FDBKeeper: Enabling Scalable Coordination Services for Metadata Management using Distributed Key-Value Databases

Jun-Peng Zhu
East China Normal University, China
zjp.dase@stu.ecnu.edu.cn

Lingfeng Zhang
East China Normal University, China
fzhang.chn@outlook.com

Peng Cai
East China Normal University, China
pcai@dase.ecnu.edu.cn

Xuan Zhou
East China Normal University, China
xzhou@dase.ecnu.edu.cn

Peisen Zhao
East China Normal University, China
zps@stu.ecnu.edu.cn

Xue Wang
Moqi Inc
China
xuew@myscale.com

Linpeng Tang
Moqi Inc
China
chnttlp@gmail.com

ABSTRACT

High-reliability distributed coordination services have become an indispensable part of modern large-scale distributed systems. Popular coordination services (e.g., ZooKeeper) adopt a single-writer design to provide a centralized service for managing system metadata, including various configuration information and data catalogs, and to provide distributed synchronization functions. With the continuous increase in metadata size and the scale of distributed systems, these coordination services gradually become performance bottlenecks due to their limitations in capacity, read and write performance, and scalability.

To bridge the gaps, we propose FDBKeeper, a novel solution that enables scalable coordination services on distributed ACID key-value database systems. Our motivation is that transactional key-value stores (i.e., FoundationDB) meet the demands of performance and scalability required by large-scale distributed systems over coordination service. To leverage these advantages, coordination services can be implemented as an upper layer on top of distributed ACID key-value databases. Our experimental results demonstrate that FDBKeeper significantly outperforms ZooKeeper across key metrics. Additionally, FDBKeeper reduces hardware resource costs on average by 33% in the production environment, resulting in substantial monetary cost savings. We have successfully replaced ZooKeeper with FDBKeeper in the production-grade ClickHouse cluster deployment.

PVLDB Reference Format:
Jun-Peng Zhu, Lingfeng Zhang, Peng Cai, Xuan Zhou, Peisen Zhao, Xue Wang, Linpeng Tang. FDBKeeper: Enabling Scalable Coordination Services for Metadata Management using Distributed Key-Value Databases. PVLDB, 18(12): 5004 - 5016, 2025.
doi:10.14778/3750601.3750623

PVLDB Artifact Availability:

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750623

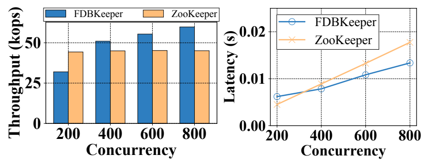


Figure 1: ZooKeeper's scalability issue when processing a high-concurrency workload, where the ratio of reads to writes is 9:1. For experimental settings, see Section §6.2.

The source code, data, and/or other artifacts have been made available at https://github.com/DASE-iDDS/FDBKeeper.

1 INTRODUCTION

Coordination services such as ZooKeeper [35], etcd [31], and Consul [33] have become an indispensable part of modern large-scale distributed systems [23, 26, 34]. These systems implement complex consistency protocols, simplify the implementation of distributed coordination tasks, and ensure consistent and reliable shared storage. They are essential for building reliable distributed systems and reducing system complexity. However, with the rapid increase in data size, the scale of distributed systems is expanding, and the requirements for capacity, read/write performance, and scalability of coordination services are increasing continuously. Consequently, the prevalent single-primary design in coordination services is increasingly becoming a bottleneck that affects the performance of data systems.

Figure 1 demonstrates the ZooKeeper scalability issue under a high-concurrency workload in the ClickHouse cluster. Traditionally, ClickHouse stored only minimal configuration data in ZooKeeper, resulting in low access frequency and stable performance. However, when ClickHouse started storing large volumes of metadata (including user authentication, permissions, resource quotas, and data file metadata) in ZooKeeper [9], access frequency increased significantly, causing performance bottlenecks.

Existing coordination services face significant challenges in managing metadata in big data systems, especially at the exabyte (EB) level [30]. To address this challenge, these data systems have adopted diverse strategies. For example, systems such as Kafka [1] choose to reduce their dependence on coordination services by reconstructing distributed modules using the Raft protocol [42].

Cloud-native systems like Snowflake [28] and ByConity [25] implemented metadata management modules using distributed key-value databases to address the growing demands for read and write performance of metadata. Leveraging distributed transaction databases improves adaptability to the intricate requirements of large-scale data systems, thus improving the performance and efficiency of distributed coordination tasks.

The tight coupling of existing systems with prevalent coordination services (e.g., ZooKeeper) makes replacing the underlying interface challenging, with substantial costs associated with refactoring. Furthermore, mature distributed databases often lack critical capabilities for coordination services, such as ensuring the linearizability of operations and session management. These issues significantly hinder the use of mature database systems as an alternative to coordination services. Consequently, efforts have been made to reimplement well-known coordination services to minimize disruption to legacy system code. However, these efforts continue to face challenges in meeting performance requirements for large-scale deployment [2, 6].

In this paper, we propose FDBKeeper, a scalable distributed coordination service solution based on a distributed ACID key-value database (i.e., FoundationDB). Several alternative distributed database systems exist, including TiDB [34], CockroachDB [47], and YugabyteDB [10]. In contrast to these alternatives, FoundationDB provides a low-level key-value store API (offering primitives such as *Create* and *Set*) that enables the construction of fully customized metadata storage layers, whereas the aforementioned systems implement SQL interfaces that provide higher levels of abstraction. FoundationDB's transaction system is architected specifically to process high volumes of small, rapid transactions, aligning precisely with the characteristics of metadata operations. FoundationDB excels at high-concurrency, small random read and write operations, which constitute the predominant access pattern for metadata management. A primary design objective of FoundationDB is delivering consistently low latency, a critical requirement for metadata operations that frequently function as bottlenecks in query execution pathways and demand minimal response times. FoundationDB provides a strict ACID transaction and strong consistency model, which is critical for managing critical metadata. It should also be particularly noted that systems such as Snowflake also implement metadata management on top of FoundationDB. In comparison, a notable feature of FDBKeeper is its compatibility with the ZooKeeper API, which allows existing systems that rely on ZooKeeper to migrate to FDBKeeper with minimal changes to application logic. As a result, different systems may have differences in the specific implementation and operational practices of metadata management, and the choice of solution can be made based on actual business requirements and technology stack. FDBKeeper facilitates smooth scaling to larger deployments while minimizing disruption to the original system and addressing the scalability issues of existing coordination services. We confront mainly three challenges in the design of the proposed solution. The first challenge is how to efficiently map the hierarchical namespace of the coordination service based on the key-value store. Existing coordination services often employ hierarchical or multi-tier namespaces to organize different functional modules or provide services to multiple tenants. Scalable distributed key-value databases generally do not inherently support this mapping scheme. Although hierarchical namespaces can be emulated by mapping a node UUID to key and its data contents to value, this straightforward approach introduces transaction conflicts (§3.1).

The second challenge is how to implement session management based on key-value databases without introducing code intrusion. Session management is a general feature of coordination services. It is the foundation for building high-level services, such as service discovery and distributed locking schemes [35]. In ZooKeeper, it also helps to identify client faults and remove the ephemeral nodes created by the faulty client. Each client connection has a corresponding session. The lifecycles of all sessions can be easily managed by the single leader, which is one of the servers in the high-available coordination service. We aim to implement session management without modifying the core codes of the underlying key-value databases. There is a lack of straightforward and efficient strategies to manage these complex session schemes.

The third challenge is how to implement the consistency requirements of the coordination service based on distributed transactional key-value databases. Common distributed key-value databases enforce transactions with strict serializability, whereas coordination services typically ensure linearizability. In strictly serializable transactions, the system ensures that concurrent transactions execute in the real-time order they occur. In linearizability, operations issued by a client are executed in the order they occur in real time. Although both guarantee operations are executed in real-time order, the consistency model differs between distributed databases and coordination services in practice. This paper aims to achieve the consistency model of the coordination service in distributed systems using distributed key-value database transactions without compromising system performance.

The main contributions of this paper are summarized as follows.

(1) **Key-value-based hierarchical namespace management.** To mitigate transaction conflicts, this paper initially flattens the hierarchical namespace and subsequently decomposes the nodes into fine-grained keys. Each node is converted to a series of keys prefixed with the node path and suffixed with a detailed attribute name. Based on the mapping scheme, each interface reads and writes specific keys as required, minimizing read and write conflicts, and thereby improving performance. (§3)

(2) **Self-management of client sessions.** Without modifying the database server code, FDBKeeper explores using multiple clients to manage sessions, including lease management, ephemeral data cleanup, and client fault detection and recovery. (§4)

(3) **Consistency model based on ACID transactions.** To guarantee the same consistency semantics in ZooKeeper, FDBKeeper allows multiple client nodes to determine operation order based on transaction completion times (i.e., transaction committed times), while single nodes maintain operation order based on their start time. The consistency of the coordination service is ensured through strictly serializable transactions and local lock schemes implemented on the client nodes. (§5)

(4) **Extensive experimental performance evaluation.** A system called FDBKeeper, which is compatible with the ZooKeeper interface, is implemented based on the open-source FoundationDB database. The performance of the overall scheme is evaluated on this

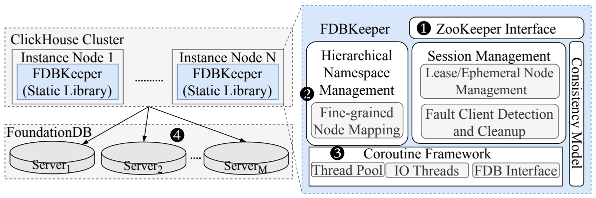


Figure 2: The cluster system overview with FDBKeeper.

