FoundationDB: A Distributed Unbundled Transactional Key Value Store

Jingyu Zhou
Meng Xu
Alexander Shraer
Bala Namasivayam
Apple Inc.

Alex Miller
Evan Tschannen
Steve Atherton
Andrew J. Beamon
Snowflake Inc.

Rusty Sears
John Leach
Dave Rosenthal
Xin Dong
Apple Inc.

Will Wilson
Ben Collins
David Scherer
antithesis.com

Alec Grieser
Young Liu
Alvin Moore
Apple Inc.

Bhaskar Muppana
Xiaoge Su
Vishesh Yadav
Apple Inc.

Abstract

FoundationDB is an open source transactional key value store created more than ten years ago. It is one of the first systems to combine the flexibility and scalability of NoSQL architectures with the power of ACID transactions (a.k.a. NewSQL). FoundationDB adopts an unbundled architecture that decouples an in-memory transaction management system, a distributed storage system, and a built-in distributed configuration system. Each sub-system can be independently provisioned and configured to achieve the desired scalability, high-availability and fault tolerance properties. FoundationDB uniquely integrates a deterministic simulation framework, used to test every new feature of the system under a myriad of possible faults. This rigorous testing makes FoundationDB extremely stable and allows developers to introduce and release new features in a rapid cadence. FoundationDB offers a minimal and carefully chosen feature set, which has enabled a range of disparate systems (from semi-relational databases, document and object stores, to graph databases and more) to be built as layers on top. FoundationDB is the underpinning of cloud infrastructure at Apple, Snowflake and other companies, due to its consistency, robustness and availability for storing user data, system metadata and configuration, and other critical information.

CCS Concepts

• **Information systems → Key-value stores**; **Distributed database transactions**; • **Computing methodologies → Distributed simulation**.

Keywords

unbundled database, OLTP, strict serializability, multiversion concurrency control, optimistic concurrency control, simulation testing

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457559

ACM Reference Format:
Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3457559

1 Introduction

Many cloud services rely on scalable, distributed storage backends for persisting application state. Such storage systems must be fault tolerant and highly available, and at the same time provide sufficiently strong semantics and flexible data models to enable rapid application development. Such services must scale to billions of users, petabytes or exabytes of stored data, and millions of requests per second.

More than a decade ago, NoSQL storage systems emerged offering ease of application development, making it simple to scale and operate storage systems, offering fault-tolerance and supporting a wide range of data models (instead of the traditional rigid relational model). In order to scale, these systems sacrificed transactional semantics, and instead provided eventual consistency, forcing application developers to reason about interleavings of updates from concurrent operations. FoundationDB avoids this trade-off by providing serializable transactions while scaling to handle the large workloads these systems target at. In addition to helping applications correctly manage data stored as simple key-value pairs, this also enables FoundationDB's users to implement more advanced features, such as consistent secondary indices and referential integrity checks [28]. Realizing their importance for building applications, many NoSQL systems retrofitted ACID transactions in recent years. For example Apache Cassandra [45], MongoDB [10], CouchBase [1] and other NoSQL systems now all support some form of ACID, accompanied by SQL dialects.

FoundationDB (FDB) [5] was created in 2009 and gets its name from the focus on providing what we saw as the foundational set of building blocks required to build higher-level distributed systems. It is an ordered, transactional, key-value store natively supporting

multi-key strictly serializable transactions across its entire keyspace. Unlike most databases, which bundle together a storage engine, data model, and query language, forcing users to choose all three or none, FDB takes a modular approach: it provides a highly scalable, transactional storage engine with a minimal yet carefully chosen set of features. It provides no structured semantics, no query language, data model or schema management, secondary indices or many other features one normally finds in a transactional database. Offering these would benefit some applications but others that do not require them (or do so in a slightly different form) would need to work around. Instead, the NoSQL model leaves application developers with great flexibility. While FDB defaults to strictly serializable transactions, it allows relaxing these semantics for applications that don't require them with flexible, fine-grained controls over conflicts.

One of the reasons for its popularity and growing open source community is FoundationDB's focus on the "lower half" of a database, leaving the rest to its "layers"—stateless applications developed on top to provide various data models and other capabilities. With this, applications that would traditionally require completely different types of storage systems, can instead all leverage FDB. Indeed, the wide range of layers that have been built on FDB in recent years are evidence to the usefulness of this unusual design. For example, the FoundationDB Record Layer [28] adds back much of what users expect from a relational database, and JanusGraph [9], a graph database, provides an implementation as a FoundationDB layer [8]. In its newest release, CouchDB [2] (arguably the first NoSQL system) is being re-built as a layer on top of FoundationDB.

Testing and debugging distributed systems is at least as hard as building them. Unexpected process and network failures, message reorderings, and other sources of non-determinism can expose subtle bugs and implicit assumptions that break in reality, which are extremely difficult to reproduce or debug. The consequences of such subtle bugs are especially severe for database systems, which purport to offer perfect fidelity to an unambiguous contract. Moreover, the stateful nature of a database system means that any such bug can result in subtle data corruption that may not be discovered for months. Model checking techniques can verify the correctness of distributed protocols, but often fall short of checking the actual implementation. Deep bugs [46], which only happen when multiple crashes or restarts occur in a particular sequence, pose a challenge even for end-to-end testing infrastructure. The development of FDB took a radical approach—before building the database itself, we built a deterministic database simulation framework that can simulate a network of interacting processes and a variety of disk, process, network, and request-level failures and recoveries, all within a single physical process. This rigorous testing in simulation makes FDB extremely stable, and allows its developers to introduce new features and releases in a rapid cadence. This is unusual not only for distributed databases, but even for centralized systems.

FDB adopts an unbundled architecture [50] that comprises a control plane and a data plane. The control plane manages the metadata of the cluster and uses Active Disk Paxos [27] for high availability. The data plane consists of a transaction management system, responsible for processing updates, and a distributed storage layer serving reads; both can be independently scaled out. FDB achieves strict serializability through a combination of optimistic concurrency control (OCC) [44] and multi-version concurrency control (MVCC) [18]. Besides a lock-free architecture, one of the features distinguishing FDB from other distributed databases is its approach to handling failures. Unlike most similar systems, FDB does not rely on quorums to mask failures, but rather tries to eagerly detect and recover from them by reconfiguring the system. This allows us to achieve the same level of fault tolerance with significantly fewer resources: FDB can tolerate f failures with only $f + 1$ (rather than $2f + 1$) replicas. This approach is best suited for deployments in a local or metro area. For WAN deployments, FDB offers a novel strategy that avoids cross-region write latencies while providing automatic failover between regions without losing data.

The contributions of this paper are:

- An open source distributed storage system, FoundationDB, combining NoSQL and ACID, used in production at Apple, Snowflake, VMWare and other companies, satisfying their stringent scalability, availability, and durability requirements;
- A carefully chosen feature set that has been used to implement a range of widely disparate storage systems;
- An integrated deterministic simulation framework that makes FoundationDB one of the most stable systems of its kind; and
- A unique architecture and approach to transaction processing, fault tolerance, and high availability.

The remainder of this paper is organized as follows. Section 2 details the design of FDB. Section 3 describes geo-replication and failover. Section 4 discusses the deterministic simulation framework. Section 5 evaluates the performance of FDB. Section 6 describes our lessons learned from developing FDB. Section 7 summarizes related work and Section 8 concludes the paper.

2 Design

A production database needs to solve many problems, including data persistence, data partitioning, load balancing, membership and failure detection, failure recovery, replica placement and synchronization, overload control, scaling, concurrency and job scheduling, system monitoring and alerting, backup, multi-language client library, system upgrade and deployment, and configuration management. Discussing all these details is not possible, so this paper focuses on the architectural design of FDB and its implications on transaction management, replication, and fault tolerance.

2.1 Design Principles

The main design principles of FDB are:

- *Divide-and-Conquer (or separation of concerns).* FDB decouples the transaction management system (write path) from the distributed storage (read path) and scales them independently. Within the transaction management system, processes are assigned various roles representing different aspects of transaction management, including timestamp management, accepting commits, conflict detection, and logging. Furthermore, cluster-wide orchestrating tasks, such as overload control, load balancing, and failure recovery are also divided and serviced by additional heterogeneous roles.

- *Make failure a common case.* For distributed systems, failure is a norm rather than an exception. In the transaction management system of FDB, we handle all failures through the recovery path: instead of fixing all possible failure scenarios, the transaction system proactively shuts down when it detects a failure. As a result, all failure handling is reduced to a single recovery operation, which becomes a common and well-tested code path. Such error handling strategy is desirable as long as the recovery is quick, and pays dividends by simplifying the normal transaction processing.
- *Fail fast and recover fast.* To improve availability, FDB strives to minimize Mean-Time-To-Recovery (MTTR), which includes the time to detect a failure, proactively shut down the transaction management system, and recover. In our production clusters, the total time is usually less than five seconds (see Section 5.3).
- *Simulation testing.* FDB relies on a randomized, deterministic simulation framework for testing the correctness of its distributed database. Because simulation tests are both efficient and repeatable, they not only expose deep bugs [46], but also boost developer productivity and the code quality of FDB.

