\ g_2 : \{\{1,2\}\},\ g_3 : \{\{3,4\}\},\ g_4 : \{\{3,4\}\}$$

Those overlapped 2-combinations result in $\forall i, S_i = 1$. Therefore, measuring the *currentScheme* according to the criteria described above can lead to a better result. As shown in Figure 10 (b), the DataRegionGroups correspond to the following 2-combination sets:

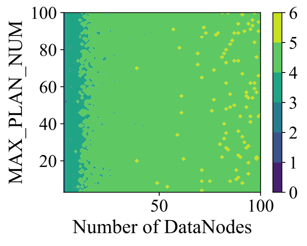
$$g_1 : \{\{1,3\}\},\ g_2 : \{\{3,4\}\},\ g_3 : \{\{1,2\}\},\ g_4 : \{\{2,4\}\}$$

Since the 2-combinations are all disjoint, each DataNode reaches its maximal scatter width $\forall_i, S_i = 2$. This solution not only distributes the load evenly but also maximizes the disaster recovery capability for the cluster.

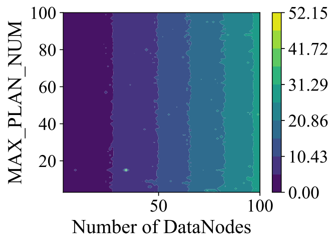
Algorithm 2 illustrates the exhaustive DFS search process, where each new scheme is compared against the best ones found so far. To enhance efficiency, if the search encounters a multitude of solutions with identical keyword weights and they are the optimal solutions at that point, the process will be terminated once the solution count hits the predefined *MAX_PLAN_NUM*. This pruning will ensure swift determination of the next DataRegionGroup placement scheme in our simulated experiment. Additionally, to expedite the assessment of how a placement scheme impacts the DataNode scatter width within the cluster, our algorithm employs the method that counts 2-combinations. This approach is advantageous because calculating the 2-combinations for the current scheme requires only near-constant time complexity $O(\binom{R}{2})$, compared to the at least $O(N)$ time needed to recalculate a specific DataNode's scatter width. In cases where multiple placement schemes are equally viable, any one of them may be selected as the definitive placement scheme for the ensuing DataRegionGroup.

Algorithm 2 GCR DFS

```
Input: $optimalPlans, currentPlan$
1:  if WorseThanOptimal($optimalPlans, currentPlan$) then
2:      return
3:  end if
4:  if $currentPlan.size() == replicationFactor$ then
5:      if BetterThanOptimal($optimalPlans, currentPlan$) then
6:          $optimalPlans \leftarrow \{currentPlan\}$
7:      else
8:          $optimalPlans \leftarrow optimalPlans \cup \{currentPlan\}$
9:      end if
10: end if
11: for all $dataNode \in cluster.dataNodes$ do
12:     if $!currentPlan.contains(dataNode)$ then
13:         $currentPlan \leftarrow currentPlan \cup \{dataNode\}$
14:         UpdateCharacteristic($currentPlan$)
15:         GCRDfs($optimalPlans, currentPlan$)
16:         if $optimalPlans.size() == MAX\_PLAN\_NUM$ then
17:             return
18:         end if
19:         $currentPlan \leftarrow currentPlan - \{dataNode\}$
20:         UpdateCharacteristic($currentPlan$)
21:     end if
22: end for
```



(a) Minimum scatter width



(b) Allocation avg time(us)

Figure 11: MAX_PLAN_NUM simulation

Concerning the GCR's complexity, the pre-processing of all 2-combinations of placed DataRegionGroups requires $O(\frac{\sum_i D_i}{R} \times \binom{R}{2})$ time. Sorting all DataNodes takes $O(N \log N)$ time, and the DFS incurs a maximal cost of $O(\binom{N}{R})$. The introduction of a solution cap through pruning ensures the algorithm can swiftly generate a placement scheme for 100 DataNodes in microseconds during simulations as presented in Figure 11.

4.2 Data Disaster Recovery Model

This section will present the disaster recovery model and prove the disaster tolerance of the cluster.

4.2.1 The Disaster Model.

DEFINITION 5 (DISABLED DATAREGIONGROUP). *Define g_i is disabled when $\forall d_i \in g_i$ are down.*

THEOREM 1 (THE PROBABILITY OF DISABLED DATAREGIONGROUP). *Given the set of N DataNodes $\{d_1, d_2, \ldots, d_N\}$ and the placed G DataRegionGroup set $\{g_1, g_2, \ldots, g_G\}$ currently in the cluster. Let $X(N, M)$ be the number of disabled DataRegionGroups when there are M DataNodes in the cluster that are down. The probability that exists at least one disabled DataRegionGroup is given by Formula 2.*

$$P(X(N,M) \geq 1) = 1 - e^{-\binom{M}{R} \cdot \frac{G}{\binom{N}{R}}} \tag{2}$$

PROOF. In cluster comprising N DataNodes with a replication factor of R. The cluster computes the total number of DataRegionGroups G, which can support based on the load factor of each DataNode, where $G = \frac{\sum_i W_i}{R}$. The number of potential placement schemes for R DataRegions within a DataRegionGroup is $\binom{N}{R}$. Assuming that there are R DataNodes in the cluster are broken down. These DataNodes correspond to a potential DataRegionGroup placement scheme. The probability that this scheme happens to be selected by the cluster as an actual placement scheme for a particular DataRegionGroup is given by Formula 3.

$$p = \frac{G}{\binom{N}{R}} \tag{3}$$

Consider the scenario where M DataNodes are broken down in the cluster. These DataNodes correspond to $F(M) = \binom{M}{R}$ potential DataRegionGroup placement schemes. Then $\forall g_i' \in F(M)$, where g_i' denotes a potential scheme that is impacted by the break down DataNodes, g_i' has the probability of p to be selected by the cluster as an actual scheme. Therefore, $X(N, M)$ follows the binomial distribution. i.e. $X(N, M) \sim B(F(M), p)$ as described in Formula 4. Furthermore, the binomial distribution can be approximated by using the Poisson distribution. i.e. $X(N, M) \sim Possion(\lambda = F(M)p)$ as described in Formula 5. Thereby, the Theorem1 can be proved through Formula 6.

$$P(X(N,M) = k) = \binom{F(M)}{k} \cdot p^k \cdot (1-p)^{F(M)-k}, k \in [0, F(M)] \tag{4}$$

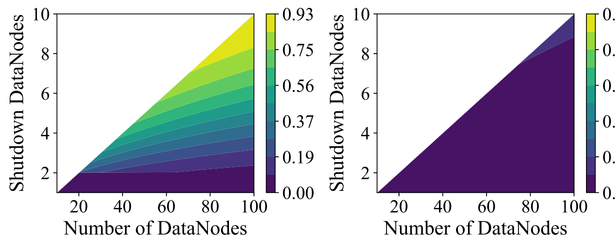
$$P(X(N,M) = k) = \frac{e^{-\lambda}\lambda^k}{k!}, \lambda = F(M)p, k \in [0, F(M)] \tag{5}$$

$$\begin{aligned} P(X(N,M) \geq 1) &= 1 - P(X(N,M) = 0) \\ &= 1 - e^{-F(M)p} \\ &= 1 - e^{-\binom{M}{R} \cdot \frac{G}{\binom{N}{R}}} \end{aligned} \tag{6}$$

□

Given the typical IoTDB cluster size $N \in [10, 100]$ and the default configuration parameters $\forall i, W_i = 6$, and considering the scenario where up to 10% of the DataNodes are down ($M \in [1, 0.1N]$), the probability distribution for $P(X(N, M) \geq 1)$ when set common replication factor $R \in \{2, 3\}$ is shown as Figure 12.