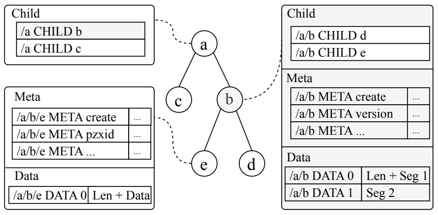


Figure 3: An example of a fine-grained mapping.

system, and its effectiveness is validated by replacing the ZooKeeper component in the ClickHouse cluster across different scenarios. (§6)

2 THE SYSTEM OVERVIEW

The overall architecture of the system using FDBKeeper, as shown in Figure 2, is divided into four layers. It should be noted that FDBKeeper is integrated into each ClickHouse instance as a static library. The first layer (❶) is the ZooKeeper interface layer, which provides a C++ interface compatible with the ZooKeeper request format and interconnects with the lower functional modules.

The second layer (❷) is the core functional module layer of coordination services, consisting of hierarchical namespace management (§3), session management (§4), and the consistency model (§5). The hierarchical namespace management module primarily facilitates mapping between the nodes and the key-value model. The session management module monitors client activity, distinguishes between regular and ephemeral nodes, and removes ephemeral nodes upon detecting client failures. These functions are implemented based on the FDB. The consistency model ensures the coordination service's consistency by enforcing strict serializability (supported by FDB) and local locking schemes (supported by FDBKeeper) while optimizing client performance through batch processing of multiple operations.

The third layer (❸) is the coroutine framework, which provides asynchronous interfaces for the upper layers, such as the FDB interface, delay, and abort transactions. It utilizes coroutines to minimize thread performance overhead, conceal thread scheduling, and simplify programming complexities. FDBKeeper enhances client concurrency and reduces resource usage by utilizing coroutines, which prevent excessive thread consumption in high-concurrency scenarios. The fourth layer (❹) is the database layer, where all persistent data from the coordination service is stored in FDB.

3 KEY-VALUE BASED HIERARCHICAL NAMESPACE MANAGEMENT

In this section, we first analyze the challenges to mapping a node in the hierarchical namespace (§3.1) Then, we present the design of efficient fine-grained mapping schemes (§3.2).

3.1 Problem Analysis

A straightforward approach, similar to metadata management in traditional distributed file systems [41], is to use the generated UUID of the node as the key and the *Meta + Data* of this node as the corresponding value. There are three challenges in implementing this mapping in FDB.

• **Hot Spots.** It is necessary to modify the metadata of the parent node when creating or removing its child nodes. For example, the operations of *Create(/a/b)* and *Remove(/a/c)* under the same parent node (i.e., */a*) introduce transaction conflict since they all modify the metadata of this parent node. Thus, in the case that frequently creating or removing child nodes, it may generate hot spots on the parent node.

• **Query Latency.** Retrieving the nodes in deep paths requires multiple queries. For example, to find the data in */a/b/d*, it is required to first find the UUID of */b* from the value of */a*. Then, the UUID of */d* is found from the data of */b*. Performing multiple reads from the KV store increases query latency.

• **Redundant Metadata.** Most operations involve only a small fraction of the metadata. For example, the operations of *Watch List*, *Watch Get*, and *Watch Exists* monitor only the *version* and *cversion* fields [12]. Storing all metadata and data under a single value causes certain operations to retrieve redundant information from the KV store, leading to additional overhead.

3.2 Fine-grained Mapping

Based on the above considerations, we classify the node metadata and data mapping into the following three key categories. Let us illustrate these three categories with an example as shown in Figure 3.

• The **CHILD** key represents the list of child nodes, consisting of the parent node path name and the node's name. For example, the *CHILD* key of */a/b/c* is */a/b* + *CHILD* + *c*. *List* is a frequently used operation. Under this design, all child nodes of a given parent node can be retrieved by one Range query call in FDB.

• The **META** key stores metadata other than *dataLength*. Fields like *pzxid*, *numChild*, *version*, and *cversion* are stored using a single key. The metadata fields *czxid* and *ctime*, as well as *mzxid* and *mtime*, are each individually combined into one key because they are always modified simultaneously. For example, the creating metadata KV pair for */a/b* is */a/b* + *META* + *CREATE* $\longrightarrow$ <*ctime*, *czxid*>, and *ctime* and *czxid* form a 128-bit integer.

• The **DATA** key represents the data, including *dataLength* and data. For example, the data key-value pair for */a/b* is */a/b* + *DATA* $\longrightarrow$ *dataLength* + *data*. Due to the length limitations of some KV databases, larger data values are split into multiple segments. For example, ZooKeeper enforces a maximum value size of 1 MB, and FDB imposes a 100 KB limit, requiring the implementation of value segmentation in FDBKeeper when the threshold is exceeded. When the value exceeds 100 KB, FDBKeeper splits it into multiple continuous keys and stores them in the FDB, which uses a distributed B-tree storage architecture based on SQLite. B-tree is particularly well-suited for range queries due to its ordered structure and efficient sequential access properties. For example, a 300 KB value is split into multiple consecutive 100 KB KV pairs, i.e., "prefix/seg1": 100 KB value, "prefix/seg2": 100 KB value, and "prefix/seg3": 100 KB value. Each segment key is stored consecutively, with no other keys in

Algorithm 1: Create Lease

```
Data: Expiration Time: $outdate\_interval$
Result: Client ID: $client\_id$, Lease Key: $lease\_key$
1  Transaction:
2      Loop:
3          $client\_id \leftarrow random\ unsigned\ int64$ ;
4          $exist \leftarrow$ Get($CLIENT + client\_id$) ;
5          if $exist$ then continue;
6          Set($CLIENT + client\_id$) ;
7          break;
8      $outdate \leftarrow nowtimestamp + outdate\_interval$ ;
9      $lease\_key \leftarrow LEASE + outedate + client\_id$ ;
10     Set($lease\_key, NULL$) ;
```

Algorithm 2: Update Lease

```
Data: Lease Refresh Time: $refresh\_interval$, Expiration
      Time: $outdate\_interval$, Client ID: $client\_id$, Lease
      Key: $lease\_key$
Result: Lease Key: $lease\_key$
1  Loop:
2      Transaction:
3          Wait $refresh\_interval$ ;
4          $exists \leftarrow$ Get($lease\_key$) ;
5          if $not\ exist$ then break;
6          Clear ($lease\_key$) ;
7          $outdate \leftarrow nowtimestamp + outdate\_interval$ ;
8          $lease\_key \leftarrow LEASE + outedate + client\_id$ ;
9          Set($lease\_key, NULL$) ;
```

between. When querying, the range uses the "prefix/seg*" pattern. These segments are stored consecutively in the B-tree, preventing significant performance degradation. On the other hand, FDBKeeper ensures the atomicity of this multi-segment write through one FDB transaction, ensuring no correctness issues.

The constants in key names such as *CHILD*, *META*, and *DATA* are enumerated and represented by a byte. The operation for each node is implemented as a transaction in FDB, where all FDB read and write operations are performed within a transaction

This fine-grained mapping strategy does not fully address the issue of hot spots in the parent node when nodes are created or removed simultaneously under the same parent. In the node metadata, the *pzxid* field records the transaction ID which indicates the last modified timestamp, while *cversion* and *numChild* are counters. FDBKeeper uses atomic operations in FDB [51], capable of pushing operations like ± 1 (i.e., counter plus or minus one) and updating on *pzxid* down to the storage layer to further alleviate conflicts in the transaction layer.

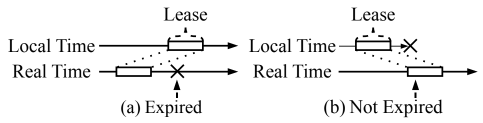


Figure 4: An illustration of accuracy issues in local time.

4 SESSION MANAGEMENT

4.1 Fault Client Detection

Leases are widely used in distributed systems for node failure detection. In typical coordination services, a single primary node monitors the client's heartbeat, detects failures, and clears the faulty client. However, this approach requires modifications of the FDB servers to implement lease management for FDBKeeper. Instead, we implement the logic of the lease on the client-side based on the FDB transaction. Each client creates a lease key in FDB at startup and then periodically updates the lease key. A lease key consists of three parts: a prefix that indicates that it is a lease key, a 64-bit expiration timestamp, and a 64-bit client ID.