2.2 System Interface

FDB exposes operations to read and modify single keys as well as ranges of keys. The *get()* and *set()* operations read and write a single key-value pair, respectively. For ranges, *getRange*() returns a sorted list of keys and their values within the given range; and *clear*() deletes all keys-value pairs whithin a range or starting with a certain key prefix.

An FDB transaction observes and modifies a snapshot of the database at a certain version and changes are applied to the underlying database only when the transaction commits. A transaction's writes (i.e., *set()* and *clear()* calls) are buffered by the FDB client until the final *commit()* call, and read-your-write semantics are preserved by combining results from database look-ups with uncommitted writes of the transaction. Key and value sizes are limited to 10 KB and 100 KB respectively for better performance. Transaction size is limited to 10 MB, including the size of all written keys and values as well as the size of all keys in read or write conflict ranges that are explicitly specified.

2.3 Architecture

An FDB cluster has a control plane for managing critical system metadata and cluster-wide orchestration, and a data plane for transaction processing and data storage, as illustrated in Figure 1.

2.3.1 Control Plane The control plane is responsible for persisting critical system metadata, i.e., the configuration of transaction systems, on `Coordinators`. These `Coordinators` form a disk Paxos group [27] and select a singleton `ClusterController`. The `ClusterController` monitors all servers in the cluster and recruits three singleton processes, `Sequencer`, `DataDistributor`, and `Ratekeeper`, which are re-recruited if they fail or crash. The `Sequencer` assigns read and commit versions to transactions. The `DataDistributor` is responsible for monitoring failures and balancing data among `StorageServers`. `Ratekeeper` provides overload protection for the cluster.

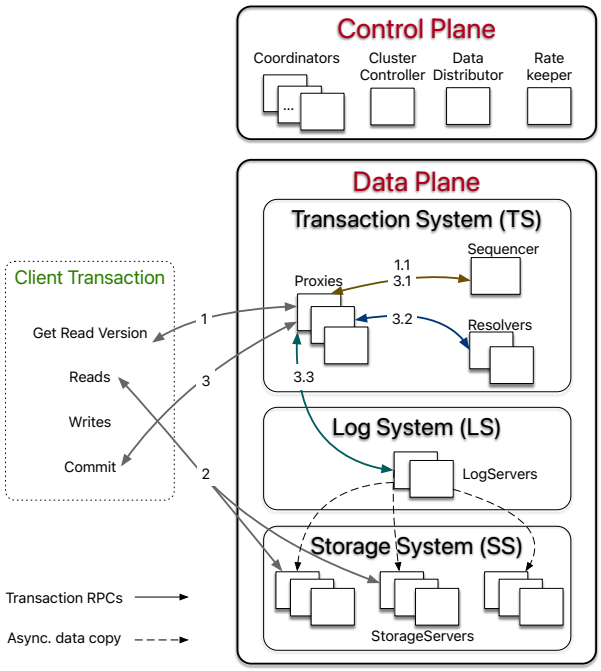


Figure 1: The architecture and the transaction processing of FDB.

2.3.2 Data Plane FDB targets OLTP workloads that are read-mostly, read and write a small set of keys, have low contention, and require scalability. FDB chooses an unbundled architecture [50]: a distributed transaction management system (TS) performs in-memory transaction processing, a log system (LS) stores Write-Ahead-Log (WAL) for TS, and a separate distributed storage system (SS) is used for storing data and servicing reads. The TS provides transaction processing and consists of a `Sequencer`, `Proxies`, and `Resolvers`, all of which are stateless processes. The LS contains a set of `LogServers` and the SS has a number of `StorageServers`. This scales well to Apple's largest transactional workloads [28].

The `Sequencer` assigns a read version and a commit version to each transaction and, for historical reasons, also recruits `Proxies`, `Resolvers`, and `LogServers`. `Proxies` offer MVCC read versions to clients and orchestrate transaction commits. `Resolvers` check for conflicts between transactions. `LogServers` act as replicated, sharded, distributed persistent queues, where each queue stores WAL data for a `StorageServer`.

The SS consists of a number of `StorageServers` for serving client reads, where each `StorageServer` stores a set of data shards, i.e., contiguous key ranges. `StorageServers` are the majority of processes in the system, and together they form a distributed B-tree. Currently, the storage engine on each `StorageServer` is a modified version of SQLite [41], with enhancements that make range clears faster, defer deletion to a background task, and add support for asynchronous programming.

2.3.3 Read-Write Separation and Scaling FDB's design is decoupled; processes are assigned different roles (e.g., `Coordinators`,

StorageServers, Sequencer), and the database scales by expanding the number of processes for each role. This separates the scaling of client reads from client writes (i.e., transaction commits). Because clients directly issue reads to sharded StorageServers, reads scale linearly with the number of StorageServers. Similarly, writes are scaled by adding more processes to Proxies, Resolvers, and LogServers in TS and LS. For this reason, MVCC data is stored in the SS, which is different from Deuteronomy [48, 51] that stores the MVCC data in TS. The singletons (e.g., ClusterController and Sequencer) and Coordinators on the control plane are not performance bottlenecks, because they only perform limited metadata operations.

2.3.4 Bootstrapping FDB has no external dependency on other services. All user data and most of the system metadata (keys that start with 0xFF prefix) are stored in StorageServers. The metadata about StorageServers is persisted in LogServers, and the configuration of LS (i.e., information about LogServers) is stored in all Coordinators. Using Coordinators as a disk Paxos group, servers attempt to become the ClusterController if one does not exist. The newly elected ClusterController recruits a new Sequencer, which reads the configuration of old LS stored in Coordinators and spawns a new TS and LS. From the old LS, Proxies recover system metadata, including information about all StorageServers. The Sequencer waits until the new TS finishes recovery (see Section 2.4.4), and then writes the new LS configuration to all Coordinators. At this time, the new transaction system becomes ready to accept client transactions.

2.3.5 Reconfiguration Whenever there is a failure in the TS or LS, or a database configuration change, a reconfiguration process brings the transaction management system to a new configuration, i.e., a clean state. Specifically, the Sequencer process monitors the health of Proxies, Resolvers, and LogServers. If any one of the monitored processes fails or the database configuration changes, the Sequencer process terminates. The ClusterController will detect the Sequencer failure event, then recruit a new Sequencer, which follows the above bootstrapping process to spawn the new TS and LS instance. In this way, transaction processing is divided into epochs, where each epoch represents a generation of the transaction management system with its unique Sequencer process.

2.4 Transaction Management

In the following, we first describe the end-to-end transaction processing and strict serializability, then discuss transaction logging and recovery.

2.4.1 End-to-end Transaction Processing As illustrated in Figure 1, a client transaction starts by contacting one of the Proxies to obtain a read version (i.e., a timestamp). The Proxy then asks the Sequencer for a read version that is guaranteed to be no less than any previously issued transaction commit version, and this read version is sent back to the client. Then the client may issue multiple reads to StorageServers and obtain values at that specific read version. Client writes are buffered locally without contacting the cluster. At commit time, the client sends the transaction data, including the read and write sets (i.e., key ranges), to one of the Proxies and waits for a commit or abort response from the Proxy. If the transaction cannot commit, the client may choose to restart the transaction from the beginning again.

A Proxy commits a client transaction in three steps. First, the Proxy contacts the Sequencer to obtain a commit version that is larger than any existing read versions or commit versions. The Sequencer chooses the commit version by advancing it at a rate of one million versions per second. Then, the Proxy sends the transaction information to range-partitioned Resolvers, which implement FDB's optimistic concurrency control by checking for *read-write* conflicts. If all Resolvers return with no conflict, the transaction can proceed to the final commit stage. Otherwise, the Proxy marks the transaction as aborted. Finally, committed transactions are sent to a set of LogServers for persistence. A transaction is considered committed after all designated LogServers have replied to the Proxy, which reports the committed version to the Sequencer (to ensure that later transactions' read versions are after this commit) and then replies to the client. At the same time, StorageServers continuously pull mutation logs from LogServers and apply committed updates to disks.

In addition to the above *read-write transactions*, FDB also supports *read-only transactions* and *snapshot reads*. A read-only transaction in FDB is both serializable (happens at the read version) and performant (thanks to the MVCC), and the client can commit these transactions locally without contacting the database. This is particularly important because the majority of transactions are read-only. Snapshot reads in FDB selectively relax the isolation property of a transaction by reducing conflicts, i.e., concurrent writes will not conflict with snapshot reads.