As a result, in the scenario where users demand high availability, our algorithm guarantees that an IoTDB cluster with configuration $R = 3, N \in [10, 100]$ can satisfy this requirement. The likelihood of encountering a disabled DataRegionGroup remains below 3% when the downtime affects less than 5% of the DataNodes. Furthermore, should the downtime extend to 10% of the DataNodes, the probability of occurring at least one disabled DataRegionGroup is approximated 14%. However, this level of availability can't be assured, when the user sets $R = 2$ due to storage cost, because of the limitation of usable combinations. Considering this existing $R = 2$



(a) R=2, P(X(N, M)>=1) (b) R=3, P(X(N, M)>=1)

Figure 12: $P(X(N,M) \geq 1)$ simulation

IoT scenario, we present the recovery model described in section 4.2.2.

4.2.2 The Recovery Model. In the ideal DataRegion placement scheme, each DataRegion can offer $R-1$ scatter width to the DataNode where it is located, thus the maximum scatter width for d_i is $w_i(R-1)$. However finding a perfect placement scheme is difficult and often fraught with constraints, which will be presented in the section 7. Therefore, the DFS is used in our algorithm and gives the following theorem 2.

THEOREM 2 (MINIMAL SCATTER WIDTH). *Given the set of N DataNodes $\{d_1, d_2, \ldots, d_N\}$ and the placed G DataRegionGroup set $\{g_1, g_2, \ldots, g_G\}$ generated by the GCR algorithm. The minimal scatter width of each DataNode is given by Formula 7.*

$$\forall i, S_i \geq \min\{w_i - 1, N - 1\} \tag{7}$$

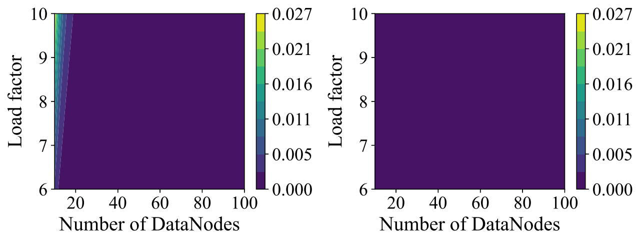
PROOF. The GCR algorithm imposes a constraint on the first criteria of the DataRegionGroup placement scheme. Ensuring that in the IoTDB cluster, each DataNode is always placed in the first DataRegion before being placed in the second one. Until $\forall i, W_i - 1 \leq w_i \leq W_i$. This constraint can be simplified by generating $N-$*permutation* conforming to GCR constraints in turn and dividing each permutation into N/R actual placement schemes of DataRegionGroups. The probability of a potential DataRegionGroup placement scheme g appearing in a given permutation is represented by Formula 8. And $\forall i, d_i$ will appear in exactly w_i permutations.

$$p = \frac{N/R}{\binom{N}{R}} \tag{8}$$

$$\begin{aligned} P(Y(g,w) > 1) &= 1 - \sum_{k=0}^{2} P(Y(g,w) = k) \\ &= 1 - e^{-\lambda} - \lambda e^{-\lambda} - \frac{\lambda^2 e^{-\lambda}}{2} \end{aligned} \tag{9}$$

Let $Y(g,w)$ denote the number of occurrences of a specific DataRegionGroup placement scheme in all w permutations. Then $Y(g,w)$ follows a binomial distribution. i.e. $Y(g,w) \sim B(w,p)$. It can be approximated as a Poisson distribution. i.e. $Y(g,w) \sim Possion(\lambda = wp)$. Thereby, the probability of $Y(g,w) > 2$ is given in Formula 9. The Figure 13 shows the probability distribution. Proving that $\forall w \in [6,10], N \in [10,100]$, it is almost impossible for a specific g to be selected more than twice.

Therefore, the worst scenario to scatter width for a particular d_i is placing d_i to a certain g twice, and the S_i will still be increased



(a) R=2, P(Y(g, w)>1) (b) R=3, P(Y(g, w)>1)

Figure 13: $P(Y(g,w) > 1)$ simulation

Table 2: Concepts used in primary replica selection algorithm

Definition	Description
p_i	The d_i where the primary replica of DataRegionGroup i located.
dp_i	The number of primary replicas distributed to d_i.

1. However, each of the remaining $w-2$ placement schemes can increase S_i at least 1, which leads to the $S_i \geq w_i - 1$. On the other hand, considering the fact that $S_i \leq N-1$, the lower bound of S_i should be $\min\{w_i - 1, N - 1\}$. □

Now considering the cost-effective scenario the user only demands 2 replicas. This minimal amount of scatter width among the cluster DataNodes still provides a dependable disaster recovery capability. For one thing, our algorithm ensures that any single DataNode's computing load will be split into multiple other DataNodes to prevent potential overload. For another thing, any shutdown DataNode will generate a network transmit load due to log synchronization. Those benefits to the IoTDB cluster will be proved in section 6.

5 PRIMARY REPLICA SELECTION ALGORITHM

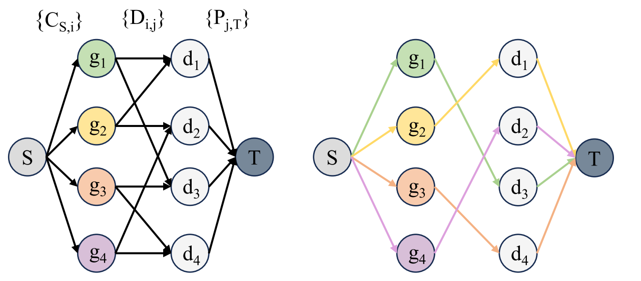
This section defines the primary replica selection problem and presents our Cost Flow Distribute (CFD) algorithm that can always find the best distribution for this problem. Table 2 shows the concepts used in this section.

