DOI:10.1145/3592838

To view the accompanying Technical Perspective, visit doi.acm.org/10.1145/3592837

FoundationDB: A Distributed Key-Value Store

By 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, Yang Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav

Abstract

FoundationDB is an open-source transactional key-value store created more than 10 years ago. It is one of the first systems to combine the flexibility and scalability of NoSQL architectures with the power of ACID transactions. 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 scalability, high availability, and fault tolerance. FoundationDB includes a deterministic simulation framework, used to test every new feature 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 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.

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 (FDB)[3] 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 key space. 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 them. Instead, The NoSQL model leaves application developers with great flexibility. Applications can manage data stored as simple key-value pairs, but at the same time implement advanced features, such as consistent secondary indices and referential integrity checks.[10] FDB defaults to strictly serializable transactions but 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 is evidence of the usefulness of this unusual design. For example, the FoundationDB Record Layer[10] adds back much of what users expect from a relational database, and JanusGraph,[6] a graph database, provides an implementation as a FoundationDB layer.[5] In its newest release, CouchDB[1] (arguably the first NoSQL system) is being rebuilt 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

The original version of this paper, entitled "FoundationDB: A Distributed Unbundled Transactional Key Value Store," was published in SIGMOD '21, June 20–25, 2021, Virtual Event, China; https://doi.org/10.1145/3448016.3457559

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,[18] which only happen when multiple crashes or restarts occur in a particular sequence, pose a challenge even for end-to-end testing infrastructure. 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. A syntactic extension to C++, called Flow,[2] was created specifically for this purpose. This rigorous testing in simulation makes FDB extremely stable and allows its developers to introduce new features and releases in a rapid cadence.

FDB adopts an unbundled architecture[19] that comprises a control plane and a data plane. The control plane manages the metadata of the cluster and uses Active Disk Paxos[9] 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)[17] and multi-version concurrency control (MVCC).[8] 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.

This paper makes three primary contributions. First, we describe an open-source distributed storage system, FoundationDB, combining NoSQL and ACID, used in production at Apple, Snowflake, and other companies. Second, an integrated deterministic simulation framework makes FoundationDB one of the most stable systems of its kind. Third, we describe a unique architecture and approach to transaction processing, fault tolerance, and high availability.

2. DESIGN

The main design principles of FDB are:

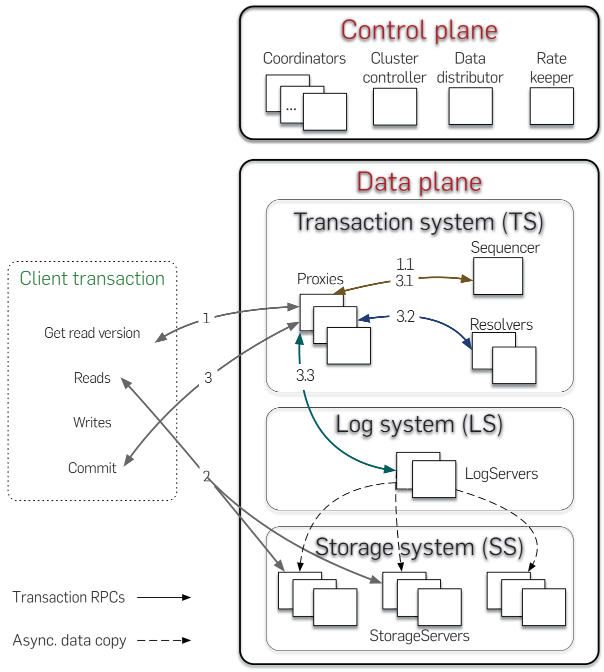
- *Divide-and-Conquer (or separation of concerns).* FDB decouples the transaction management system (write path) from the distributed storage (read path) and scales them independently. Within the transaction management system, processes are assigned various roles representing different aspects of transaction management. Furthermore, cluster-wide orchestrating tasks, such as overload control and load balancing 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. To cope with failures in the transaction management system of FDB, we handle all failures through the recovery path: the transaction system proactively shuts down when it detects a failure. Thus, all failure handling is reduced to a single recovery operation, which becomes a common and well-tested code path. To improve availability, FDB strives to minimize Mean-Time-To-Recovery (MTTR). In our production clusters, the total time is usually less than five seconds.
- *Simulation testing.* FDB relies on a randomized, deterministic simulation framework for testing the correctness of its distributed database. Simulation tests not only expose deep bugs,[18] but also boost developer productivity and the code quality of FDB.

2.1. 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.