2.4.2 Support Strict Serializability FDB implements Serializable Snapshot Isolation (SSI) by combining OCC with MVCC. Recall that a transaction T_x gets both its read version and commit version from Sequencer, where the read version is guaranteed to be no less than any committed version when T_x starts and the commit version is larger than any existing read or commit versions. This commit version defines a serial history for transactions and serves as Log Sequence Number (LSN). Because T_x observes the results of all previous committed transactions, FDB achieves strict serializability. To ensure there is no gaps between LSNs, the Sequencer returns the previous commit version (i.e., previous LSN) with commit version. A Proxy sends both LSN and previous LSN to Resolvers and LogServers so that they can serially process transactions in the order of LSNs. Similarly, StorageServers pull log data from LogServers in increasing LSNs as well.

Algorithm 1 illustrates the lock-free conflict detection algorithm on Resolvers. Specifically, each Resolver maintains a history *lastCommit* of recently modified key ranges by committed transactions, and their corresponding commit versions. The commit request for T_x comprises two sets: a set of modified key ranges R_w, and a set of read key ranges R_r, where a single key is converted to a single key range. The read set is checked against the modified key ranges of concurrent committed transactions (line 1—5), which prevents phantom reads. If there are no read-write conflicts, Resolvers admit the transaction for commit and update the list of modified key ranges with the write set (line 6—7). For snapshot reads, they are not included in the set R_r. In practice, *lastCommit* is represented as a version-augmented probabilistic SkipList [56].

Unlike *write-snapshot isolation* [68], which assigns the timestamp after checking R_r, FDB decides the commit version before the conflict detection. This allows FDB to efficiently batch-process both version assignments and conflict detection. Our micro-benchmark shows that one single-threaded `Resolver` can easily handle 280K TPS (each transaction reads a random key range and writes another random key range).

Algorithm 1: Check conflicts for transaction T_x.

Require: *lastCommit*: a map of key range → last commit version

```
for each range ∈ R_r do
    ranges = lastCommit.intersect(range)
    for each r ∈ ranges do
        if lastCommit[r] > T_x.readVersion then
            return abort;
// commit path
for each range ∈ R_w do
    lastCommit[range] = T_x.commitVersion;
return commit;
```

The entire key space is divided among `Resolvers` so that the above read-write conflict detection algorithm may be performed in parallel. A transaction can commit only when all `Resolvers` admit the transaction. Otherwise, the transaction is aborted. It is possible that an aborted transaction is admitted by a subset of `Resolvers`, and they have already updated their history of *lastCommit*, which may cause other transactions to conflict (i.e., a false positive). In practice, this has not been an issue for our production workloads, because transactions' key ranges usually fall into one `Resolver`. Additionally, because the modified keys expire after the MVCC window, the false positives are limited to only happen within the short MVCC window time (i.e., 5 seconds). Finally, the key ranges of `Resolvers` are dynamically adjusted to balance their loads.

The OCC design of FDB avoids the complicated logic of acquiring and releasing (logical) locks, which greatly simplifies interactions between the TS and the SS. The price paid for this simplification is to keep the recent commit history in `Resolvers`. Another drawback is not guaranteeing that transactions will commit, a challenge for OCC. Because of the nature of our multi-tenant production workload, the transaction conflict rate is very low (less than 1%) and OCC works well. If a conflict happens, the client can simply restart the transaction.

2.4.3 Logging Protocol After a `Proxy` decides to commit a transaction, the log message is broadcast to all `LogServers`. As illustrated in Figure 2, the `Proxy` first consults its in-memory shard map to determine the `StorageServers` responsible for the modified key range. Then the `Proxy` attaches `StorageServer` tags 1, 4, and 6 to the mutation, where each tag has a preferred `LogServer` for storage. In this example, tags 1 and 6 have the same preferred `LogServer`. Note the mutation is only sent to the preferred `LogServers` (1 and 4) and an additional `LogServer` 3 to meet the replication requirements. All other `LogServers` receive an empty message body. The log message header includes both LSN and the previous LSN

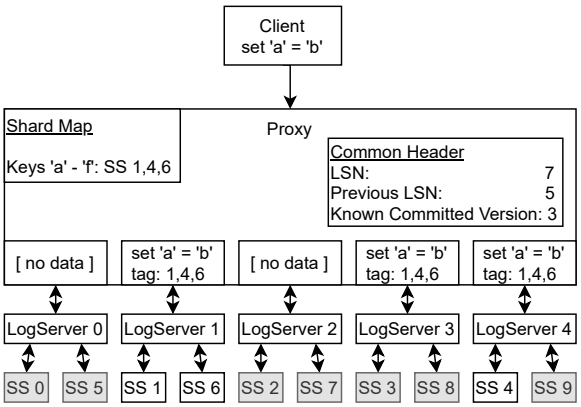


Figure 2: Proxy writes a client mutation to `LogServers` after sequencing and resolution. Later, the mutation will be asynchronously replicated to `StorageServers`.

obtained from the `Sequencer`, as well as the known committed version (KCV) of this `Proxy`. `LogServers` reply to the `Proxy` once the log data is made durable, and the `Proxy` updates its KCV to the LSN if all replica `LogServers` have replied and this LSN is larger than the current KCV.

Shipping the redo log from the LS to the SS is not a part of the commit path and is performed in the background. In FDB, `StorageServers` aggressively fetch redo logs from `LogServers` before they are durable on the LS, allowing very low latency for serving multi-version reads. Figure 3 shows the time lag between `StorageServers` and `LogServers` in one of our production clusters for a 12-hour period, where the 99.9 percentile of the average and maximum delay is 3.96 ms and 208.6 ms, respectively. Because this lag is small, when client read requests reach `StorageServers`, the requested version (i.e., the latest committed data) is usually already available. If due to a small delay the data is not available to read at a `StorageServer` replica, the client either waits for the data to become available or issues a second request to another replica [32]. If both reads timed out, the client gets a retryable error to restart the transaction.

Because the log data is already durable on `LogServers`, `StorageServers` can buffer updates in memory and only persist batches of data to disks with a longer delay, thus improving I/O efficiency by coalescing the updates. Aggressively pulling redo logs from `LogServers` means that semi-committed updates, i.e., operations in transactions that are aborted during recovery (e.g., due to `LogServer` failure), need to be rolled back (see Section 2.4.4).

2.4.4 Transaction System Recovery Traditional database systems often employ the ARIES recovery protocol [53], which depends on a write-ahead log (WAL) and periodic, coarse-grained checkpoints. During the recovery, the system processes redo log records from the last checkpoint by re-applying them to the relevant data pages. This will bring the database to a consistent state at the point of failure and in-flight transactions during the crash can be rolled back by executing the undo log records.

In FDB, the recovery is purposely made very cheap—there is no checkpoint, and no need to re-apply redo or undo log during

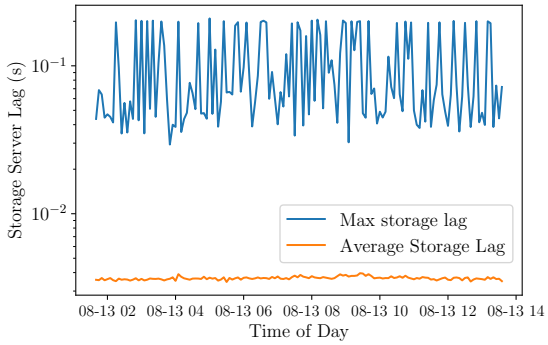


Figure 3: The lag from `StorageServers` to `LogServers`.

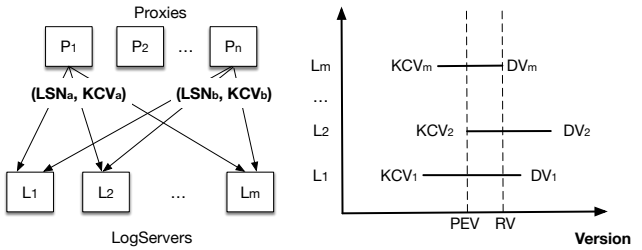


Figure 4: An illustration of RV and PEV. On the left, a `Proxy` sends redo logs to `LogServers` with a *KCV* and the *LSN*, and `LogServers` keep the maximum *KCV* received. On the right, recovery uses the maximum of *KCV*s and the minimum of *DV*s on a set of `LogServers` as *PEV* and *RV*, respectively.

recovery. This is possible because of a great simplifying principle of traditional databases: the redo log processing is the same as normal log forward path. In FDB, `StorageServers` always pull logs from `LogServers` and apply them in the background, which essentially decouples redo log processing from the recovery. The recovery process starts by detecting a failure, recruits a new transaction system, and ends when old `LogServers` are no longer needed. The new transaction system can even accept transactions before all the data on old `LogServers` is processed, because the recovery only needs to find out the end of redo log and re-applying the log is performed asynchronously by `StorageServers`.