Algorithm 1 outlines the procedure for creating a lease using an FDB transaction. In creating a lease transaction, a 64-bit ID for the client is first generated randomly locally. The client key is used in the database to record already occupied IDs to ensure uniqueness. If an ID for a client is found to be occupied, a new one is generated until there are no conflicts. Then, the expiration timestamp is calculated from the local current timestamp and the expiration time, and the expiration timestamp and the client ID are combined as the lease key. Finally, the new client key and the lease key are submitted. After a lease is created, the client periodically updates the lease. As shown in Algorithm 2, the old lease key's existence is first checked in the updating lease transaction. If it exists, the old lease key is deleted, and a new lease key is created to replace it. If the lease does not exist, it is deemed invalid, and the client is subsequently terminated.

To quickly find expired leases, the lease key is prefixed with the big-endian order of the expiration time, allowing the lease keys to be incrementally sorted by expiration time. When querying expired leases, it only needs to search for keys with strings smaller than *LEASE + big-endian order of the current timestamp* to find all expired client IDs. Such queries are efficient in FDB primarily because: (1) FDB employs a distributed B-tree storage architecture based on SQLite, where the ordered and balanced properties of B-trees enable range query operations with O(log N + K) time complexity, where N is the total number of records and K is the result set size; (2) our lease key design ensures leases are stored in expiration time order within the B-tree, reducing expired lease retrieval to a single range scan operation; (3) the ordered nature of B-tree internal nodes guarantees efficient sequential access, with leases of adjacent expiration times exhibiting good storage locality, allowing range queries to fully leverage sequential I/O advantages at the storage layer while avoiding random access overhead. Furthermore, keys serve as a natural index, eliminating the need for additional secondary index structures and thereby simplifying system design while enhancing query efficiency. However, there is a drawback in using the client's local time to determine expired leases. When the local time is later than the true time in Figure 4 (a), the client cannot obtain a lease, and when the local time is earlier than the true time in Figure 4 (b), it

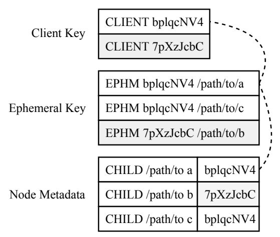


Figure 5: Ephemeral node index.

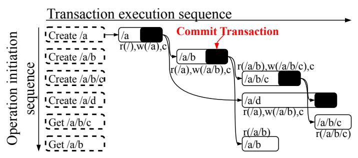


Figure 6: An illustration of the FIFO client order in FDBKeeper.

is impossible to correctly determine the activity of the client. Therefore, our policy is based on synchronizing individual client time and requires the lease timeout to be greater than 2× the maximum time error. Considering that the clients of the coordination service are usually distributed systems, it is relatively easy to require time synchronization. Common time synchronization protocols, such as NTP, can keep time errors within milliseconds, which is acceptable for coordination services.

4.2 Ephemeral Node Management

Coordination services require distinguishing between ephemeral nodes and persistent nodes. The ephemeral node lifecycle is bound to the client and is automatically deleted upon client termination or failure. The persistent nodes persist even after the client terminates. The scenarios that must be considered for the management of ephemeral data include creating ephemeral nodes, obtaining node types (e.g., *Get* and *List* operations), and batch removing ephemeral nodes belonging to a certain client ID. As shown in Figure 5, to meet the requirements of these three scenarios, ephemeral data management records the client ID in the node metadata and adds a set of indexes to quickly query the ephemeral nodes associated with a single client ID. When removing ephemeral nodes in batches, it scans the ephemeral node keys prefixed with *EPHM+client_id* to obtain the paths of all the ephemeral nodes and then removes the metadata keys and data keys of those nodes.

4.3 Fault Client Cleanup

A client is elected as the leader to clean other faulty clients whose leases have expired. When a client is started, the cleaning election is also started at fixed intervals to elect a client as the cleaning leader. To mitigate election conflicts among different clients, the interval between each election is extended randomly by a certain amount of time. As shown in Algorithm 3, the expiration time of the current cleaning leader is obtained in the key CLEANER (Step 4). If the current cleaning leader has not expired (Step 5), the module of cleaning expired leases attempts to elect a client as cleaner after waiting for *cleaner_interval*. To prevent high conflict, a random period is added between each election (Step 2). The election process sets the expiration time of the current cleaner in the cleaner key if the cleaner has expired (Step 7). Because local time may be out of synchronization, which is similar to the lease implementation in Section §4.1, the election algorithm cannot guarantee that only one cleaner is running on the system. Therefore, this issue needs to be considered during the cleanup process.

During the lease cleaning process, the expired lease key is first queried in a transaction to identify potentially failing clients. The ephemeral nodes belonging to these clients are then located and

Algorithm 3: Cleanup Expired Lease

```
Data: Cleaning Interval: $cleaner\_interval$, The upper limit
      of the number of expired leases: $batch\_limit$
1  Loop:
2      Wait $cleaner\_interval \pm random$;
3      Transaction:
4          $outdate \leftarrow$ Get($CLEANER$);
5          if The previous cleaner is active then continue;
6          $outdate \leftarrow now\ timestamp + outdate\_interval$ ;
7          Set($CLEANER$, $outdate$) ;
8      Transaction:
9          $outdate\_leases \leftarrow$ Get expired leases;
10     Loop:
11         Transaction:
12             $n \leftarrow 0$;
13             while $n < batch\_limit$ do
14                 $lease\_key \leftarrow First(outdate\_leases)$;
15                 if $lease\_key$ not exist then break;
16                 update $lease\_key$, Set the expiration time 0;
17                 $paths \leftarrow$ Get ephemeral nodes;
18                 if $paths$ not NULL then
19                     $n \leftarrow len(paths) + n$;
20                     foreach $paths$ do Remove node;
21                 else
22                     $Pop(outdate\_leases)$;
23                     Clear($lease\_key$)
```

removed. The cleanup is divided into multiple transactions, and each transaction cleans up to the *batch_limit* of ephemeral nodes, typically set to 100, which corresponds to approximately 15,000 keys. The purpose of limiting the number of cleaned keys is to prevent long transactions and to help quickly identify conflicting cleaners. When the cleanup task starts to clean up a client, it first queries whether the lease key exists or not; if the lease key does not exist, there are two possible scenarios: there is a delay in the client, or the client has been deleted by other cleaners, i.e., there is more than one cleaner in the system. For both cases, it can be assumed that the cleaner has failed, so the cleanup process ends, and the next election is awaited. If the lease key exists, its expiration time is reset to zero. Ephemeral nodes are not directly removed because the cleanup task is split into multiple transactions. If a failure occurs during the cleaning process, subsequent cleaners can still detect uncleaned leases. Then, the cleaner queries and removes the ephemeral nodes associated with the client. If no ephemeral nodes are found, the cleanup is considered complete.

It is essential to note that by appropriately configuring the CLEANER election interval and the expiration period of ephemeral nodes, where the election interval is typically shorter than the expiration period, the system establishes a fundamental fault-tolerance window for handling CLEANER failures. In addition, the cleanup process is designed to be idempotent and batched. Even if repeated CLEANER failures occur, subsequent CLEANERs can detect and compensate for any unfinished cleanup tasks from previous failed CLEANERs. In conclusion, these mechanisms effectively prevent the issue of the system being unable to efficiently clean fault clients due to the prolonged absence of an effective cleaner. In Moqi's production deployment, we set the CLEANER election interval to 2 seconds and the expiration period of ephemeral nodes (i.e., the Zookeeper connection timeout) to 4 seconds, achieving robust fault tolerance and system stability.

5 CONSISTENCY GUARANTEES

5.1 Problem Analysis

ZooKeeper offers two basic ordering guarantees [35]. The first is to guarantee linearizable writes issued by all clients. These concurrent writes are serialized by the leader node under the Zab protocol [36]. ZooKeeper guarantees linearizability across all clients once requests arrive at ZooKeeper. However, in practice, the time it takes for requests from the client to reach ZooKeeper is not guaranteed. In ZooKeeper, each client can have multiple outstanding operations. The second guarantee is the FIFO client order, where all the operations of a given client are executed according to the sending order via a single TCP connection between the client and the ZooKeeper server. The FIFO client orders can ensure the correct execution of *Create(/a)*, *Create(/a/b)* and *Create(/a/b/c)* sequentially called by a client, although the three requests are sent asynchronously.

FDB does not provide a guarantee of the FIFO client order. In FDBKeeper, each ZooKeeper operation is implemented by the FDB transaction. FDB currently does not support transactions exceeding five seconds [17]. When ZooKeeper operations are implemented as FDB transactions in our production environment, they exclusively involve metadata operations. We have not observed any performance constraints related to the five-second transaction duration. When an FDB client sends three transactions in the order that is *Create(/a)*, *Create(/a/b)* and *Create(/a/b/c)*, FDB cannot guarantee the execution and commit of the three transactions in the sending order. The transaction *Create(/a/b)* may be scheduled to execute before the transaction *Create(/a)* is committed. In this case, *Create(/a/b)* failed because the parent node (i.e., */a*) of */a/b* has not been created by *Create(/a)*. In a word, FDB cannot ensure all three *Create* transactions sent by a client are successfully executed every time, as the transaction is not executed one after the other in the sending order. On the other hand, ZooKeeper ensures the three *Create* requests sent by a client are sequentially applied to the state of the hierarchical namespace. From ZooKeeper's perspective, the time ZooKeeper sends a request should align with when FDB begins a transaction. That is, all three requests are always executed successfully in ZooKeeper.