5.1 Problem Definition

add an overview paragraph

The data partition algorithm ensures that each DataRegionGroup holds an equal amount of data partitions under the latest time partition. And the DataRegion placement algorithm ensures that each DataNode holds an equal amount of DataRegions. Based on the balanced distribution of partitions and DataRegions provided by the above two algorithms, we believe that as long as each DataNode holds an equal number of primary replicas, the cluster can achieve computing load balance. Therefore we define the primary replica selection problem as follows:

PROBLEM 2 (PRIMARY REPLICA SELECTION PROBLEM). *Given the set of N DataNodes $\{d_1, d_2, \ldots, d_N\}$ and the placed G DataRegionGroup set $\{g_1, g_2, \ldots, g_G\}$ currently in the cluster. The primary replic-*



(a) Traffic network (b) Optimal primary replica distribution

Figure 14: CFD model example

a selection problem is to select a primary replica p_i for each g_i while meeting the following constraint:

- **Constraint:** $\forall i \neq j, |dp_i - dp_j| \leq 1$. Each primary replica undertakes the write load and the data synchronization task for the DataRegionGroup. Therefore, if the difference in the number of primary replicas held by different DataNodes is too large, the computing load of DataNodes varies significantly.

5.2 Min Cost Flow Distribution

The primary replica shoulders the principal workload of the replica group. Considering the balanced data partition table, it's reasonable to assume that the workload brought to each DataRegionGroup's primary replica is approximately uniform. Consequently, it's crucial to ensure that the DataNodes within the cluster maintain an equal number of primary replicas.

Therefore, we introduce the CFD algorithm modeling based on the classic min cost flow problem that can be solved by comprising of Dinic algorithm [5] and SPFA algorithm [15] in polynomial time, to generate the optimal primary replica distribution. For example, Figure 14 (a) is the traffic network modeled from Figure 10 (b):

Definition 6 (point set). *Given the set of N DataNodes $\{d_1, d_2, \ldots, d_N\}$ and the placed G DataRegionGroup set $\{g_1, g_2, \ldots, g_G\}$ currently in the cluster. The point set in traffic network is defined as: $V = \{S\} \cup \{g_i\} \cup \{d_j\} \cup \{T\}$, where:*

- S: represents the source point of the traffic network.
- $\{g_i\}$: represents the set of DataRegionGroup points, where g_i corresponds to the DataRegionGroup i.
- $\{d_j\}$: represents the set of DataNode points, with d_j representing DataNode j.
- T: represents the traffic network convergence point.

Definition 7 (edge set). *Given the set of N DataNodes $\{d_1, d_2, \ldots, d_N\}$ and the placed G DataRegionGroup set $\{g_1, g_2, \ldots, g_G\}$ currently in the cluster. The edge set in traffic network is defined as: $E = \{C_{S,i}\} \cup \{D_{i,j}\} \cup \{P_{j,T}\}$, where:*

- $\{C_{S,i}\}$ (capacity edge): Connects S with $\{g_i\}$ to define the number of primary replicas each DataRegionGroup should select. Capacity: 1. Only one primary replica can be designated per DataRegionGroup. Cost: 0. Since each DataRegionGroup requires a primary replica, the capacity edge does not incur any cost.
- $\{D_{i,j}\}$ (distribution edge): Connects $\{g_i\}$ with $\{d_j\}$ to represent the selection of a DataRegionGroup's primary replica to a specific DataNode. Capacity: 1. A DataRegionGroup can generate at most one primary replica. Cost: $\Delta(i, j)$ is 0 if $p_i = d_j$, and 1 otherwise. Emphasizing the preference to reduce the number of switchover times for the primary replicas if possible.
- $\{P_{j,T}\}$ (load edge): Links $\{d_j\}$ with T, indicating dp_j in the final distribution. Capacity: ∞, $\forall_i, dp_i$ doesn't need to be limited. Cost: $f(dp_j) = \sum_{k=1}^{dp_j} k^2$, represents the cost incurred when d_j holds dp_j primary replicas. Based on the Jensen's inequality (shown at Formula 10), this cost function promotes the distribution of primary replicas equity across DataNodes.

$$nf(k) < \sum_{i=1}^{n} f(k_i), \forall 1 \leq k_1 \leq \cdots \leq k_n, nk = \sum_{i=1}^{n} k_i, \exists k_i \neq k \quad (10)$$

An optimal primary replica distribution of Figure 14 (a) is shown as Figure 14 (b). Where $p_1 = d_3, p_2 = d_1, p_3 = d_4, p_4 = d_2$, satisfies the problem constraint.

6 EVALUATION

This section will first present our programming simulated evaluation on both replica placement algorithm and primary replica selection algorithm. Then illustrate the evaluation on IoTDB.

6.1 Simulated Evaluation

6.1.1 GCR Algorithm simulation. Set the number of DataNodes $N \in [3, 100]$ and the load factor $W = 6$. We process the simulated evaluation 100 times for every N. Figure 15 shows the range of the number of DataRegions held by different DataNodes w_i, and the minimal scatter width S_i of each DataNode. The GCR algorithm (denoted as green line) ensures the minimum w_i range while providing the maximal minimum S_i. The Copyset (denoted as red line) and Tiered Replication (denoted as yellow line) algorithms provide a good minimum scatter width, but because of the randomness in the algorithms, there is no guarantee that DataNodes hold an equal number of DataRegions. The Greedy algorithm (denoted as blue line), on the contrary, can ensure a balanced DataRegion distribution, but the scatter width obtained by DataNode is very limited.

increase fi
height

remove the
white spac
in surroun
ing figs

6.1.2 CFD Algorithm simulation. Set the number of DataNodes $N \in [3, 100]$ and the load factor $W = 6$. Figure 16 simulates the range of the number of primary replicas among DataNodes between different algorithms. We also process the simulated evaluation 100 times for each N. Only the CFD algorithm (denoted as green line) ensures the optimal distribution. The Random (denoted as red line) algorithm doesn't perform well in simulation experiments, because it has the possibility of falling into an awful solution, and the more experiments the greater the probability of occurrence. The greedy algorithm (denoted as blue line) may obtain a locally optimal solution. Notably, the greedy algorithm performed worse in the experiment with $R = 2$, which is because the setting of $R = 2$ produced more replica groups, increasing the number of times the greedy algorithm needed to make a decision. Because the decision process of finding the optimal solution is more complex, and the more decisions the greedy algorithm makes, the easier for it to deviate from the optimal solution.