For each epoch, the `Sequencer` executes recovery in several steps. First, the `Sequencer` reads the previous transaction system states (i.e. configurations of the transaction system) from `Coordinators` and locks the coordinated states to prevent another `Sequencer` process from recovering at the same time. Then the `Sequencer` recovers previous transaction system states, including the information about all older `LogServers`, stops these `LogServers` from accepting transactions, and recruits a new set of `Proxies`, `Resolvers`, and `LogServers`. After previous `LogServers` are stopped and a new transaction system is recruited, the `Sequencer` then writes the coordinated states with current transaction system information. Finally, the `Sequencer` accepts new transaction commits.

Because `Proxies` and `Resolvers` are stateless, their recoveries have no extra work. In contrast, `LogServers` save the logs of committed transactions, and we need to ensure all previously committed transactions are durable and retrievable by `StorageServers`. That is, for any transactions that the `Proxies` may have sent back a commit response, their logs are persisted in multiple `LogServers` satisfying the configured replication degree.

The essence of the recovery of old `LogServers` is to determine the end of redo log, i.e., a Recovery Version (RV). Rolling back undo log is essentially discarding any data after RV in the old `LogServers` and `StorageServers`. Figure 4 illustrates how RV is determined by the `Sequencer`. Recall that a `Proxy` request to `LogServers` piggybacks its KCV, the maximum LSN that this `Proxy` has committed. Each `LogServer` keeps the maximum KCV received and a Durable Version (DV), which is the maximum persisted LSN. During a recovery, the `Sequencer` attempts to stop all m old `LogServers`, where each response contains the DV and KCV on that `LogServer`. Assume the replication degree for `LogServers` is k. Once the `Sequencer` has received more than $m - k$ replies [1], the `Sequencer` knows the previous epoch has committed transactions up to the maximum of all KCVs, which becomes the previous epoch's end version (PEV). All data before this version has been fully replicated. For current epoch, its start version is $PEV + 1$ and the `Sequencer` chooses the minimum of all DVs to be the RV. Logs in the range of $[PEV + 1, RV]$ are copied from previous epoch's `LogServers` to the current ones, for healing the replication degree in case of `LogServer` failures. The overhead of copying this range is very small because it only contains a few seconds' log data.

When `Sequencer` accepts new transactions, the first is a special recovery transaction that informs `StorageServers` the RV so that they can roll back any data larger than RV. The current FDB storage engine consists of an unversioned SQLite [41] B-tree and in-memory multi-versioned redo log data. Only mutations leaving the MVCC window (i.e., committed data) are written to SQLite. The rollback is simply discarding in-memory multi-versioned data in `StorageServers`. Then `StorageServers` pull any data larger than version *PEV* from new `LogServers`.

2.5 Replication

FDB uses a combination of various replication strategies for different data to tolerate f failures:

- *Metadata replication.* System metadata of the control plane is stored on `Coordinators` using Active Disk Paxos [27]. As long as a quorum (i.e., majority) of `Coordinators` are live, this metadata can be recovered.
- *Log replication.* When a `Proxy` writes logs to `LogServers`, each sharded log record is synchronously replicated on $k = f+1$ `LogServers`. Only when all k have replied with successful persistence can the `Proxy` send back the commit response to the client. Failure of a `LogServer` results in a transaction system recovery (see Section 2.4.4).
- *Storage replication.* Every shard, i.e., a key range, is asynchronously replicated to $k = f + 1$ `StorageServers`, which

[1] This $m - k$ can be reduced to m/k by organizing `LogServers` into Copysets [29], i.e., one from each set, thus increasing fault tolerance.

is called a *team*. A `StorageServer` usually hosts a number of shards so that its data is evenly distributed across many teams. A failure of a `StorageServer` triggers `DataDistributor` to move data from teams containing the failed process to other healthy teams.

Note the storage team abstraction is more sophisticated than the Copyset policy [29]. Copyset reduces the chance of data loss during simultaneous process failures by assigning shards to a limited number of possible k-process groups. Otherwise, any k-process failure can cause a higher probability of data loss. In our deployment, teams need to consider multiple dimensions: each replica group needs to satisfy several constraints at the same time. For instance, a cluster can have a number of hosts and each host runs multiple processes. In this case, a failure can happen at the host level, affecting many processes. Thus, a replica group cannot place two processes on the same host. More generally, the placement needs to ensure at most one process in a replica group can be placed in a *fault domain*, e.g., racks or availability zones in a cloud environment.

To solve the above problem, we designed a hierarchical replication policy to reduce the chance of data loss during simultaneous failures. Specifically, we construct the replica set at both host and process levels and ensure that each process group belongs to a host group that satisfies the fault domain requirement. This policy has the benefits that data loss can only happen when all hosts in a selected host group fail simultaneously; that is, when we experience concurrent failures in multiple fault domains. Otherwise, each team is guaranteed to have at least one process live and there is no data loss if any one of the fault domains remains available.

2.6 Other Optimizations

Transaction batching. To amortize the cost of committing transactions, the `Proxy` groups multiple transactions received from clients into one batch, asks for a single commit version from the `Sequencer`, and sends the batch to `Resolvers` for conflict detection. The `Proxy` then writes committed transactions in the batch to `LogServers`. The transaction batching reduces the number of calls to obtain a commit version from the `Sequencer`, allowing `Proxies` to commit tens of thousands of transactions per second without significantly impacting the `Sequencer`'s performance. Additionally, the batching degree is adjusted dynamically, shrinking when the system is lightly loaded to improve commit latency, and increasing when the system is busy in order to sustain high commit throughput.

Atomic operations. FDB supports atomic operations such as atomic add, bitwise "and" operation, compare-and-clear, and set-versionstamp. These atomic operations enable a transaction to write a data item without reading its value, saving a round-trip time to the `StorageServers`. Atomic operations also eliminate *read-write* conflicts with other atomic operations on the same data item (only *write-read* conflicts can still happen). This makes atomic operations ideal for operating on keys that are frequently modified, such as a key-value pair used as a counter. The set-versionstamp operation is another interesting optimization, which sets part of the key or part of the value to be the transaction's commit version. This enables client applications to later read back the commit version and has been used to improve the performance of client-side caching. In the FDB Record Layer [28], many aggregate indexes are maintained using atomic mutations, allowing concurrent, conflict-free updates, and the set-versionstamp operation is used to maintain low-contention synchronization indices.

3 Geo-replication and failover

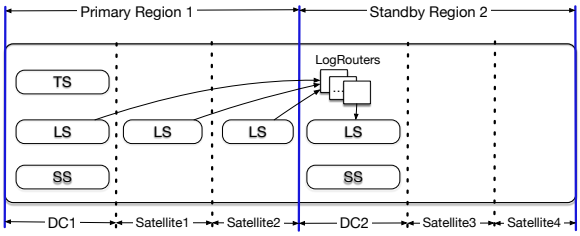


Figure 5: A two-region replication setup for an FDB cluster. Both regions have a data center and two satellite sites.

The main challenge of providing high availability during region failures is the trade-off of performance and consistency [12]. Synchronous cross-region replication provides strong consistency, but pays the cost of high latency. Conversely, asynchronous replication reduces latency by only persisting in the primary region, but may lose data when performing a region failover. FDB can be configured to perform either synchronous or asynchronous cross-region replication. However, there is a third possibility that leverages multiple availability zones within the same region, and provides a high level of failure independence, notwithstanding the unlikely event of a complete region outage. Our design (1) always avoids cross-region write latencies, as for asynchronous replication, (2) provides full transaction durability, like synchronous replication, so long as there is no simultaneous failure of multiple availability zones in a region, (3) can do rapid and completely automatic failover between regions, (4) can be manually failed-over with the same guarantees as asynchronous replication (providing A, C, and I of ACID but potentially exhibiting a Durability failure) in the unlikely case of a simultaneous total region failure, and (5) only requires full replicas of the database in the primary and secondary regions' main availability zones, not multiple replicas per region. The rest of this section is dedicated to this design.

Figure 5 illustrates the layout of a two-region replication of a cluster. Both regions have a data center (DC) as well as one or more satellite sites. Satellites are located in close proximity to the DC (in the same region) but are failure independent. The resource requirements from satellites are insignificant as they only need to store log replicas (i.e., a suffix of the redo logs), while data centers host LS, SS, and (when primary) the TS. Control plane replicas (i.e., coordinators) are deployed across three or more failure domains (in some deployments utilizing an additional region), usually with at least 9 replicas. Relying on majority quorums allows the control plane to tolerate one site (data center/satellite) failure and an additional replica failure.