For multiple clients or sessions, ZooKeeper implements update linearizability, which is *naturally* guaranteed by strict serializability [7, 51] in FDB. Each update request corresponds to an FDB transaction. In FDBKeeper, it is necessary to align ZooKeeper's receive order of operations with FDB's transaction commit time. FDB uses a single sequencer to assign the commit number to each committed transaction. Transactions are serialized according to their commit numbers.

5.2 FIFO client Order in FDBKeeper

To support the FIFO client order, FDBKeeper uses three types of locks on the client side to control how the transactions sent by a client are scheduled on the server side.

Object read lock. The granularity of a lock is a node in the hierarchical namespace. A read lock is a shared lock. The operations sent by a client are allowed to start the corresponding transactions concurrently if these operations have just the same read locks.

Object write lock. The granularity is the same as the read lock. A write lock is exclusive. If several operations sent by a client need to request the same write locks, FDBKeeper allows only one operation that holds the write lock to start a transaction. Read and write locks ensure the FIFO client order of accessing a single object.

Commit lock. It is used to control the order of transaction commits. In an FDBKeeper client, all its initiated transactions are required to request the commit lock. Essentially, the commit lock can be regarded as a FIFO queue. The commit lock ensures that all write operations in this FDBKeeper client are committed in the order in which they are sent. Transactions that have no write lock conflict and just wait for the same commit lock can be concurrently executed by the FDB server-side. The FIFO client order of write operations is guaranteed by the locking scheme of each client and the serializability in the FDB server.

Figure 6 illustrates how an FDBKeeper client uses the locking scheme to control the start and commit of the transaction. The vertical axis represents the order in which local operations are sent, and the horizontal axis represents the order in which corresponding transactions are started and committed. The figure uses *r(x)* for the read lock, *w(x)* for the write lock, and *c* for the commit lock. In the transaction legend, the white portion represents the transaction execution process, while the black portion represents the commit process. The operations of *Create(/a)*, *Create(/a/b)*, *Create(/a/b/c)*, and *Get(/a/b/c)* are executed *sequentially* in terms of their conflicted read and write locks. The operations *Get(/a/b)* and *Create(/a/b/c)* share read locks, allowing them to execute *simultaneously*. Due to the commit lock, all operations are committed in the sending order. In particular, the three types of locks in FDBKeeper are essentially client-scoped, in-memory locks that exist only within each client process's local memory, designed to maintain FIFO ordering of operations within a single client session rather than serving as traditional distributed locks that require cross-client coordination. Each client maintains its own independent set of locks, and the failure of one client does not impact the lock state or operation execution of other clients. When a client fails, these in-memory locks are automatically cleaned up as the process terminates, requiring no explicit recovery mechanism because these locks exist only in the failing client's memory space, and no other clients depend on or are blocked by these locks.

In an FDBKeeper client, each operation adopts a strict two-phase locking (2PL) policy. To avoid deadlock, all operations in wait form a logically directed acyclic graph (DAG) (i.e., precedence graph or dependency graph [46]). The DAG represents the dependency relationship among operations with various locking requests. Given the input parameter in an operation (e.g., */a/b* as the parameter for *Create(/a/b)*), it is straightforward to calculate all the required read and write locks. Thus, before operating on an object, the FDBKeeper client calculates all necessary locks to be acquired and then adds the operation to the DAG. The DAG is sorted according to the lock type of each operation, and new operations are appended to the end of the DAG. If an operation has no dependency on any other operations, it is executed immediately. After its execution is completed, the operation is removed from the DAG, allowing the next operation without dependency to proceed. FDBKeeper can guarantee the linearizability of operations by combining the strict serializability of FDB with client-side locking schemes. We provide formal proof of linearizability in our technical report [20].

It should be noted that the DAG we use primarily ensures the FIFO order of operations within the same client session, rather than serving as a global order mechanism. This means that the complexity of the DAG primarily depends on the number of concurrent operations within a single session, rather than on the overall workload of the cluster. In typical coordination service usage scenarios, the concurrent operations within a single client session are usually limited because most applications follow the request-response interaction pattern. Therefore, the number of nodes and the complexity of edges in the DAG are controllable in practical scenarios. Additionally, in the large-scale application of the Moqi Inc. business, we have not encountered any bottleneck issues with the DAG.

It is important to emphasize that the DAG mechanism in FDBKeeper is strictly used to maintain the execution order of operations within a single client session rather than creating dependency-waiting relationships between operations. When a client-initiated operation involves a non-existent parent node (such as *Create(/a/b)* when /a does not exist), FDBKeeper follows standard ZooKeeper semantics by immediately returning a *NoNode* error to the client, rather than having the operation wait for parent node creation. The DAG's role is solely to ensure that operations within the same session are processed and committed in the order sent by the client, rather than creating waiting mechanisms based on the logical dependencies of node paths. This design avoids operation blocking due to path dependencies, ensuring that each operation receives a definitive result (success or failure) immediately based on the current system state, thereby guaranteeing system responsiveness and predictability.

5.3 Batch Processing Optimization

We ensure coordination service consistency by controlling the started and committed times of transactions. However, waiting for transactions to complete sequentially significantly increases the overall operation latency. Therefore, in addition to locks, the consistency module merges and reduces the number of waiting operations. Operation merging occurs when an operation is added and removed from the DAG. For example, when a *Get* operation follows a *Create* operation, they are merged into a single *Create* operation transaction. The results of multiple operations are then returned immediately after the create operation transaction is completed. There are two merge strategies for object read/write locks and transaction (commit) locks:

(1) Merge transactions based on the number of pending write operations. Configure *N* to merge multiple write operations into a single transaction, committing them all at once. As shown in Figure 7, the operations of *Create(/a)*, *Create(/a/b)*, *Create(/a/b/c)*, and *Create(/a/d)* are merged into a single transaction. These four

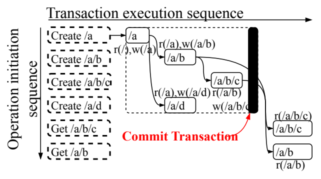


Figure 7: An illustration of merging multiple writes in a single transaction.

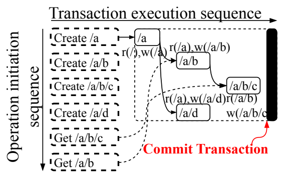


Figure 8: An example of heuristic rules that combine multiple operations.

operations are atomically written to the database in a single transaction. Merging multiple write operations reduces the number of transactions, thus decreasing the overall average latency.

(2) For object read and write locks, heuristic rules combine two operations into one. As shown in Figure 8, the *Create* and *Get* operations are combined into a single *CreateOrGet* operation, which first attempts to *Get* the object, *Create* object it if it does not exist, and returns the results of both *Create* and *Get*. Although there are many potential rules, the performance gains from this merging strategy are limited because multiple operations on a single object are not typically initiated simultaneously. Currently, only two heuristic rules are added for specific scenarios: *Create+Get* and *Create+Create*. In scenarios with many concurrent cascading operations, root nodes are frequently created. These creation operations are executed sequentially, so when the same objects are created simultaneously, they are merged into a single *Create* operation. The first *Create* operation is executed normally, and subsequent *Create* operations fail immediately without a transaction. FDBKeeper preserves linearizability guarantees despite implementing batch processing optimizations. We also provide formal proof in our technical report [20].

6 EXPERIMENTAL EVALUATION

6.1 Experimental Setup

6.1.1 System. The experiments utilize ZooKeeper 3.9.1 and OpenJDK 1.8.0392. The cluster consists of one leader node, two follower nodes, and two observer nodes. Each ZooKeeper node sets *maxClientCnxns* to 0 to disable client connection limits and increase the maximum heap memory of the JVM to 15GB (i.e., -Xmx = 15G) to prevent memory insufficiency.

The experimental evaluation utilizes FDB version 7.1.27 in this paper. FDB requires manual configuration of the process count and roles, with each process utilizing up to one CPU core. In this experiment, a node acts as a coordinator, configured with 1 coordinator process, 1 transaction process, and 4 storage processes. The other four nodes are configured with 1 transaction process, 6 storage processes, and 2 stateless processes each. In total, there are 1 coordinator process, 5 transaction processes, 28 storage processes, and 8 stateless processes. FDB utilizes a 3-copy SSD engine (configured as a new triple SSD).