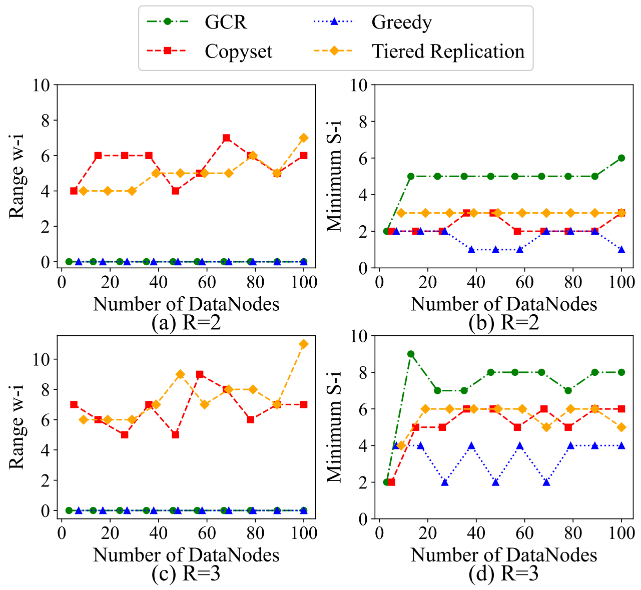


Figure 15: Replica placement algorithm simulation

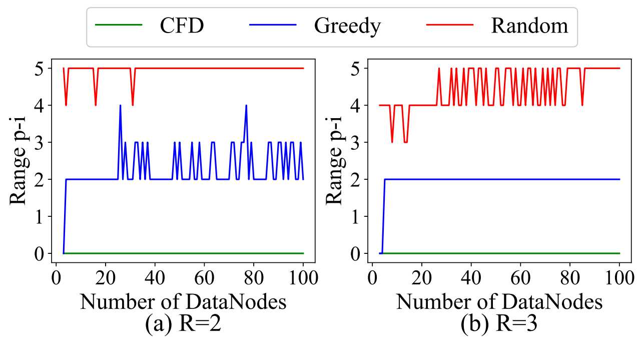


Figure 16: Primary replica selection algorithm simulation

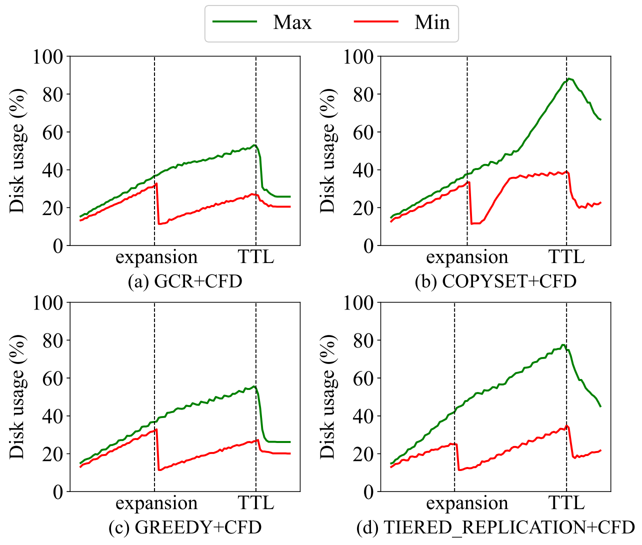


Figure 17: Disk usage during expansion experiment

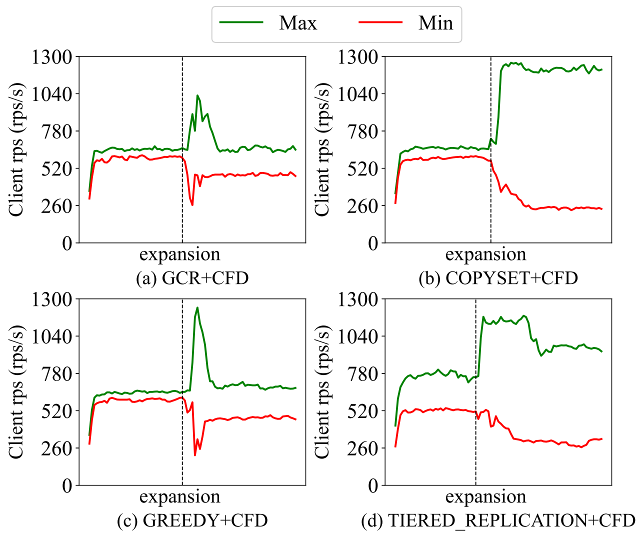


Figure 18: Client rps during expansion experiment

6.2 IoTDB Evaluation

We evaluated our data partition, replica placement, and primary replica selection algorithms on an IoTDB cluster consisting of 11 nodes on Tencent Cloud. Each node is equipped with a 4-core CPU and 8 GB of main memory, with 100 GB of SSD storage and 1.5 GB ethernet.

We employed the control variable method to select algorithm combinations for our evaluation. The replica placement algorithm can be GCR, Copyset, Tiered Replication, or Greedy when the primary replica selection algorithm is fixed as the CFD algorithm. When the replica placement algorithm is fixed as the GCR algorithm, the primary replica selection algorithm can be CFD, Greedy, or Random.

The IoTDB evaluation process is divided into two parts. The first part is an expansion experiment, evaluating the adaptability of our algorithms to cluster expansion scenarios. The second part is a disaster recovery experiment, presenting the tolerance of our algorithms to single DataNode failure. The computing and storage load balance capabilities provided by our algorithms are demonstrated in the course of two evaluations.

6.2.1 *Expansion Experiment.*

We performed the expansion experiment as follows:

- **Step 1.** Deploy an IoTDB cluster consisting of 1 ConfigNode and 4 DataNodes with TTL sets to 5 minutes and the sampling frequency is once every 15 seconds.
- **Step 2.** Enable the IoT-benchmarks deployed on 2 nodes to continuously write time series data to the IoTDB cluster.