A typical deployment configuration is illustrated in Figure 5, depicting two regions with a data center and two satellites in each region. One of the data centers (DC1), configured with a higher priority compared to DC2, is designated as the primary (its region

is denoted as the primary region, accordingly) and contains the full TS, LS, and SS; DC2 in the secondary region has replicas of data with its own LS and SS. Reads can be served from storage replicas at both primary and secondary data centers (consistent reads do require obtaining a read version from the primary data center). All client writes are forwarded to the primary region and processed by Proxies in DC1, then synchronously persisted onto `LogServers` in DC1 and one or both satellite sites in the primary region (depending on the configuration), avoiding the cross-region WAN latency. The updates are then asynchronously replicated to DC2, where they are stored on multiple LS servers and eventually spread out to multiple `StorageServers`. `LogRouters` implement a special type of FDB role that facilitates cross-region data transfer. They were created to avoid redundant cross-region transfers of the same information. Instead, `LogRouters` transfer each log entry across WAN only once, and then deliver it to all relevant LS servers locally in DC2.

The cluster automatically fails-over to the secondary region if the primary data center becomes unavailable. Satellite failures could, in some cases, also result in a fail-over, but this decision is currently manual. When the fail-over happens, DC2 might not have a suffix of the log, which it proceeds to recover from the remaining log server in the primary region. Next, we discuss several alternative satellite configurations which provide different levels of fault-tolerance.

Satellite configuration can be specified per region. Each satellite is given a static priority, which is considered relatively to other satellites in the same region. FDB is usually configured to store multiple log replicas at each location. Three main alternatives are supported: (1) synchronously storing updates on all log replicas at the satellite with the highest priority in the region. In this case, if the satellite fails, another satellite with the next priority is recruited for the task, (2) synchronously storing updates on all replicas of two satellites with the highest priorities in the region. In this case, if a satellite fails, it can be similarly replaced with a different satellite of lower priority, or, if none available, fall back to option (1) of using a single satellite. In either case, the secondary region isn't impacted, as it can continue to pull updates from remaining `LogServers` in the primary region. Finally, option (3) is similar to option (2) but FDB only waits for one of the two satellites to make the mutations durable before considering a commit successful. In all cases, if no satellites are available, only the `LogServers` in DC1 are used. With option 1 and 3, a single site (data center or satellite) failure can be tolerated, in addition to one or more `LogServer` failures (since the remaining locations have multiple log replicas). With option 2, two site failures in addition to one or more `LogServer` failures can be tolerated. In options 1 and 2, however, commit latency is sensitive to the tail network latencies between the primary data center and its satellites, which means that option 3 is usually faster. The choice ultimately depends on the number of available satellite locations, their connectivity to the data center and the desired level of fault tolerance and availability.

When DC1 in the primary region suddenly becomes unavailable, the cluster (with the help of `Coordinators`) detects the failure and starts a new transaction management system in DC2. New `LogServers` are recruited from satellites in the secondary region, in accordance with the region's replication policy. During recovery, `LogRouters` in DC2 may need to fetch the last few seconds' data from primary satellites, which, due to the asynchronous replication, may not have made it to DC2 prior to the failover. After the recovery, if the failures in Region 1 are healed and its replication policy can again be met, the cluster will automatically fail-back to have DC1 as the primary data center due to its higher priority. Alternatively, a different secondary region can be recruited.

4 Simulation Testing

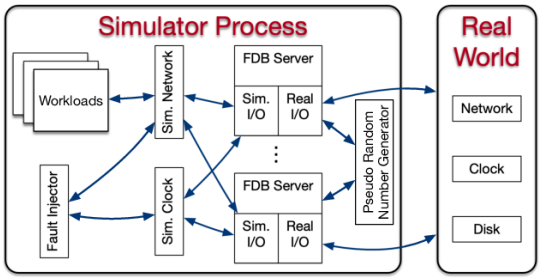


Figure 6: The FDB deterministic simulator.

Testing and debugging distributed systems is a challenging and inefficient process. This problem is particularly acute for FDB, which offers a very strong concurrency control contract, any failure of which can produce almost arbitrary corruption in systems layered on top. Accordingly, an ambitious approach to end-to-end testing was adopted from the beginning of FDB 's development: the real database software is run, together with randomized synthetic workloads and fault injection, in a deterministic discrete-event simulation. The harsh simulated environment quickly provokes bugs (including but not limited to distributed systems bugs) in the database, and determinism guarantees that every bug found this way can be reproduced, diagnosed, and fixed.

Deterministic simulator. FDB was built from the ground up to make this testing approach possible. All database code is deterministic; accordingly multithreaded concurrency is avoided (instead, one database node is deployed per core). Figure 6 illustrates the simulator process of FDB, where all sources of nondeterminism and communication are abstracted, including network, disk, time, and pseudo random number generator. FDB is written in Flow [4], a novel syntactic extension to C++ adding async/await-like concurrency primitives. Flow provides the Actor programming model [13] that abstracts various actions of the FDB server process into a number of actors that are scheduled by the Flow runtime library. The simulator process is able to spawn multiple FDB servers that communicate with each other through a simulated network in a single discrete-event simulation. The production implementation is a simple shim to the relevant system calls.

The simulator runs multiple workloads (also written in Flow) that communicate with simulated FDB servers through the simulated network. These workloads include fault injection instructions, mock applications, database configuration changes, and direct internal database functionality invocations. Workloads are composable to exercise various features and are reused to construct comprehensive test cases.

Test oracles. FDB uses a variety of test oracles to detect failures in simulation. Most of the synthetic workloads used in simulation

have assertions built in to verify the contracts and properties of the database (for example, by checking invariants in their data that can only be maintained through transaction atomicity and isolation). Assertions are used throughout the code-base to check properties that can be verified "locally". Properties like recoverability (eventual availability) can be checked by returning the modeled hardware environment (after a set of failures perhaps sufficient to break the database's availability) to a state in which recovery should be possible and verifying that the cluster eventually recovers.

Fault injection. The FDB simulator injects machine, rack, and data-center level fail-stop failures and reboots, a variety of network faults, partitions, and latency problems, disk behavior (e.g. the corruption of unsynchronized writes when machines reboot), and randomizes event times. This variety of fault injection both tests the database's resilience to specific faults and increases the diversity of states in simulation. Fault injection distributions are carefully tuned to avoid driving the system into a small state-space caused by an excessive fault rate.

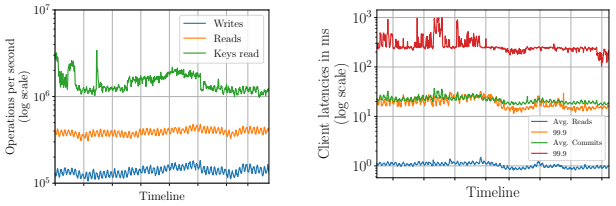
FDB itself cooperates with the simulation in making rare states and events more common, through a high-level fault injection technique informally referred to as "buggification". At many places in its code-base, the simulation is given the opportunity to inject some unusual (but not contract-breaking) behavior such as unnecessarily returning an error from an operation that usually succeeds, injecting a delay in an operation that is usually fast, choosing an unusual value for a tuning parameter, etc. This complements fault injection at the network and hardware level. Randomization of tuning parameters also ensures that specific performance tuning values do not accidentally become necessary for correctness.

Swarm testing [40] is extensively used to maximize the diversity of simulation runs. Each run uses a random cluster size and configuration, random workloads, random fault injection parameters, random tuning parameters, and enables and disables a different random subset of buggification points. We have open-sourced the swarm testing framework for FDB [7].

Conditional coverage macros are used to evaluate and tune the effectiveness of the simulation. For example, a developer concerned that a new piece of code may rarely be invoked with a full buffer can add the line `TEST( buffer.is_full() );` and analysis of simulation results will tell them how many distinct simulation runs achieved that condition. If the number is too low, or zero, they can add buggification, workload, or fault injection functionality to ensure that scenario is adequately tested.

Latency to bug discovery. Finding bugs quickly is important both so that they are encountered in testing before production, and for engineering productivity (since bugs found immediately in an individual commit can be trivially traced to that commit). Discrete-event simulation can run arbitrarily faster than real-time if CPU utilization within the simulation is low, as the simulator can fast-forward clock to the next event. Many distributed systems bugs take time to play out, and running simulations with long stretches of low utilization allows many more of these to be found per core second than in "real-world" end-to-end tests.

Additionally, bugs can be found faster simply by running more simulations in parallel. Randomized testing is embarrassingly parallel and FDB developers can and do "burst" the amount of testing they do immediately before major releases, in the hopes of catching



(a) Read and write operations.