6.1.2 Environment. The experiments run on KVM virtualized cloud hosts. A total of 9 to 11 cloud hosts are allocated, with 4 designated for clients and the remaining hosts allocated to servers. Each cloud host features a 4-core Intel Broadwell CPU running at 2.2GHz, along with 16GB of RAM. The servers utilize SATA SSDs for storing experimental data, boasting a 4k random read performance of 187k IOPS and a 4k random write performance of 24k IOPS. The cloud hosts are connected through a 10 Gbps Ethernet network. These cloud hosts are running 64-bit Ubuntu 22.04.

6.1.3 Benchmarks. This paper rewrites and extends the ClickHouse keeper-bench [44] to support experiments that simulate high-concurrency scenarios. This enhancement enables handling richer workloads and surpassing the single-machine limit by simulating over 1, 000 clients concurrently on multiple hosts. The experiment is orchestrated using a series of Ansible and Python scripts. Initially, the keeper-bench is built and deployed on 4 cloud hosts as clients along with the workload profile. Subsequently, servers 1 to 9 are initialized: the original services are stopped, experimental hard disks are cleared, the ext4 file system is formatted, and a ZooKeeper or FDB cluster is deployed. Except for the scalability experiment, all other experiments utilized five hosts as servers.

Multiple keeper-bench processes are evenly distributed across the client hosts. The experiment adjusted the number of keeper-bench processes based on the concurrency level, with each keeper-bench process simulating 100 clients. Furthermore, every 10 client coroutine shares a single thread. The keeper-bench utilizes a YAML file to define workloads, each comprising two primary phases:

• **Data initialization phase**. During the data initialization phase, a hierarchical namespace definition generates data of arbitrary depth with random names and content. In this phase, the keeper-bench processes concurrently execute *Create* requests to FDBKeeper or ZooKeeper.

• **Testing phase**. This phase, defined by a list, supports operations like *Create*, *Set*, *Get*, and others at any path, with the capability to specify weights. One or more keeper-bench processes are executed in parallel, each operating independently without mutual interference.

To verify the performance of FDBKeeper under different workloads, we also utilized five YCSB [21, 32] workloads: A (w:50%, r:50%), B (w:5%, r:95%), C (read-only), D (read latest, w:5%, r:95%), and F (w:25%, r:75%). We do not use YCSB-E primarily because ZooKeeper does not support range queries. All experiments are conducted with a degree of concurrency of 2000.

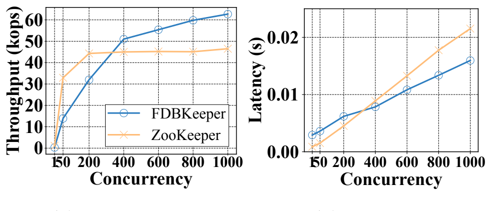
6.1.4 Evaluation Metrics. This section employs various methods to collect different monitoring indicators as evaluation metrics. It primarily covers the following aspects:

• The node exporter [4] is used to collect hardware performance metrics, including CPU, memory, hard disk, and network usage. It calculates maximum CPU memory overhead, disk I/O performance, disk capacity, and network utilization.

• Enable ZooKeeper's built-in Prometheus [5] metric provider on each server to collect JVM memory metrics, ensuring memory is not a performance bottleneck for ZooKeeper.

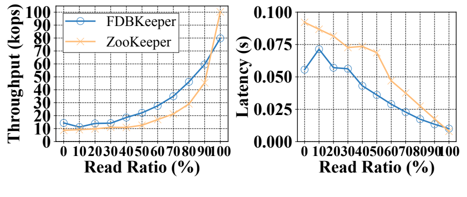
• The *status json* command in *fdbcli* collects FDB cluster metrics on a single client, extracts and converts them into Prometheus metrics, and pushes them to the *Pushgateway* [8] every second. The collected metrics include process CPU usage, transaction phase latency, transaction rate, collision rate, and QoS automatic performance limits, which help identify FDB performance bottlenecks.

• Each keeper-bench process records the number of waits, completions, and errors for different operations, along with total latency. It calculates the 10%-90% and 99%-99.9% percentile latencies, then extracts and converts these metrics to Prometheus metrics at 1-second intervals to push to the *Pushgateway*.



(a) Throughput (b) Latency

Figure 9: Performance when reads-to-writes is 9:1.



(a) Throughput (b) Latency

Figure 10: Performance on the different read ratios.

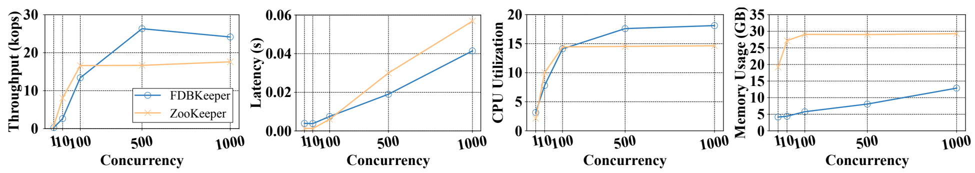
Finally, Prometheus captured and calculated all monitoring indicators, while Grafana [3] further aggregated and visualized the experimental results. The experimental evaluation code is available at https://github.com/DASE-iDDS/FDBKeeper-Evaluation.

6.2 Performance Comparison

We evaluated the throughput and latency of FDBKeeper and ZooKeeper at different levels of concurrency, as illustrated in Figure 9. The experimental evaluation utilizes a common ZooKeeper workload [11, 35], where the ratio of reads to writes is 9:1. Specifically, 81% of the operations are *Get*, 9% are *List*, 4% are *Set*, 3% are *Create*, and 3% are *Remove*.

With increasing concurrency, the throughput of both FDBKeeper and ZooKeeper also increases. For concurrency levels below 400, FDBKeeper exhibits lower throughput compared to ZooKeeper. Moreover, FDBKeeper's single operation has a minimum latency of about 3 ms, while ZooKeeper achieves approximately 1 ms. Both read-only and write transactions of FDB need to communicate with multiple nodes, which leads to multiple network communications and limits the minimum latency. In contrast, ZooKeeper has a maximum of 2 RPC operations. When the concurrency reaches 200, ZooKeeper's throughput no longer increases significantly due to the single-point bottleneck. On the other hand, FDBKeeper can fully utilize the advantages of FDB to achieve higher throughput under high concurrency. At 800 concurrency, the throughput of FDBKeeper reaches about 60 kops. These results demonstrate that, under a typical ZooKeeper workload, the throughput of FDBKeeper increases gradually with increasing concurrency, whereas ZooKeeper reaches its throughput ceiling. However, in low-concurrency scenarios, ZooKeeper is more responsive.

There is a significant disparity in the time consumed between read and write operations. The read-write ratio of the workload has a significant influence on the throughput of the coordination service. We evaluated the performance of FDBKeeper and ZooKeeper in different read/write scenarios, as shown in Figure 10. The evaluations encompass workloads spanning from 0% read operations to 100% read operations. In read operations, 90% are *Get* operations and 10% are *List* operations. In write operations, operations are



(a) Throughput (b) Latency (c) CPU Utilization (d) Memory Usage

Figure 11: Performance comparison on the ClickHouse workload.

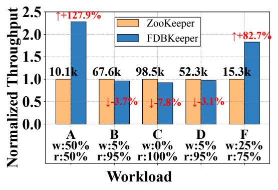


Figure 12: Throughput performance on YCSB.

divided evenly: 33.3% for *Set*, 33.3% for *Create*, and 33.3% for *Remove*. The experiment simulates 800 concurrent clients to evaluate the maximum throughput of the system.

As the percentage of read requests increases, the throughput of both FDBKeeper and ZooKeeper tends to increase. In the read-only scenario, that is, at 100% read, ZooKeeper achieves approximately 25% higher throughput compared to FDBKeeper. In the write-only scenario, that is, at 0% read, the throughput of FDBKeeper is about 65% higher than ZooKeeper. The experimental results demonstrate that ZooKeeper exhibits superior read performance, making it suitable for scenarios where read operations significantly outnumber write operations, such as in configuration management. FDBKeeper is better suited for scenarios with write requirements, such as log synchronization. This result arises from FDBKeeper's storage system, FDB, which shards transaction processing and storage across multiple physical nodes, leading to enhanced write performance. In contrast, ZooKeeper processes requests exclusively through the primary node. However, each replica node of ZooKeeper is a full replica, and in a 5-node deployment, all 5 ZooKeeper instances provide read-only services. FDB, on the other hand, has a maximum of 3 replicas, which limits performance FDBKeeper in read-only scenarios.

The goal of FDBKeeper is to replace the ZooKeeper service in ClickHouse clusters. ClickHouse, which uses ZooKeeper as its metadata repository, handles more write requests compared to a typical coordination service, with a read/write ratio of approximately 6:3 [44]. This experiment evaluates the performance of FDBKeeper in ClickHouse by simulating this workload. The percentages of operations in the experiment were 60% *Get*, 6% *List*, 11% *Set*, 11% *Create*, and 11% *Remove*. The experimental results are illustrated in Figure 11. FDBKeeper can handle greater concurrency on the same hardware, offering better latency and throughput in high-concurrency scenarios. This is primarily because (1) under high concurrency, the design of FDBKeeper leverages the FDB capability to execute transactions concurrently, enhancing its performance. (2) Under low concurrency, although the commit time of an FDB transaction exceeds the execution time of ZooKeeper, FDBKeeper also maximizes concurrency while ensuring correct FIFO order through a fine-grained lock design. The overall CPU usage of ZooKeeper reaches its maximum limit under increasing concurrency, primarily due to a single-point bottleneck, where all operations are executed sequentially. The memory footprint of FDBKeeper is much smaller than that of ZooKeeper. It should be noted that the memory reported in our experiments refers to the memory usage of the FDB servers. This experiment demonstrates that FDBKeeper can effectively replace ZooKeeper and support larger-scale ClickHouse cluster deployments.