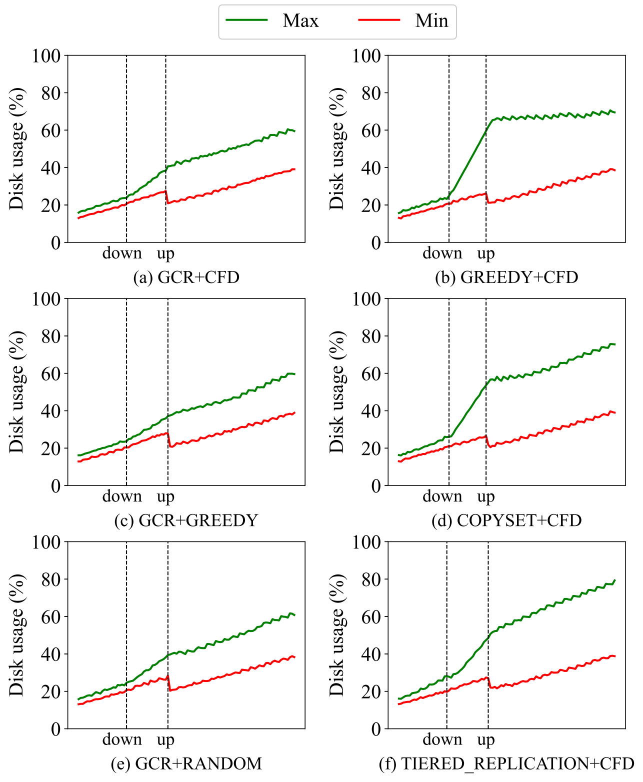


Figure 19: Disk usage during disaster recovery experiment

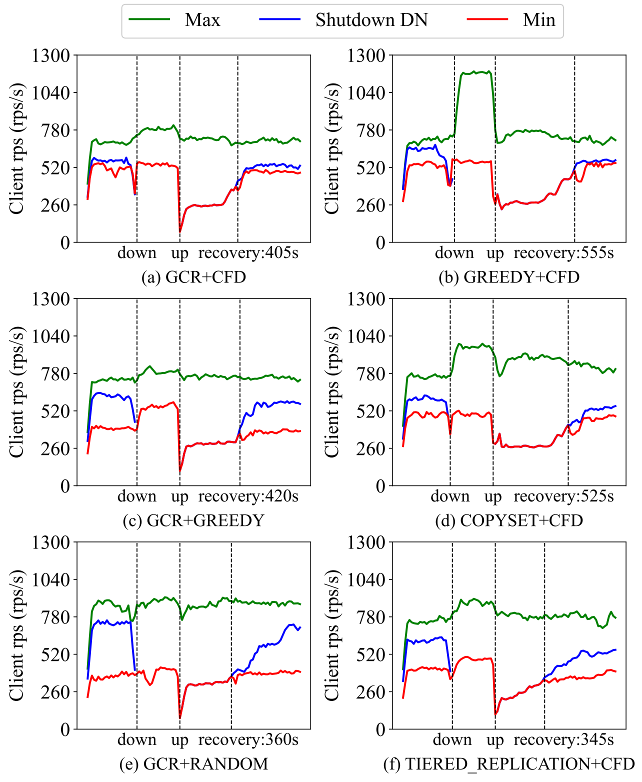


Figure 20: Client rps during disaster recovery experiment

- **Step 3.** Expand 4 DataNodes to the IoTDB cluster after 5 minutes.
- **Step 4.** Stop IoT-benchmarks after 5 minutes.
- **Step 5.** Repeat the above steps 5 times to collect experimental data.

At each sampling moment during the experiment, we collect data for all DataNodes and keep the maximum and minimum values. For all five experiments, five sets of maximum values and five sets of minimum values will be obtained at each moment. We median these values and plot them as Figure 17 and Figure 18. In the expansion experiment, we fixed the primary replica selection algorithm as CFD to observe the load balance capability of different replica placement algorithms in the expansion scenario. Figure 17 shows the maximum (denoted as green line) and minimum (denoted as red line) values of the median sample of disk usage percentage for different algorithms during the experiment, while 18 shows the maximum (denoted as green line) and minimum (denoted as red line) values of the median sample of client request per second.

As illustrated in Figure 17, the disk usage percentage of each DataNode decreases after the TTL expires. With the combination of our data partition algorithm, both GCR and Greedy algorithms ensure the storage load balance after TTL. Copyset and Tiered Replication do not ensure storage balance after cluster expansion and TTL expires, because their randomness results in an uneven number of replicas held by each DataNodes, and both the new and the old DataNodes have the same weight when deciding most placement schemes.

The metric client request per second (rps) is used to evaluate the IoTDB cluster's computing load balance as shown in Figure 18. Based on the uniformly distributed replicas, both GCR and Greedy algorithms can demonstrate better computing load balance. Copyset and Tiered Replication still fail because of their randomness and the imbalance of the computing load can grow as the cluster expands.

6.2.2 Disaster recovery Experiment. The procedure of the disaster recovery experiment is as follows:

- **Step 1.** Deploy an IoTDB cluster consisting of 1 ConfigNode and 8 DataNodes with sampling every 15 seconds.
- **Step 2.** Enable the IoT-benchmarks deployed on 2 nodes to continuously write time series data to the IoTDB cluster.
- **Step 3.** Shutdown DataNode 1 at 5 minute.
- **Step 4.** Restart DataNode 1 after 5 minutes.
- **Step 5.** Stop IoT-benchmarks after 15 minutes.
- **Step 6.** Repeat the above steps 5 times to collect experimental data.

We generate Figure 19 and Figure 20 by using the same sample selection method as in the expansion experiment. In Figure 19, we also use the green line to show the median sample with the highest DataNode disk usage percentage and the red line to present the lowest. In Figure 20, we additionally annotated the variation of the shutdown DataNode's rps during the whole experiment, denoted as the blue line.

Figure 19 shows the disk usage percentage of the cluster during the whole disaster recovery experiment. During the shutdown period of DataNode 1, as we analyzed in Figure 3, it's fixed pairing DataNode has to store additional logs to process data synchronization when it restarts, which causes the storage load on the pairing DataNode to increase rapidly as shown in Figure 19 (b). In addition, after DataNode 1 is recovered, the load of synchronization can only be borne by the paired DataNode. As a result, the paired DataNode cannot process client requests properly. In this case, the disk space released by deleting synchronized logs and the amount of new data written by the client will be evenly balanced. With Copyset (19 (d)) and Tiered Replication (19 (f)) algorithms, because we make the IoT-benchmark write for a longer time in this experiment, the difference in the number of replicas held by DataNodes shows up as a disk usage percentage range that increases over time. It is thus easier for a particular DataNode to become the storage bottleneck in a real production environment.

Figure 20 shows the rps distribution of the cluster when a single DataNode fails. The fixed paired DataNode in the Greedy algorithm will bear very high service pressure during the failure phase, as shown in Figure 20 (b). While the GCR (Figure 20 (a)), Copyset (Figure 20 (d)) and Tiered Replication (Figure 20 (f)) algorithms can distribute the client requests that should be handled by DataNode 1 more evenly.

Further, we define the time when the rps of DataNode 1 is greater than the cluster's minimum as the recovery time. Because the bottleneck for providing services in the cluster is no longer DataNode 1, which has failed before. Both GCR, Copyset, and Tiered algorithms all benefited from the larger scatter width of DataNode 1, which spreads the log synchronization task over more cluster DataNodes, making their recovery time superior to that of Greedy. Though under this definition, the Tiered Replication algorithm takes less recovery time than the GCR algorithm, for one thing, DataNode 1 in the Tiered Replication algorithm has not yet restored its rps to the same level before the shutdown. The recovery time required for the same performance as shutdown before is similar to that of the GCR algorithm. For another thing, DataNodes that assist in recovery then become the bottleneck for the cluster to client service because the randomness of the Tiered Replication algorithm makes it hold fewer replicas than the DataNode can achieve its better service performance.

In Figure 20 (a)(c)(e), the CFD algorithm makes the computing load of DataNodes in the cluster most evenly distributed by generating a stable optimal primary replica selection scheme. In this problem, Greedy is prone to a local optimal solution, so it cannot guarantee the same stable optimal distribution as CFD. The Random algorithm is our simulation of the self-election algorithm, because each replica has the potential to be the first one to initiate a vote request, and the internal network of a cluster is often very variable, each replica thereby can become the primary one.