(b) The average and 99.9-percentile latency of client read and commit requests.

Figure 7: Measurement from a production cluster for a month (hourly plots).

exceptionally rare bugs that have thus far eluded the testing process. Since the search space is effectively infinite, simply running more tests results in more code being covered and more potential bugs being found, in contrast to scripted functional or system testing.

Limitations. Simulation is not able to reliably detect performance issues, such as an imperfect load balancing algorithm. It is also unable to test third-party libraries or dependencies, or even first-party code not implemented in Flow. As a consequence, we have largely avoided taking dependencies on external systems. Finally, bugs in critical dependent systems, such as a filesystem or the operating system, or misunderstandings of their contract, can lead to bugs in FDB. For example, several bugs have resulted from the true operating system contract being weaker than it was believed to be.

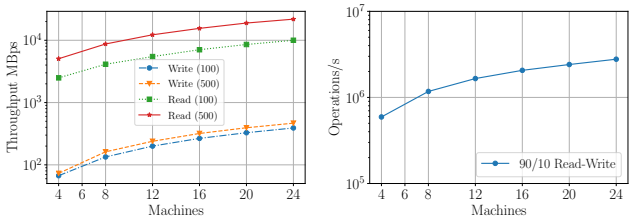
5 Evaluation

We first measure the performance of a production geo-replicated FDB cluster. Then we study the scalability of FDB. Finally, we provide some data on the time of reconfiguration.

5.1 Production Measurement

The measurement was taken from one of Apple's production geo-replicated cluster described in Section 3. This cluster consists of a total of 58 machines: both the primary DC and remote DC have 25 machines, and two satellites in the primary region contain 4 machines each. This cluster is configured to use one satellite (6.5 ms latency) for storing log data and utilize the other (65.2 ms latency) running `Coordinators`. The network latency between primary DC and remote DC is 60.6 ms. In total, there are 862 FDB processes running on these machines, with additional 55 spare processes reserved for emergency usage. The cluster stores 292 TB data with a total of 464 SSD disks (8 per machine). An SSD disk is bound to either one `LogServer` or two `StorageServer` processes to maximize I/O utilization. We measured client read operations at `StorageServers` and write (or commit) operations at `Proxies`.

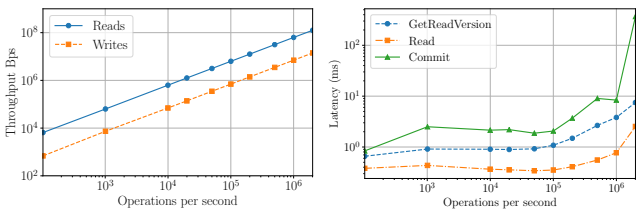
Traffic pattern. Figure 7a shows the traffic of the cluster in a month, which exhibits a clear diurnal pattern. The average numbers of read operations, write operations, and keys read are 390.4K, 138.5K, and 1.467M, respectively. The number of keys read is several times the number of reads because many of them are range reads that return values for multiple keys.



(a) Read and write throughput with 100 and 500 operations per transaction.

(b) 90/10 Read-write.

Figure 8: Scalability test.



(a) Throughput.

(b) Average latency.

Figure 9: Throughput and average latency for different operation rate on a 24-machine cluster configuration.

Latency. Figure 7b shows the average and 99.9-percentile of client read and commit latencies. For reads, the average and 99.9-percentile are about 1 and 19 ms, respectively. For commits, the average and 99.9-percentile are about 22 and 281 ms, respectively. The commit latencies are higher than read latencies because commits always write to multiple disks in both primary DC and one satellite. Note the average commit latency is lower than the WAN latency of 60.6 ms, due to asynchronous replication to the remote region. The 99.9-percentile latencies are an order of magnitude higher than the average, because they are affected by multiple factors such as the variability in request load, queue length, replica performance, and transaction or key value sizes. Due to the multi-tenancy nature of CloudKit [59], the average transaction conflict rate is 0.73% during the month.

Recovery and availability. In August 2020, there was only one transaction system recovery, which took 8.61 seconds. This corresponds to five 9s availability.

5.2 Scalability Test

The experiments were conducted on a test cluster of 27 machines in a single data center. Each machine has a 16-core 2.5 GHz Intel Xeon CPU with hyper-threading enabled, 256 GB memory, 8 SSD disks, connected via 10 Gigabit Ethernet. Each machine runs 14 `StorageServers` on 7 SSD disks and reserves the other one SSD for `LogServer`. In the experiments, we use the same number of `Proxies` and `LogServers`. The replication degrees for both `LogServers` and `StorageServers` are set to three.

We use a synthetic workload to evaluate the performance of FDB. Specifically, there are four types of transactions: 1) *blind writes* that update a configured number of random keys; 2) *range reads* that fetch a configured number of continuous keys starting at a random key; 3) *point reads* that fetch 10 random keys; and 4) *point writes* that fetch 5 random keys and update another 5 random keys. We use blind writes and range reads to evaluate the write and read performance, respectively. The point reads and point writes are used together to evaluate the mixed read-write performance. For instance, 90% reads and 10% writes (90/10 read-write) is constructed with 80% point reads and 20% point writes transactions. Each key is 16 bytes and the value size is uniformly distributed between 8 and 100 bytes (averaging 54 bytes). The database is pre-populated with data using the same size distribution. In the experiments, we make sure the dataset cannot be completely cached in the memory of `StorageServers`.

Figure 8 illustrates the scalability test of FDB from 4 to 24 machines using 2 to 22 `Proxies` or `LogServers`. Figure 8a shows that the write throughput scales from 67 to 391 MBps (5.84X) for 100 operations per transaction (T100), and from 73 to 467 MBps (6.40X) for 500 operations per transaction (T500). Note the raw write throughput is three times higher, because each write is replicated three times to `LogServers` and `StorageServers`. `LogServers` are CPU saturated at the maximum write throughput. Read throughput increases from 2,946 to 10,096 MBps (3.43X) for T100, and from 5055 to 21,830 MBps (4.32X) for T500, where `StorageServers` are saturated. For both reads and writes, increasing the number operations in a transaction boosts throughput. However, increasing operations further (e.g. to 1000) doesn't bring significant changes. Figure 8b shows the operations per second for 90/10 read-write traffic, which increases from 593k to 2,779k (4.69X). In this case, `Resolvers` and `Proxies` are CPU saturated.

The above experiments study saturated performance. Figure 9 illustrates the client performance on a 24-machine cluster with varying operation rate of 90/10 read-write load. This configuration has 2 `Resolvers`, 22 `LogServers`, 22 `Proxies`, and 336 `StorageServers`. Figure 9a shows that the throughput scales linearly with more operations per second (Ops) for both reads and writes. For latency, Figure 9b shows that when Ops is below 100k, the mean latencies remain stable: about 0.35ms to read a key, 2ms to commit, and 1ms to get a read version (GRV). Read is a single hop operation, thus is faster than the two-hop GRV request. The commit request involves multiple hops and persistence to three `LogServers`, thus higher latency than reads and GRVs. When Ops exceeds 100k, all these latencies increase because of more queuing time. At 2m Ops, `Resolvers` and `Proxies` are saturated. Batching helps to sustain the throughput, but commit latency spike to 368ms due to saturation.

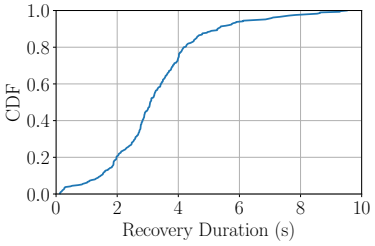


Figure 10: CDF plot for reconfiguration duration in seconds.

5.3 Reconfiguration Duration

We collected 289 reconfiguration (i.e., transaction system recovery) traces from our production clusters that typically host hundreds of TBs data. Because of the client-facing nature, short reconfiguration time is critical for the high availability of these clusters. Figure 10 illustrates the cumulative distribution function (CDF) of the reconfiguration times. The median and 90-percentile are 3.08 and 5.28 seconds, respectively. The reason for these short recovery times is that they are not bounded by the data or transaction log size, and are only related to the system metadata sizes. During the recovery, read-write transactions were temporarily blocked and were retried after timeout. However, client reads were not impacted, because they are served by `StorageServers`. As a result, clients were often unaware the cluster was unavailable for a few seconds. The causes of these reconfigurations include automatic failure recovery from software or hardware faults, software upgrades, database configuration changes, and the manual mitigation of production issues by the Site Reliability Engineering (SRE) team.

6 Lessons Learned

FDB has been under active development since 2009 and became an open source project under Apache license in 2018 [5]. This section discusses our experience and lessons of FDB.