To verify the performance of FDBKeeper under different workloads, we conducted performance comparison experiments using five YCSB workloads. Workloads A, B, C, D, and F employ the default Zipfian access pattern with a Zipfian value of 0.99. The experimental setup remains the same as described above. Figure 12 demonstrates the experimental results. It is crucial to note that YCSB-B (95% read), YCSB-C (read-only), and YCSB-D (95% read) are workloads with more reads than writes. On the read-only YCSB-C workload, FDBKeeper's throughput has decreased by as much as 7.8%. For YCSB-B and YCSB-D, FDBKeeper demonstrates slightly lower throughput than ZooKeeper. When the proportion of reads is higher, the experimental results align with those discussed in Figure 10. Our evaluation shows that FDBKeeper outperforms ZooKeeper for both YCSB-A and YCSB-F workloads. The throughput performance has increased by 127.9% and 82.7%, respectively, as shown in Figure 12. The reasons for its performance differences are also consistent with those shown in Figure 10.

6.3 Scalability

Scalability is a challenge for ZooKeeper, primarily due to its single-primary architecture, which limits horizontal scaling. We evaluated the impact of FDBKeeper and ZooKeeper on the throughput, latency, and resource consumption of the entire cluster as the number of cluster nodes increased. The number of nodes and the corresponding relationship between the configurations in the experiment are shown in Table 1. FDB utilizes a maximum of one CPU core per process, resulting in multiple processes launched on a single node during deployment. However, the first node has two fewer storage processes because it starts the coordinator process. All other nodes deploy 1 transaction, 6 storage, and 2 stateless processes. In a single-node deployment, the cluster utilizes a single replica; for deployments with three or more nodes, FDB employs three replicas. ZooKeeper restricts voting to the first 3 nodes, and the remaining nodes are configured in the observer role and do not participate in voting. In this section, we use the ClickHouse workload [44] to evaluate the scalability.

In a single-node deployment, ZooKeeper shows double the throughput of FDBKeeper, and its standalone performance significantly surpasses that of FDBKeeper as shown in Figure 13. In cluster deployments with more than 3 nodes, ZooKeeper faces limitations due to a single-point bottleneck. In scenarios dominated by write operations, enhancing overall system throughput by adding observer nodes proves challenging. FDBKeeper demonstrates linear throughput growth as the number of nodes increases, offering superior performance in large-scale deployments. Regarding resource usage, ZooKeeper's CPU and memory resources increase linearly with the number of nodes. The memory footprint of FDBKeeper remains stable because, in larger FDB deployments, the increase

Table 1: Scalability experiment configurations.

	FDB				ZooKeeper		
Nodes	Replicas	Transaction	Storage	Stateless	Leader	Follower	Observer
1	1	1	4	2	1	0	0
3	3	3	16	6	1	2	0
5	3	5	28	10	1	2	2
7	3	7	40	14	1	2	4
9	3	9	52	18	1	2	6
11	3	11	64	22	1	2	8

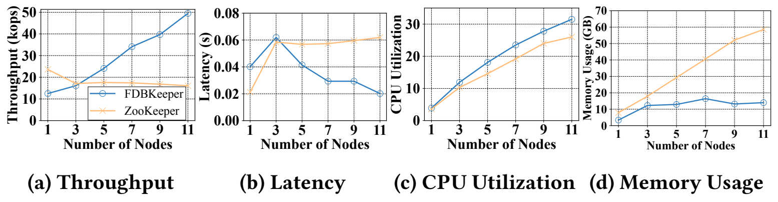


Figure 13: Performance comparison of scalability.

in the number of processes reduces the workload (i.e., shards) handled by each, maintaining a consistent memory footprint. This is primarily because the design of FDBKeeper fully leverages FDB's scalability, including its ability to execute transactions concurrently and partition-based storage. As a result, it is worth noting that FDBKeeper inherits the near-linear scalability of FDB [19].

6.4 Bulk Loading

This experiment compares the time and resource consumption of FDBKeeper and ZooKeeper during bulk loading operations. The experiment simulated the load process for 800 clients simultaneously, bulk loading data sizes of 50, 100, 200, 300, 400, and 500 MB, respectively. These data are generated by keeper-bench and each data entry per node is 128 bytes.

Figure 14 (a) illustrates that the bulk loading time for both FDBKeeper and ZooKeeper increases linearly with the size of the data, but FDBKeeper exhibits a relatively lower growth rate. For data sizes exceeding 500 MB, FDBKeeper demonstrates approximately 14% faster performance than ZooKeeper. The CPU and memory usage represent the peak values recorded during a single experiment. In terms of CPU utilization, FDBKeeper utilizes nearly all available CPU resources, whereas ZooKeeper utilizes approximately 15 cores. This is primarily because FDB distributes writes across multiple nodes, while ZooKeeper primarily performs writes on a single primary node. FDBKeeper consumes less memory than ZooKeeper, as shown in Figure 14 (c). Based on the experimental findings above, FDBKeeper can utilize additional CPU resources to achieve faster speeds during bulk loading, effectively overcoming the limitations of single-point bottlenecks. However, FDBKeeper exhibits higher write amplification and occupies 3× more storage space than ZooKeeper. This is primarily due to (1) path redundancy introduced by flattening the tree structure of znode into key-value pairs, requiring full path storage rather than node names alone; (2) FDB's write operations generating substantial amounts of WALs and incurring the cost of maintaining the B-tree, which directly leads to write amplification in FDBKeeper. In contrast, ZooKeeper temporarily reduced storage usage during bulk loading by disabling the snapshot mechanism and retaining only the log persistence process. However, these storage overheads provide a significant improvement in write throughput, overcoming the bottleneck of the single-writer architecture through multi-node parallel processing and leading to notable performance gains in high-concurrency scenarios. More importantly, FDBKeeper's distributed architecture, based on FDB, supports horizontal scalability of storage and is not limited by the capacity of a single node, which is crucial for large-scale metadata management scenarios. In contrast, ZooKeeper is limited by the memory and disk capacity of a single node.

It is also worth noting that, although FDBKeeper exhibits a significant increase in storage usage, the bulk loading completion time does not increase. The main reason lies in the architectural differences between the two systems. Zookeeper processes write requests serially through a single leader node, so all write operations must be committed sequentially, which limits overall throughput. In contrast, FDBKeeper, built on FDB, can distribute write operations concurrently across multiple storage nodes, fully leveraging the parallelism of the distributed storage system. As a result, it significantly reduces the total time required for bulk loading.

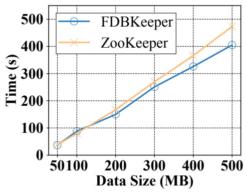
In terms of resource utilization, our experimental results show that FDBKeeper can fully utilize all available CPU resources. In contrast, ZooKeeper can only use approximately 15 cores due to its single-writer architecture limitations. Interestingly, although FDBKeeper has higher storage usage, its memory consumption is lower than that of ZooKeeper, which indicates that FDBKeeper has certain advantages in terms of memory usage efficiency. In the metadata management scenario, the trade-off between storage and performance is reasonable, as metadata typically constitutes only a small portion of the total data volume, while performance improvement is critical for the rapid restart of the cluster.

6.5 Hardware Resource Cost at Moqi

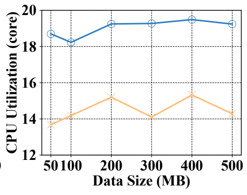
To demonstrate the cost efficacy of our study, this section reports a detailed case study of the production environment at Moqi Inc. Each ClickHouse cluster requires a ZooKeeper cluster with three nodes. In contrast, with FDBKeeper, multiple ClickHouse clusters can share a single FDB cluster and allocate resources as needed. Our objective is to compare the hardware resource costs of replacing ZooKeeper with FDBKeeper in ClickHouse clusters of varying deployment sizes at Moqi Inc. The experimental results are illustrated in Figure 15, where C1, C2, C3, and C4 represent 40, 60, 80, and 100 ClickHouse clusters, respectively. The cost of resources is assessed based on the resources utilized by a single ZooKeeper cluster. For example, compared to ZooKeeper, the resource that supports the stable running of the cluster in the FDB cluster for C2 (60 ClickHouse clusters equipped with 60 ZooKeeper clusters) is roughly equivalent to 40 ZooKeeper clusters. It is evident that, when enabling a ClickHouse cluster of the same scale, FDBKeeper requires fewer resources than ZooKeeper, reducing hardware resource costs by an average of 33%, which directly reduces the monetary cost at Moqi Inc. This enables FDBKeeper to serve as a replacement for ZooKeeper in large-scale production environments.