7 RELATED WORK

Finding the optimal DataRegion allocation schemes that maximize the scatter width of cluster DataNodes can be reduced to the BIBD [6] (Balanced Incomplete Block Design) issue previously explored in combinatorial design research. A corresponding BIBD model $(N, R, \lambda) - design$ can be formulated for the Problem 1, where N is the number of DataNodes, R is the replication factor and $\lambda = 1$. Previous research [22] has proved that a (W, R) difference set can generate perfect BIBD-designs when $N = W(R-1)+1, G = (W^2(R-1)+W)/W, W, R, 1)$. There are also studies on designing perfect allocation schemes based on the BIBD model [7]. However, it's impractical to mandate the deployment of a fixed number of nodes in a cluster or to set a fixed load factor without considering the varying capacities of different machines in our production environments.

A recent hot topic in replica placement algorithms is dynamic replication, and there are examples of implementation and testing on HDFS [1, 29] and other distributed systems [2, 19]. The Copyset mathematical model is also used to combine with this strategy [13]. But this kind of algorithm doesn't work in our production environment. Users either choose 2 replicas for cost and efficiency reasons or choose 3 replicas to get higher data availability. Given the strong consistency consensus protocol, 3 replicas happen to be the minimum requirement to provide consistency, so the cluster has little need to adjust the number of replicas.

The widely used AI technology also affects the ecology of database systems. And some scholars have combined the AI technology with the Copyset model [14, 28]. AI technology is also used in cluster computing load balance, such as the ZHT [12] and the Dejavu [26]. However, the computing resources required by AI models also require additional cost burdens on growing clusters. Recall that in our IoT scenario, the computing load generated by the time series is relatively stable. Therefore, we choose to implement compatible and effective algorithms in problem 1 and problem 2. Such that the users can consider bringing higher value in data mining and data analysis when they want to introduce AI models.

8 CONCLUSION

Our research is based on the TTL feature of time series data, the requirement for users to gradually expand their cluster according to the number of time series, and the high intensive write load in IoT scenarios. Therefore, we propose a data partition algorithm using TTL to avoid data migration after cluster expansion, and the evaluation proves that our algorithms can eventually complete the storage balance after TTL. Our proposed GCR replica placement algorithm achieves better load balance and disaster recovery capability in the evaluation, which makes it more suitable for IoT scenarios where cluster are gradually expanded. To withstand the high intensive write load in the IoT scenario, we propose the CFD primary replica selection algorithm that can always obtain the theoretical optimal solution, and its effectiveness is proved in our evaluations.

ACKNOWLEDGMENTS

This work is supported in part by 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] Cristina L. Abad, Yi Lu, and Roy H. Campbell. 2011. DARE: Adaptive Data Replication for Efficient Cluster Scheduling. In *2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011*. IEEE Computer Society, 159–168. https://doi.org/10.1109/CLUSTER.2011.26

[2] Motaz A. Ahmed, Mohamed Helmy Khafagy, Masoud E. Shaheen, and Mostafa R. Kaseb. 2023. Dynamic Replication Policy on HDFS Based on Machine Learning Clustering. *IEEE Access* 11 (2023), 18551–18559. https://doi.org/10.1109/ACCESS.2023.3247190

[3] 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

[4] 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

[5] 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.

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

[7] 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

[8] HBase. 2024. *HBase.* https://hbase.apache.org/ (2024, March). [Online]. Available: https://hbase.apache.org/.

[9] IoTDB. 2024. *IoTDB.* https://iotdb.apache.org/ (2024, March). [Online]. Available: https://iotdb.apache.org/.

[10] 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

[11] 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

[12] 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

[13] Jinwei Liu and Haiying Shen. 2016. A popularity-aware cost-effective replication scheme for high data durability in cloud storage. In *2016 IEEE International Conference on Big Data (IEEE BigData 2016), Washington DC, USA, December 5-8, 2016*, James Joshi, George Karypis, Ling Liu, Xiaohua Hu, Ronay Ak, Yinglong Xia, Weijia Xu, Aki-Hiro Sato, Sudarsan Rachuri, Lyle H. Ungar, Philip S. Yu, Rama Govindaraju, and Toyotaro Suzumura (Eds.). IEEE Computer Society, 384–389. https://doi.org/10.1109/BIGDATA.2016.7840627

[14] Jinwei Liu, Haiying Shen, Hongmei Chi, Husnu S. Narman, Yongyi Yang, Long Cheng, and Wingyan Chung. 2021. A Low-Cost Multi-Failure Resilient Replication Scheme for High-Data Availability in Cloud Storage. *IEEE/ACM Trans. Netw.* 29, 4 (2021), 1436–1451. https://doi.org/10.1109/TNET.2020.3027814

[15] 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.

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

[17] 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

[18] OpenTSDB. 2024. *OpenTSDB.* http://opentsdb.net/ (2024, March). [Online]. Available: http://opentsdb.net/docs/build/html/user_guide/backends/hbase.html.

[19] Evangelos Papapetrou and Aristidis Likas. 2022. A replication strategy for mobile opportunistic networks based on utility clustering. *Ad Hoc Networks* 125 (2022), 102738. https://doi.org/10.1016/J.ADHOC.2021.102738

[20] Patroni. 2024. *Patroni.* (2024, March). [Online]. Available: https://github.com/zalando/patroni.

[21] Karen Rose, Scott Eldridge, and Lyman Chapin. 2015. The internet of things: An overview. *The internet society (ISOC)* 80, 15 (2015), 1–53.

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

[23] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, Mohammed G. Khatib, Xubin He, and Michael Factor (Eds.). IEEE Computer Society, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[24] Emiliano Sisinni, Abusayeed Saifullah, Song Han, Ulf Jennehag, and Mikael Gidlund. 2018. Industrial internet of things: Challenges, opportunities, and directions. *IEEE transactions on industrial informatics* 14, 11 (2018), 4724–4734.

[25] TimescaleDB. 2024. *TimescaleDB.* https://www.timescale.com/ (2024, March). [Online]. Available: https://www.timescale.com/.