6.1 Architecture Design

The divide-and-conquer design principle has proven to be an enabling force for flexible cloud deployment, making the database extensible as well as performant. First, separating the transaction system from the storage layer enables greater flexibility in placing and scaling compute and storage resources independently. Further, operators are free to place heterogeneous roles of FDB on different server instance types, optimizing for performance and costs. Second, the decoupling design makes it possible to extend the database functionality, such as our ongoing work of supporting RocksDB [38] as a drop-in replacement for the current SQLite engine. Finally, many of the recent performance improvements are specializing functionality as dedicated roles, e.g., separating `DataDistributor` and `Ratekeeper` from `Sequencer`, adding storage cache, dividing `Proxies` into get-read-version proxy and commit proxy. This design pattern successfully allows new features and capabilities to be added frequently.

6.2 Simulation Testing

Simulation testing has enabled FDB to maintain a very high development velocity with a small team by shortening the latency between a bug being introduced and a bug being found, and by allowing deterministic reproduction of issues. Adding additional logging, for instance, generally does not affect the deterministic ordering of events, so an exact reproduction is guaranteed. The productivity of this debugging approach is so much higher than normal production debugging, that in the rare circumstances when a bug was first found "in the wild", the debugging process was almost always first to improve the capabilities or the fidelity of the simulation until the issue could be reproduced there, and only then to begin the normal debugging process.

Rigorous correctness testing via simulation makes FDB extremely reliable. In the past several years, CloudKit [59] has deployed FDB for more than 0.5M disk years without a single data corruption event. Additionally, we constantly perform data consistency checks by comparing replicas of data records and making sure they are the same. To this date, no inconsistent data replicas have ever been found in our production clusters.

What is hard to measure is the productivity improvements stemming from increased confidence in the testability of the system. On numerous occasions, the FDB team executed ambitious, ground-up rewrites of major subsystems. Without simulation testing, many of these projects would have been deemed too risky or too difficult, and not even attempted.

The success of simulation has led us to continuously push the boundary of what is amenable to simulation testing by eliminating dependencies and reimplementing them ourselves in Flow. For example, early versions of FDB depended on Apache Zookeeper for coordination, which was deleted after real-world fault injection found two independent bugs in Zookeeper (circa 2010) and was replaced by a de novo Paxos implementation written in Flow. No production bugs have ever been reported since.

6.3 Fast Recovery

Fast recovery is not only useful for improving availability, but also greatly simplifies the software upgrades and configuration changes and makes them faster. Traditional wisdom of upgrading a distributed system is to perform rolling upgrades so that rollback is possible when something goes wrong. The duration of rolling upgrades can last from hours to days. In contrast, FoundationDB upgrades can be performed by restarting all processes at the same time, which usually finishes within a few seconds. Because this upgrade path has been extensively tested in simulation, all upgrades in Apple's production clusters are performed in this way. Additionally, this upgrade path simplifies protocol compatibility between different versions—we only need to make sure on-disk data is compatible. There is no need to ensure the compatibility of RPC protocols between different software versions.

An interesting discovery is that fast recovery sometimes can automatically heal latent bugs, which is similar to software rejuvenation [42]. For instance, after we separated the `DataDistributor` role from the `Sequencer`, we were surprised to discover several unknown bugs in the `DataDistributor`. This is because before the change, `DataDistributor` is restarted with `Sequencer`, which effectively reinitializes and heals the states of the `DataDistributor`. After the separation, we made `DataDistributor` a long running process independent of transaction system recovery (including `Sequencer` restart). As a result, the erroneous states of the `DataDistributor` are never healed and cause test failures.

6.4 5s MVCC Window

FDB chooses a 5-second MVCC window to limit the memory usage of the transaction system and storage servers, because the multi-version data is stored in the memory of `Resolvers` and `StorageServers`, which in turn restricts transaction sizes. From our experience, this 5s window is long enough for the majority of OLTP use cases. If a transaction exceeds the time limit, it is often the case that the client application is doing something inefficient, e.g., issuing reads one by one instead of parallel reads. As a result, exceeding the time limit often exposes inefficiency in the application.

For some transactions that may span more than 5s, many can be divided into smaller transactions. For instance, the continuous backup process of FDB will scan through the key space and create snapshots of key ranges. Because of the 5s limit, the scanning process is divided into a number of smaller ranges so that each range can be performed within 5s. In fact, this is a common pattern: one transaction creates a number of jobs and each job can be further divided or executed in a transaction. FDB has implemented such a pattern in an abstraction called `TaskBucket` and the backup system heavily depends on it.

7 Related Work

Key/value stores and NoSQL systems. BigTable [25], Dynamo [33], PNUTS [30], MongoDB [10], CouchDB [2], Cassandra [45] do not provide ACID transactions. RAMCloud [54] is an in-memory key value system that only supports single-object transactions. Google's Percolator [55], Apache Tephra [11], and Omid [20, 58] layered transactional APIs atop key value stores with snapshot isolation. FDB supports strict serializable ACID transactions on a scalable key-value store that has been used to support flexible schema and richer queries [6, 28, 47]. Similar SQL-over-NoSQL architecture has been adopted in Hyder [19], Tell [49], and AIM [21].

Concurrency Control. Many systems [26, 31, 35, 36, 48, 50, 51, 55, 62] use the time of acquiring all locks to establish the serial order among transactions and to guarantee atomicity and isolation. For instance, Spanner [31] uses True-time for determining the commit timestamps when all locks are acquired. CockroachDB [62] uses the hybrid-logical clock that is a combination of physical and logical time. Like FDB, a number of other systems [16, 19, 22, 34, 37, 58, 64] order transactions without locks. H-Store [61], Calvin [64], Hekaton [34], and Omid [58] execute transactions in timestamp order. Hyder [19], Tango [16], and ACID-RAIN [37] use a shared log to establish ordering. Sprint [22] orders transactions with total-order multicast. FDB provides strict serializability by a lock-free concurrency control combining MVCC and OCC. The serial order is determined by a `Sequencer`.

Unbundled database systems. These databases separate transaction component (TC) from data component (DC) [11, 20, 23, 48–51, 58, 66, 67, 71]. Deuteronomy [48] creates virtual resources that can be logically locked in the transaction system, where a DC knows nothing about transactions, their commit or abort. Solar [71] combines the benefits of scalable storage on a cluster of nodes with a single server for transaction processing. Amazon Aurora [66, 67] simplifies database replication and recovery with shared storage. These systems use lock-based concurrency control. Tell [49] uses advanced hardware to achieve high performance and implements snapshot isolation with a distributed MVCC protocol, while FDB uses commodity hardware with serializable isolation. In FDB, TC is further decomposed into a number of dedicated roles and the transaction logging is decoupled from TC. As a result, FDB chooses a lock-free concurrency management with a deterministic transaction order.

Bundled database systems. Traditional database systems have tight coupling of the transaction component and data component. Silo [65] and Hekaton [34] have achieved high throughput using a single server for transaction processing. Many distributed databases partition data to scale out [24, 26, 31, 35, 36, 43, 61, 62, 64]. Among these systems, FaRM [35, 36], and DrTM [26] exploit advanced hardware to improve transaction performance. FDB adopts an unbundled design with commodity hardware in mind.

Recovery. Traditional database systems often choose to implement a recovery protocol based on ARIES [53]. VoltDB [52] uses command logging so that recovery starts from a checkpoint and replays the commands in the log. NVRAM devices have been used to reduce recovery time in [15, 36]. Amazon Aurora [66] decouples redo log processing from the database engine by leveraging smart storage, and only undo log is processed by the database engine. RAMCloud [54] recovers in parallel redo logs on multiple machines. The recovery time for these systems is proportional to the log size. In comparison, FDB completely decouples redo and undo log processing from the recovery by a mindful separation of log servers and storage servers.

Testing. Non-deterministic fault-injection has been widely used in the testing of distributed systems. Various approaches have included faults such as network partition [14], power failures [70], and storage faults [39]. Jepsen [3] is a commercial effort that has tested a large number of distributed databases. All of these approaches lack deterministic reproducibility. Model checking has also been widely-applied to distributed systems [46, 69]. While model checking can be more exhaustive than simulation, it can only verify the correctness of a model rather than of the actual implementation. Finally there are numerous approaches to testing the correctness of database subsystems in the absence of faults, including the query engine [17, 57, 60] and concurrency control mechanism [63]. The FDB deterministic simulation approach allows verification of database invariants and other properties against the real database code, together with deterministic reproducibility.

8 Conclusions

This paper presents FoundationDB, a key value store designed for OLTP cloud services. The main idea is to decouple transaction processing from logging and storage. Such an unbundled architecture enables the separation and horizontal scaling of both read and write handling. The transaction system combines OCC and MVCC to ensure strict serializability. The decoupling of logging and the determinism in transaction orders greatly simplify recovery by removing redo and undo log processing from the critical path, thus allowing unusually quick recovery time and improving availability. Finally, deterministic and randomized simulation has ensured the correctness of the database implementation. Our evaluation and experience with cloud workloads demonstrate FoundationDB can meet challenging business requirements.