6.6 Throughput with Failure at Moqi

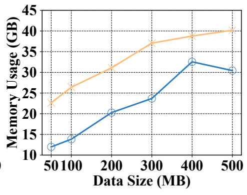
In this section, we evaluate the fault tolerance capabilities of ZooKeeper and FDBKeeper when subjected to the fault scenarios using Moqi's production workload with a read-to-write ratio of 6:4 (60% reads, 40% writes). For both ZooKeeper and FDBKeeper, we utilized the nine-node configuration. For ZooKeeper, we designated node 1 as the leader, nodes 2-3 as followers, and nodes 4-9 as observers.



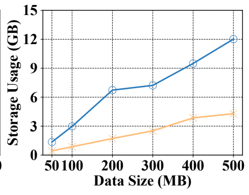
(a) Time



(b) CPU Utilization



(c) Memory Usage



(d) Storage Usage

Figure 14: Performance comparison of bulk loading.

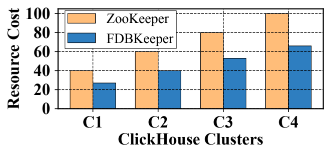


Figure 15: Resource cost comparison.

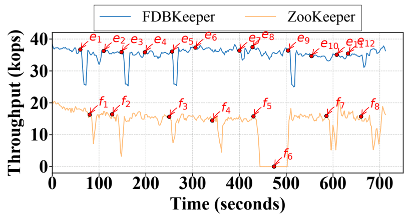


Figure 16: Throughput under failure.

For FDB, we configured node 1 with one coordinator process, 1 transaction process, and 4 storage processes. Nodes 2-9 were each configured with 1 transaction process, 6 storage processes, and 2 stateless processes. Additionally, we use three replica settings for FDBKeeper. Detailed experimental configurations are presented in Table 1. For the ZooKeeper evaluation, we have injected five distinct failure events [35]: (1) failure and subsequent recovery of node 2 (i.e., f_1, f_2); (2) failure and subsequent recovery of node 3 (i.e., f_3, f_4); (3) simultaneous failure of node 2 and node 3, followed by their recovery (i.e., f_5, f_6); (4) failure of node 1 (i.e., f_7); (5) recovery of node 1 (i.e., f_8). Similarly, for FDBKeeper, we have also injected seven failure events: (1) failure and subsequent recovery of the transaction process on node 2 (i.e., e_1, e_2); (2) failure and subsequent recovery of the transaction process on node 3 (i.e., e_3, e_4); (3) simultaneous failure of the transaction processes on node 2 and node 3, followed by their recovery (i.e., e_5, e_6); (4) failure of the busy storage node, followed by its recovery (i.e., e_7, e_8); (5) failure of node 2, followed by its recovery (i.e., e_9, e_{10}); (6) failure of the coordinator process on node 1 (i.e., e_{11}); (7) recovery of the coordinator process on node 1 (i.e., e_{12}).

The experimental results are illustrated in Figure 16. The throughput data is sampled every 2 seconds. The experimental results demonstrate that FDBKeeper maintains significantly higher throughput than ZooKeeper when operating under failure conditions. When the transaction process of a node fails (e.g., e_1, e_3, e_5), FDBKeeper experiences a reduction in throughput. This is primarily because each transaction requires consensus among three transaction processes before it can be committed. When a transaction process fails, certain transactions are temporarily blocked from committing, causing a measurable decrease in system throughput. These pending transactions can resume committing only after the cluster reallocates and selects three transaction processes. When the busy storage process fails (i.e., e_7, e_8), the throughput is barely affected. This is primarily because the FDBKeeper uses a three-replica configuration, which distributes metadata partition replicas across distinct storage nodes. Consequently, this architecture ensures metadata integrity during storage node failures while maintaining stable throughput performance under failure. When node 2 fails (i.e., e_9), we observe a significant degradation in system throughput (i.e., e_{10}). This occurs primarily because node 2 hosts 1 transaction process, 2 stateless processes, and 6 storage processes. The failure of stateless processes produces no measurable impact on system performance. The critical factor is that both transaction and storage processes enter a failure state. This cause is the same as transaction or storage process failures. When the coordinator process of a node fails (i.e., e_{11}), throughput is almost unaffected (i.e., e_{12}). This is primarily because the FDB client directly sends transactions and storage requests, so a temporary failure in the coordinator does not impact throughput.

In ZooKeeper, when a follower failure occurs (e.g., f_1, f_3), its throughput changes, which has the same impact as the transaction process of FDBKeeper. When the followers recover (e.g., f_2, f_4), the leader is re-elected, reducing throughput during the process. When both followers fail (i.e., f_5), the entire system stops serving. After recovery, a new leader is elected to resume service (i.e., f_6). After the leader is killed (i.e., f_7), the throughput does not drop to zero. In a ZooKeeper cluster with n nodes, at least $(n/2) + 1$ nodes are required to form a quorum and provide services. Therefore, the 2-node cluster votes for a new leader and continues providing services (i.e., f_8).

FDBKeeper is built on top of FDB, so failure recovery in FDBKeeper is handled by FDB. FDB provides robust high availability and failure recovery capabilities, addressing process failure, node failure, network partitioning, cascading failures, and more. This is one of the reasons we chose FDB to implement the coordination service. The cluster controller (CC) plays a pivotal role in the FDB recovery process. It acts as the leader, elected by the coordinators, and determines whether recovery should be triggered. The failure recovery process of FDB consists of four stages: **(1) Reading and locking the coordinated state**. In this phase, the coordinated state (i.e., configurations of the transaction system) is read from the coordinators. The coordinated state (cstate) is locked to ensure that only one CC can modify it. **(2) Recovering previous transaction system states**. In this phase, the active transaction logs are collected, and all roles are informed of the recovery details, triggering the recovery process when necessary. **(3) Writing the coordinated state**. The coordinators store the transaction system's information. The CC needs to write the new transaction logs into the coordinators' states to ensure consensus and fault tolerance. **(4) Accepting new transactions**. Finally, the transaction system starts to accept new transactions. The CC waits for all transaction logs to join the system and for all storage servers to roll back their prefetched uncommitted data before declaring the system fully recovered. Details of FDB failure recovery are available in [14, 16, 51].

6.7 Discussion

The motivation behind this work was to explore the implementation of a coordination service based on a distributed key-value database to address the issue in ZooKeeper, particularly the bottleneck caused by its single-writer architecture. However, we found that FDBKeeper suffers from performance degradation (e.g., throughput) in high-conflict scenarios, such as when multiple clients concurrently and frequently modify the same key.As a result, the current design of FDBKeeper primarily addresses performance issues in high-concurrency scenarios. In the future, we aim to resolve the performance issues in high-conflict scenarios within the current coordination service. The current design of FDBKeeper, based on high-concurrency distributed databases, makes it possible to solve this issue. One promising approach is to reduce transaction latency by optimizing the conflict validation process in concurrency control, thereby improving throughput in high-conflict scenarios.

7 DEPLOYMENT EXPERIENCES

We derived three lessons from our deployment of FDBKeeper.

System configuration and maintenance. Our deployment experience reveals that optimal configuration requires balancing two layers: FDB's system parameters and ZooKeeper-compatible operational parameters. We extract representative workloads from production clusters for systematic stress testing, focusing on key metrics including read-to-write ratios, the degree of concurrency, throughput, latency, CPU utilization, memory, and storage usage. We use these stress test results to guide and validate our configuration tuning process. We found that starting with FDB's recommended configurations [15] and iteratively tuning FDBKeeper-specific parameters (e.g., session timeout) based on actual workload characterization yields the best results for most deployments.

Upgrade and migration. In our production deployment, we implemented a rapid migration strategy to minimize service disruption. The migration process commences by transitioning the ZooKeeper cluster to read-only mode while concurrently initiating high performance bulk loading to transfer data to FDBKeeper. During this brief migration window, incoming write requests are rejected with retryable error responses. Upon successful completion of data migration and verification, we execute a traffic switchover, redirecting all client requests from ZooKeeper to FDBKeeper. The migration process typically occurs during the business's off-peak period, minimizing the impact on the business. Although this migration approach necessitates a maintenance window, our high-concurrency bulk loading optimization ensures minimal service disruption throughout the migration process.

Monitoring and debugging. FDBKeeper provides comprehensive observability through integrated monitoring and debugging capabilities. At the metrics layer, FDBKeeper seamlessly integrates both ZooKeeper-compatible monitoring tools (e.g., 4-letter words command [13]) and FDB's native metrics using *status json* from *fdbcli* [18] for system insights. In our production deployments, we also utilize Prometheus to capture and calculate all monitoring indicators, while Grafana further aggregates and visualizes the operational metrics [3]. Key monitoring dimensions include client- and server-side latency (p50, p95, p99), throughput, client busyness, CPU utilization, memory, and storage usage. This monitoring stack has proven effective in identifying performance anomalies and maintaining SLA compliance across large-scale deployments. FDBKeeper captures client operation events (e.g., *Set*, *Get*) and their execution status, while monitoring system metrics to detect performance anomalies.