[26] Nedeljko Vasic, Dejan M. Novakovic, Svetozar Miucin, Dejan Kostic, and Ricardo Bianchini. 2012. DejaVu: accelerating resource allocation in virtualized environments. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, Tim Harris and Michael L. Scott (Eds.). ACM, 423–436. https://doi.org/10.1145/2150976.2151021

[27] 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

[28] Jun Wang, Huafeng Wu, and Ruijun Wang. 2017. A new reliability model in replication-based big data storage systems. *J. Parallel Distributed Comput.* 108 (2017), 14–27. https://doi.org/10.1016/J.JPDC.2017.02.001

[29] Qingsong Wei, Bharadwaj Veeravalli, Bozhao Gong, Lingfang Zeng, and Dan Feng. 2010. CDRM: A Cost-Effective Dynamic Replication Management Scheme for Cloud Storage Cluster. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing, Heraklion, Crete, Greece, 20-24 September, 2010*. IEEE Computer Society, 188–196. https://doi.org/10.1109/CLUSTER.2010.24

Reviewer1

Our proposal has been implemented in an open source system. The code is committed to the repository of Apache IoTDB [N1].

it details

e ref

O1: (1) Our proposal has been implemented in an open source system. The code is committed to the repository of Apache IoTDB [N2, N3, N4, N5]. The corresponding Java classes can be found at A.1 [N1]. (2) Customers we served are enterprises in IoT scenarios with millions of time series sampled at second frequencies. The detailed user scenarios is shown at Sec A.2 [N1].

t true. the ference is key distri- tion, hot titions not nec- ary recent nestamps

O2, D4: The difference is on key distribution, hot partitions are not necessary recent timestamps in OLTP. Thereby OLTP techniques may have to migrate hot partitions more frequently to keep load balance as shown at Sec 1.2.1. In IoT scenarios the hot partitions will shift with time and the customers we served usually set TTL to deleted historial cold partitions automatically as shown at Sec A.2 [N1]. We'll add comparison to this difference at Sec 3.

O3: We will do revamp.

t answer e ques- ns

O4, D6: (1) We setup an 11-node IoTDB cluster in Tencent Cloud, where 8 nodes run DataNodes, 2 nodes run IoT-benchmark and 1 node runs ConfigNode. (2) We generate an average of 60 KB data per write requests and each DataNode processes about 1000 write requests per second on average. (3) We let all DataRegion placement algorithms generate the same number of placement schemes, so that each DataNode holds an average of 6 DataRegions, where 6 is a reasonable amount of DataRegion held by each DataNode we pre-tested before evaluation. We let the primary replica selection algorithms select a primary replica for each DataRegionGroup to host related write requests. (4) We chose this setup because it covers the real customer scenarios, mentioned in Sec A.2 [N1], in both cluter size (8 DataNodes) and write throughput (About 40 million data points per second in maximum). We'll reorganize the beginning of the Sec 6 to form a separate experimental setup subsection. (5) Different sensors generate a variety of loads in real-world environment, so we simulate this situation by having sensors generate different sizes of data in benchmark. We discuss the series partition algorithm at Sec 3. This algorithm evenly distributes the time series to make the load of partitions as even as possible.

mbers

D1, D2: For instance, Kafka was used in AUTO AI's production environment, the IoTDB cluster we deployed had to carry a write load from 64.8 million time series that sampled every three seconds, where the average write throughput was 15.5 million data points per second, and would reach about 30 million data points during peak business hours as shown in Table 3 Sec A.3 [N1]. We'll use this actual data to better motivate our assumption at Sec 1.1.

e actual plication h num- s

D3: The out of order data exists but has little effect on our partition algorithm. This paper [N6] shows that about 7% data is out of order and delay in seconds, which represents that the old and new data partitions are written at the same time only in the seconds before and after switching time partition. We'll add instructions in Sec 1.1.3 and using real datasets with out of order data.

plain efly, and d exp on t-of-order gree

s point not about l applica- n

D5: (1) The scatter width concept is referred from the CopySet paper [4]. And this concept identifies the disaster recovery capability of a specific DataNode, which is orthogonal with the partition. (2) The series partition algorithm mentioned at Sec 2.2 distributes time series evenly to create uniform partitions. (3) We use the distribution of write requests per second to verify the computing load balance and the distribution of disk usage to verify the storage balance at Sec 6.2.

Reviewer2

replace sta tistical ana ysis with a more ex- tensive dis cussion of related wo

O1, D2: We'll replace the statistical analysis with a more extensive discussion of related work.

O2, D4: Our proposal has been implemented in an open source system. The code is committed to the repository of Apache IoTDB [N2, N3, N4, N5]. The corresponding Java classes can be found at A.1 [N1].

Our proposal has been implemented in an open source system. The code is committed to the repository of Apache IoTDB [N1].

O3, D7: For instance, we deployed an IoTDB cluster, which has been runnig for over 4 months, for AUTO AI. This cluster carries a write load from 64.8 million time series where the average write throughput was 15.5 million data points per second as shown in Table 3 Sec A.2 [N1]. Our algorithm makes the cluster have a good write and storage balance wth the coefficiet of variation is less than 4% as shown in Table 4 Sec A.2 [N1]. We'll put these data in Sec 1.

why menti small scale give real customers instead

D1: (1) Series corresponds to storage load balance while time corresponds to computing load balance. (2) We'll add new time series after cluster expansion for the expansion experiment mentioned at Sec 6.2.1 to better demonstrate the scaling power of IoTDB cluster. The supplementary experiment is shown at Sec A.4 [N1].

main dime sion, series dimension

D3: TTL motivates us to propose algorithms mentioned at Sec 3. And the effectiveness is pointed out at Sec 6.2.1 through disk usage distribution.

TTL is not mentioned in the 2nd half of the paper?

D5: The syntactical problems will be solved carefully.

D6: (1) The process of "Expand 4 DataNodes to the IoTDB cluster" means add another 4 DataNodes to the IoTDB cluster. (2) We expected that the disk usage (Fig 17) of all DataNodes can be approximate after the expansion is completed and the TTL time has elapsed. And the RPS (Fig 18) of all DataNodes should be approximate immediately. We'll improve the experimental setting description at Sec 6.2.

Reviewer3

O1: Our proposal has been implemented in an open source system. The code is committed to the repository of Apache IoTDB [N2, N3, N4, N5]. The corresponding Java classes can be found at A.1 [N1].

always me tion review terms like this, same for other points

O2: Note that a typical cluster in most IoTDB customers consists of 3-6 nodes, as presented in Table 3 in Sec A.2 [N1]. Nevertheless, to have broader deployment and evaluation, we will expand the cluster to up to 19 nodes, and report the results in Sec 6 as Sec A.4 described [N1].

Note that a typical cluster in most IoTDB customers consists of 3-6 nodes, as presented in Table ?? in Section ?? [N2]. Nevertheless, to have broader deployment and evaluation, we will expand the cluster to up to 19 nodes, and report the results in Section ?? [N2].

O3: We will improve the presentation.

real world datasets, n simulated

D1: A typical IoTDB cluster deployment consists of 6 DataNodes as shown at Sec A.2 [N1]. We'll redo the Sec 6.2 IoTDB evaluation through writing real datasets by using IoT-benchmark.

D2: The implementation can be founded at Sec A.1 [N1] and the new evaluation details can be founded at Sec A.4 [N1].