Acknowledgment

We thank Ori Herrnstadt, Yichi Chiang, Henry Gray, Maggie Ma, and all past and current contributors to the FoundationDB project, as well as our SRE team, John Brownlee, Joshua McManus, Leonidas Tsampros, Shambugouda Annigeri, Tarun Chauhan, Veera Prasad Battina, Amrita Singh, and Swetha Mundla at Apple. We thank anonymous reviewers, Nicholas Schiefer, and Zhe Wu for their helpful comments, and particularly our shepherd, Eliezer Levy.

References

[1] Couchbase. https://www.couchbase.com/.

[2] CouchDB. https://couchdb.apache.org/.

[3] Distributed system safety research. https://jepsen.io/.

[4] Flow. https://github.com/apple/foundationdb/tree/master/flow.

[5] FoundationDB. https://github.com/apple/foundationdb.

[6] FoundationDB Document Layer. https://github.com/FoundationDB/fdb-document-layer.

[7] FoundationDB Joshua. https://github.com/FoundationDB/fdb-joshua.

[8] Foundationdb storage adapter for janusgraph. https://github.com/JanusGraph/janusgraph-foundationdb.

[9] Janusgraph. https://janusgraph.org/.

[10] MongoDB. https://www.mongodb.com/.

[11] Tephra: Transactions for Apache HBase. http://tephra.incubator.apache.org/.

[12] D. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, Feb. 2012.

[13] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[14] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 51–68, Carlsbad, CA, USA, 2018. USENIX Association.

[15] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *Proceedings of the VLDB Endowment*, 10:337–348, November 2016.

[16] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340, Farminton, PA, 2013.

[17] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna. A genetic approach for random testing of database systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1243–1251, Vienna, Austria, 2007. VLDB Endowment.

[18] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[19] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *CIDR*, 2011.

[20] E. Bortnikov, E. Hillel, I. Keidar, I. Kelly, M. Morel, S. Paranjpye, F. Perez-Sorrosal, and O. Shacham. Omid, reloaded: Scalable and highly-available transaction processing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 167–180, Santa Clara, CA, Feb. 2017. USENIX Association.

[21] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. Liang. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 251–264, New York, NY, USA, 2015. Association for Computing Machinery.

[22] L. Camargos, F. Pedone, and M. Wieloch. Sprint: A middleware for high-performance transaction processing. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 385–398, Lisbon, Portugal, 2007.

[23] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma. Polarfs: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, Aug. 2018.

[24] S. Chandrasekaran and R. Bamford. Shared cache - the future of parallel databases. In *Proceedings 19th International Conference on Data Engineering*, pages 840–850, 2003.

[25] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[26] H. Chen, R. Chen, X. Wei, J. Shi, Y. Chen, Z. Wang, B. Zang, and H. Guan. Fast in-memory transaction processing using rdma and htm. *ACM Transactions on Computer Systems*, 35(1), July 2017.

[27] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC'02, pages 78–87, New York, NY, USA, 2002. ACM.

[28] C. Chrysafis, B. Collins, S. Dugas, J. Dunkelberger, M. Ehsan, S. Gray, A. Grieser, O. Herrnstadt, K. Lev-Ari, T. Lin, M. McMahon, N. Schiefer, and A. Shraer. FoundationDB Record Layer: A Multi-Tenant Structured Datastore. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD'19, pages 1787–1802, New York, NY, USA, 2019. ACM.

[29] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *Presented as part of the 2013 USENIX Annual Technical Conference*, pages 37–48, San Jose, CA, 2013. USENIX.

[30] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, Aug. 2008.

[31] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation*, pages 261–264, Hollywood, CA, Oct. 2012. USENIX Association.

[32] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[33] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[34] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, New York, NY, USA, 2013.

[35] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, pages 401–414, Seattle, WA, USA, April 2014.

[36] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, New York, NY, USA, 2015. Association for Computing Machinery.

[37] I. Eyal, K. P. Birman, I. Keidar, and R. van Renesse. Ordering transactions with prediction in distributed object stores. In *LADIS*, 2013.

[38] Facebook. Rocksdb. https://rocksdb.org.

[39] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 149–165, Santa clara, CA, USA, 2017. USENIX Association.

[40] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 78–88, Minneapolis, MN, USA, 2012. ACM.

[41] R. D. Hipp. SQLite. https://www.sqlite.org/index.html, 2020.

[42] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: analysis, module and applications. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 381–390, 1995.

[43] J. W. Josten, C. Mohan, I. Narang, and J. Z. Teng. Db2's use of the coupling facility for data sharing. *IBM Systems Journal*, 36(2):327–351, 1997.

[44] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[45] A. Lakshman and P. Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of Distributed Computing*, page 5, Jan 2009.

[46] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 399–414, Broomfield, CO, 2014. USENIX Association.

[47] K. Lev-Ari, Y. Tian, A. Shraer, C. Douglas, H. Fu, A. Andreev, K. Beranek, S. Dugas, A. Grieser, and J. Hemmo. Quick: a queuing system in cloudkit. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD Conference 2021, Xian, Shanxi, China, June 03 — 05, 2021*. ACM, 2021.

[48] J. Levandoski, D. Lomet, , and K. K. Zhao. Deuteronomy: Transaction support for cloud data. In *Conference on Innovative Data Systems Research (CIDR)*. www.crdrdb.org, January 2011.

[49] S. Loesing, M. Pilman, T. Etter, and D. Kossmann. On the design and scalability of distributed shared-data databases. In *SIGMOD Conference*, pages 663–676. ACM, 2015.

[50] D. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *Fourth Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, Jan. 2009.

[51] D. Lomet and M. F. Mokbel. Locking key ranges with unbundled transaction services. *Proceedings of the VLDB Endowment*, 2(1):265–276, August 2009.

[52] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615, Chicago, IL, USA, March 2014.

[53] C. Mohan, D. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[54] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 29–41, New York, NY, USA, 2011. Association for Computing Machinery.

[55] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'10, pages 251–264, Vancouver, BC, Canada, 2010.

[56] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[57] M. Rigger and Z. Su. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 667–682. USENIX Association, Nov. 2020.

[58] O. Shacham, Y. Gottesman, A. Bergman, E. Bortnikov, E. Hillel, and I. Keidar. Taking omid to the clouds: Fast, scalable transactions for real-time cloud analytics. *Proceedings of the VLDB Endowment*, 11(12):1795–1808, Aug. 2018.

[59] A. Shraer, A. Aybes, B. Davis, C. Chrysafis, D. Browning, E. Krugler, E. Stone, H. Chandler, J. Farkas, J. Quinn, J. Ruben, M. Ford, M. McMahon, N. Williams, N. Favre-Felix, N. Sharma, O. Herrnstadt, P. Seligman, R. Pisolkar, S. Dugas, S. Gray, S. Lu, S. Harkema, V. Kravtsov, V. Hong, Y. Tian, and W. L. Yih. Cloudkit: Structured storage for mobile applications. *Proceedings of the VLDB Endowment*, 11(5):540–552, Jan. 2018.

[60] D. R. Slutz. Massive stochastic testing of SQL. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 618–622. Morgan Kaufmann, 1998.

[61] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[62] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. Cockroachdb: The resilient geo-distributed SQL database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD*, pages 1493–1509, Portland, OR, USA, June 2020. ACM.

[63] C. Tan, C. Zhao, S. Mu, and M. Walfish. Cobra: Making transactional key-value stores verifiably serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 63–80. USENIX Association, Nov. 2020.

[64] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–12, May 2012.

[65] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, Farminton, PA, 2013.

[66] A. Verbitski, X. Bao, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, and T. Kharatishvili. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1041–1052, Chicago, IL, USA, May 2017.

[67] A. Verbitski, A. Gupta, D. Saha, J. Corey, K. Gupta, M. Brahmadesam, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvilli, and X. Bao. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 789–796, 2018.

[68] M. Yabandeh and D. Gómez Ferro. A Critique of Snapshot Isolation. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys'12, pages 155–168, Bern, Switzerland, 2012.

[69] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 213–228, Boston, Massachusetts, 2009. USENIX Association.

[70] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 449–464, Broomfield, CO, 2014. USENIX Association.

[71] T. Zhu, Z. Zhao, F. Li, W. Qian, A. Zhou, D. Xie, R. Stutsman, H. Li, and H. Hu. Solar: Towards a shared-everything database on distributed log-structured storage. In *USENIX Annual Technical Conference*, pages 795–807. USENIX Association, 2018.