8 RELATED WORKS

Coordination Services. Google Chubby [24] is a distributed locking service that offers coarse-grained locking and reliable shared storage for small files. ZooKeeper [35] is another prominent distributed coordination service similar to Chubby, employing a single-primary architecture. The difference lies in ZooKeeper being based on the highly available primary backup protocol Zab [36], allowing all replica nodes to offer read services while ensuring sequential consistency of write operations. ZooKeeper is extensively employed in the industry. Etcd [31] is also a popular coordination service with a primary backup architecture, but it provides a simpler KV model. Consul [33] divides the coordinated service data into multiple shards, each with an independent primary node, which is equivalent to the deployment of multiple coordination service clusters to balance write requests. However, this approach lacks consistency guarantees across shards. ZooNet [37] further ensures consistency across multiple clusters based on a coordination service. Furthermore, ClickHouse rebuilds ZooKeeper based on the raft protocol [42] to mitigate the single-point bottleneck [44]. FaaSKeeper [27] is a ZooKeeper-based serverless coordination service with native cloud services and servers.

Hierarchical Namespace Approaches. Recent research attempts to take advantage of databases by managing metadata using KV databases to achieve scalability and high performance [29]. IndexFS [43] was the first to utilize the LSM tree-based KV database to store file system metadata, using directory IDs and file name hashes as keys and file names, attributes, data mappings, and small files as values. Similarly, Ceph [22] also adopts a similar strategy. In HDFS [45], the NameNode cluster manages the HDFS metadata. HopeFS [41] replaces the NameNode in HDFS with NewSQL to alleviate the single-point bottleneck. To maximize the performance of KV databases, some studies further optimize the mapping of metadata to KV pairs for different file systems [38–40, 48–50]. The hierarchical namespace model of ZooKeeper is similar to that of file systems.

9 CONCLUSION

In this paper, we introduce FDBKeeper, which enables a scalable coordination service based on the FDB. We implement key-value-based hierarchical namespace management and session management, which are key components of ZooKeeper. We also implement the consistency guarantee based on the transaction of FDB. In addition, we have conducted correctness analysis and experimental evaluation of the FDBKeeper. Extensive experiments demonstrate better performance and scalability in a cluster environment. We successfully replaced ZooKeeper with FDBKeeper in the production environment at Moqi Inc.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments. This work was supported in part by the National Natural Science Foundation of China under Grant No. U22B2020. Peng Cai is the corresponding author.

REFERENCES

[1] 2021. KRaft. https://kafka.apache.org

[2] 2023. ClickHouse. https://github.com/ClickHouse/ClickHouse

[3] 2023. Grafana. Retrieved in March, 2023 from https://grafana.com/.

[4] 2023. Node Exporter. Retrieved in March, 2023 from https://github.com/prometheus/node_exporter.

[5] 2023. Prometheus. Retrieved in March, 2023 from https://prometheus.io/.

[6] 2023. RaftKeeper. Retrived in May, 2023 from https://github.com/JDRaftKeeper/RaftKeeper.

[7] 2024. FoundationDB Consistency. Retrived in Sep, 2024 from https://apple.github.io/foundationdb/consistency.html,.

[8] 2024. Prometheus Pushgateway. Retrieved in September, 2024 from https://github.com/prometheus/pushgateway.

[9] 2024. system-tables-zookeeper. Retrived in May, 2024 from https://clickhouse.com/docs/en/operations/system-tables/zookeeper.

[10] 2024. YugabyteDB - the cloud native distributed SQL database for mission-critical applications. Retrieved in September, 2024 from https://www.yugabyte.com/.

[11] 2024. ZooKeeper: A Distributed Coordination Service for Distributed Applications. Retrieved in September, 2024 from https://zookeeper.apache.org/doc/current/zookeeperOver.html.

[12] 2024. ZooKeeper-watch. Retrived in May, 2024 from https://zookeeper.apache.org/doc/r3.9.1/zookeeperProgrammers.html#ch_zkWatches.

[13] 2025. 4 letter words command of ZooKeeper. Retrieved in May, 2025 from https://zookeeper.apache.org/doc/r3.9.1/zookeeperAdmin.html#sc_zkCommands.

[14] 2025. Cluster Coordination. Retrieved in May, 2025 from https://github.com/apple/foundationdb/wiki/Cluster-Coordination.

[15] 2025. Configuration. Retrieved in May, 2025 from https://apple.github.io/foundationdb/configuration.html.

[16] 2025. FDB Recovery Internals. Retrieved in May, 2025 from https://github.com/apple/foundationdb/blob/main/design/recovery-internals.md.

[17] 2025. Known Limitations - FoundationDB. Retrived in January, 2024 from https://apple.github.io/foundationdb/known-limitations.html.

[18] 2025. Machine-Readable Status. Retrieved in May, 2025 from https://apple.github.io/foundationdb/mr-status.html.

[19] 2025. Performance of FoundationDB. Retrieved in June, 2025 from https://apple.github.io/foundationdb/performance.html.

[20] 2025. Proof of Linearizability in FDBKeeper. Retrived in January, 2025 from https://github.com/DASE-iDDS/FDBKeeper/blob/main/FDBKeeper_Theorem_Proof.pdf.

[21] 2025. YCSB - Yahoo! Cloud Serving Benchmark. Retrieved in May, 2025 from https://github.com/brianfrankcooper/YCSB.

[22] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 353–369.

[23] Eric Brewer. 2017. Spanner, truetime and the cap theorem. *Google Research* (2017).

[24] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 335–350.

[25] ByteDance. 2023. *ByConity*. https://github.com/ByConity/ByConity

[26] Yuxing Chen, Anqun Pan, Hailin Lei, Anda Ye, Shuo Han, Yan Tang, Wei Lu, Yunpeng Chai, Feng Zhang, and Xiaoyong Du. 2024. TDSQL: Tencent Distributed Database System. 17, 12 (2024), 3869 – 3882. https://doi.org/10.14778/3685800.3685812

[27] Marcin Copik, Alexandru Calotoiu, Pengyu Zhou, Konstantin Taranov, and Torsten Hoefler. 2022. FaaSKeeper: Learning from Building Serverless Services with ZooKeeper as an Example. *arXiv preprint arXiv:2203.14859* (2022).

[28] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[29] Hao Dai, Yang Wang, Kenneth B Kent, Lingfang Zeng, and Chengzhong Xu. 2022. The state of the art of metadata managements in large-scale distributed file systems—scalability, performance and availability. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3850–3869.

[30] Pavan Edara and Mosha Pasumansky. 2021. Big metadata: when metadata is big data. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3083–3095.

[31] etcd. 2023. Etcd. Retrived in May, 2023 from https://github.com/etcd-io/etcd.

[32] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2021. Strong and efficient consistency with consistency-aware durability. *ACM Transactions on Storage (TOS)* 17, 1 (2021), 1–27.

[33] hashicorp. 2023. Consul. https://github.com/hashicorp/consul

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

[35] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*.

[36] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 245–256.

[37] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. 2016. Modular composition of coordination services. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 251–264.

[38] Siyang Li, Fenlin Liu, Jiwu Shu, Youyou Lu, Tao Li, and Yang Hu. 2018. A flattened metadata service for distributed file systems. *IEEE Transactions on Parallel and Distributed Systems* 29, 12 (2018), 2641–2657.

[39] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. 2017. Locofs: A loosely-coupled metadata service for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[40] Wenhao Lv, Youyou Lu, Yiming Zhang, Peile Duan, and Jiwu Shu. 2022. {InfiniFS}: An efficient metadata service for {Large-Scale} distributed filesystems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 313–328.

[41] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. 2017. {HopsFS}: Scaling hierarchical file system metadata using {NewSQL} databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 89–104.

[42] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.

[43] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 237–248.

[44] Tom Schreiber and Derek Chia. [n.d.]. *ClickHouse Keeper: A ZooKeeper Alternative Written in C++*. https://clickhouse.com/blog/clickhouse-keeper-a-zookeeper-alternative-written-in-cpp

[45] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[46] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company. https://www.db-book.com/

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

[48] Houjun Tang, Suren Byna, Bin Dong, Jialin Liu, and Quincey Koziol. 2017. Someta: Scalable object-centric metadata management for high performance computing. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 359–369.

[49] Teng Wang, Adam Moody, Yue Zhu, Kathryn Mohror, Kento Sato, Tanzima Islam, and Weikuan Yu. 2017. Metakv: A key-value store for metadata management of distributed burst buffers. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1174–1183.

[50] Yiwen Zhang, Jian Zhou, Xinhao Min, Song Ge, Jiguang Wan, Ting Yao, and Daohui Wang. 2022. PetaKV: Building Efficient Key-Value Store for File System Metadata on Persistent Memory. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2022), 843–855.

[51] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. 2021. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*. 2653–2666.