[N1] Supplementary. [N2] Data allocation table. https://github.com/apache/iotdb/pull/10648
[N3] Data partition table. https://github.com/apache/iotdb/pull/6199
[N4] Greedy CopySet replication algorithm. https://github.com/apache/iotdb/pull/11572
[N5] Min cost flow distribution algorithm. https://github.com/apache/iotdb/pull/7774
[N6] Yuyuan Kang et al. ICDE 202, IEEE 3340-3352. https://doi.org/10.1109/ICDE53745.2022.00315

A SUPPLEMENTARY

A.1 Implementations in IoTDB

All algorithms we presented in this paper have been fully implemented in Apache IoTDB, an open source time series database system. Including:

- Data allocation table
- Data partition table
- Greedy CopySet replication algorithm
- Min cost flow distribution algorithm

A.2 Customers Workload

As this blog shows, IoTDB has served over 1000 backbone and industrial leading enterprises. We will introduce some successful practices in the following.

A.2.1 Changan Automobile. Changan Automobile is one of the "Big Four" state-owned car manufacturers of China. Changan Automobile built a smart platform for querying massive amounts of connected vehicle data and remote diagnostics based on Apache IoTDB.

A.2.2 Mcc Cisdi. Mcc Cisdi is one of the core subsidiaries of MCC under China Minmetals, the Fortune Global 500. CISDI provides overall solutions for domestic and foreign steel industry customers. Cisdi built the core part of the Water and Soil Cloud Industrial IoT Platform for data development governance base on Apache IoTDB.

A.2.3 AUTO AI. AUTO AI is a provider of technologies and integrated hard/software solutions in the field of intelligent cockpit. It uses Apache IoTDB to manage the driving behavior data of all service vehicles, with a total of 1.6 million online vehicles every day.

The real customers' workload is shown at Table 3.

Taking AUTO AI as an example, we collected its production cluster metric for the past seven days at Table 4. It can be seen that the Write Throughput and the Used Disk Space of this cluster fluctuate very little (the coefficient of variation is less than 5%), so the cluster is relatively balanced.

Table 3: Customer's workload

Enterprise	Changan Automobile	Mcc Cisdi	AUTO AI
Cluster Size (DataNodes)	3	3	6
CPU core (per DataNode)	64	8	16
Memory (per DataNode)	256 GB	32 GB	128 GB
Disk (per DataNode)	10 TB	2 TB	1.2 TB
Time Series	1.2 billion	2 million	64.8 million
Write Frequency	Per 30 s	70% per 1 s 20% per 10 s 10% per 0.1 s	Per 3 s
Write Throughput	avg 6000 k/s max 10000 k/s	avg 3000 k/s	avg 1550 k/s max 3000 k/s
TTL	3 months	7 days	5 days
Time Partition	7 days	7 days	7 days

Table 4: AUTO AI's production cluster

Load Metric	Write Throughput	Used Disk Space
DataNode 1	516113/s	231 GB
DataNode 2	516792/s	235 GB
DataNode 3	521633/s	238 GB
DataNode 4	523417/s	229 GB
DataNode 5	508919/s	244 GB
DataNode 6	510220/s	252 GB
Mean	516182.33	238.17
Standard Deviation	5843.49	8.61
Coefficient of Variation	1.13%	3.62%

A.3 Experiment Setup

We first conducted the simulated evaluation on a personal Mac computer, which is equipped with 10-core M1 Pro chip and 16 GB memory. Then we deployed an IoTDB cluster consisting of 11-nodes on Tencent Cloud for processing the IoTDB evaluation. Each node is equipped with a 4-core CPU, 8 GB memory, 100 GB SSD storage and 1.5 GB ethernet. Of these 11 nodes, 8 are used to deploy DataNodes, 2 are used to deploy IoT-benchmark, and the remaining 1 is used to deploy ConfigNode.

A.3.1 IoT-Benchmark. IoT-benchmark is an open-source benchmark for TSDB in IoT scenario, which we used for generating periodic time series data according to the configuration and send the data batch by batch to IoTDB-Server. We configured IoT-benchmark to simulate sensors with different data types, configured batch size to 100, and assembled the data generated by every 100 sensors into one write request, with the average size of each write request being 60 KB. Finally, each DataNode in IoTDB cluster processes about 1000 write requests per second on average. We approximate the write load limit that the experimental environment can bear by the IoT-benchmark to simulate the high write load in the real production environment.

A.3.2 Alternative Algorithms. We implemented the following DataRegion placement algorithms to compare with our GCR algorithm. The first one is a Greedy algorithm which always place the new DataRegion to the DataNode with the smallest number of DataRegion since it's a intuitive solution. We then implemented the Copyset [4] and Tiered Replication [3] algorithms. Because the optimization of GCR algorithm in disaster recovery capability is inspired by these two papers. For all DataRegion placement algorithms, we have them generate the same number of DataRegion placement schemes so that each DataNode ends up holding an average of 6 DataRegions, where the load factor 6 is the empirical parameter, we conducted some experiments beforehand to determine the appropriate number of DataRegions hosted by each DataNode. Our implemented source code can be found here.

We implemented the following primary replica selection algorithms to compare with our CFD algorithm. The first one is a Greedy algorithm which always select the DataRegion on the DataNode that holds the fewest number of primary replica currently as the primary one to act as an intuitive solution. The second one is a Random algorithm which select a random DataRegion as the primary one. We want to simulate the situation where the DataRegionGroup elects the primary replica by itself, and each DataRegion has the same weight to be elected as the primary one by using this algorithm. Our implemented source code can be found here.

A.4 Supplementary experiments

We will redo the IoTDB evaluation, which we hope can answer the following questions that reviewers are concerned about:

- R1D6: Un-balanced partitions. We have configured IoT-benchmark to generate sensors with different load. And we will set the same configuration when redo experiments.
- R2D1: Add the comparison of time series dimension. The number of time series will be increased proportionally after the cluster is expanded in expansion experiment.
- R3O2: broader deployment and evaluation. As shown in Table 3, most of our customers we serve can meet their business requirement through deploying an IoTDB cluster with not exceeding six DataNodes. Thus, to better demonstrate the expansion capabilities that benefit from our algorithms implemented in IoTDB. We will set the number of DataNodes to 16 and deploy a 19-nodes IoTDB cluster for expansion experiment.
- R3D1: Using real world dataset. We will use IoT-benchmark to simulate a real workload from one of our customers for both expansion and disaster recovery experiment.

A.4.1 Experiment Setup. To sum up, we will use IoT-benchmark to simulate a customer's real workload and genererate sensors with different load to redo 6.2 IoTDB evaluation. Furthermore, we'll start from 8-DataNodes IoTDB cluster and extends it to 16 DataNodes, exponentially increase the number of time series after expansion for the expansion experiment.