Control plane. The control plane is responsible for persisting critical system metadata, that is, the configuration of transaction systems, on `Coordinators`. These `Coordinators` form a Paxos group[9] and elect a `ClusterController`. The `ClusterController` monitors all servers in the cluster and recruits three processes, `Sequencer` (described in Section 2.1.2), `DataDistributor`, and `Ratekeeper`, which are re-recruited if they fail or crash. The `DataDistributor` is responsible for monitoring failures and balancing data

Figure 1. Architecture and transaction processing.



among StorageServers. Ratekeeper provides overload protection for the cluster.

Data plane. FDB targets OLTP workloads that are read-mostly, read and write a small set of keys per transaction, have low contention, and require scalability. FDB chooses an unbundled architecture[19]: a distributed transaction management system (TS) consists of a Sequencer, Proxies, and Resolvers, all of which are stateless processes. 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 LS contains a set of LogServers and the SS has a number of StorageServers. This scales well with Apple's largest transactional workloads.[10]

The Sequencer assigns a read and a commit version to each transaction. Proxies offer MVCC read versions to clients and orchestrate transaction commits. Resolvers check for conflicts among transactions. LogServers act as replicated, sharded, distributed persistent queues, each queue storing WAL data for a StorageServer.

The SS consists of a number of StorageServers, each storing a set of data shards, that is, contiguous key ranges, and serving client reads. 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 an enhanced version of SQLite,[15] with enhancements that make range clears faster, defer deletion to a background task, and add support for asynchronous programming.

Read-write separation and scaling. As mentioned above, processes are assigned different roles; FDB scales by adding new processes for each role. Clients read from sharded StorageServers, so reads scale linearly with the number of StorageServers. Writes are scaled by adding more Proxies, Resolvers, and LogServers. The control plane's singleton processes (e.g., ClusterController and Sequencer) and Coordinators are not performance bottlenecks; they only perform limited metadata operations.

Bootstrapping. FDB has no dependency on external coordination services. All user data and most system metadata (keys that start with 0xFF prefix) are stored in StorageServers. The metadata about StorageServers is persisted in LogServers, and the LogServers configuration data is stored in all Coordinators. The Coordinators are a disk Paxos group; servers attempt to become the ClusterController if one does not exist. A newly elected ClusterController reads the old LS configuration from the Coordinators and spawns a new TS and LS. Proxies recover system metadata from the old LS, including information about all StorageServers. The Sequencer waits until the new TS finishes recovery (Section 2.2.4), then writes the new LS configuration to all Coordinators. The new transaction system is then ready to accept client transactions.

Before 7.1 release, the ClusterController delegates this work to the new Sequencer.

Reconfiguration. The Sequencer process monitors the health of Proxies, Resolvers, and LogServers. Whenever there is a failure in the TS or LS, or the database configuration changes, the Sequencer terminates. The ClusterController detects the Sequencer failure, then recruits and bootstraps a new TS and LS. In this way, transaction processing is divided into epochs, where each epoch represents a generation of the transaction management system with its own Sequencer.

2.2. Transaction management

This section describes end-to-end transaction processing and strict serializability, then discusses logging and recovery.

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 at least as large as all previously issued transaction commit versions, and sends this read version back to the client. The client may then issue reads to StorageServers and obtain values at that specific read version. Client writes are buffered locally without contacting the cluster and read-your-write semantics are preserved by combining results from database look-ups with uncommitted writes of the transaction. 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. If the transaction cannot commit, the client may choose to restart it.

A Proxy commits a client transaction in three steps. First, it 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. 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, that is, concurrent writes will not conflict with snapshot reads.

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 the `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 a Log Sequence Number (LSN). Because T_x observes the results of all previously committed transactions, FDB achieves strict serializability. To ensure there are no gaps between LSNs, the `Sequencer` returns the previous commit version (i.e., previous LSN) with each commit version. A `Proxy` sends both LSN and the 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 LSN order.

`Resolvers` use a lock-free conflict detection algorithm similar to *write-snapshot isolation*,[22] with the difference that in FDB the commit version is chosen before conflict detection. This allows FDB to efficiently batch-process both version assignments and conflict detection.

The entire key space is divided among `Resolvers` allowing conflict detection to 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 potentially committed transactions, 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, such false positives are limited to only happen within the short MVCC window time (i.e., 5 seconds).

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 is wasted work done by aborted transactions. In our multitenant production workload transaction conflict rate is very low (less than 1%) and OCC works well. If a conflict happens, the client can simply restart the transaction.

Logging protocol. After a `Proxy` decides to commit a transaction, it sends a message to all `LogServers`: mutations are sent to `LogServers` responsible for the modified key ranges, while other `LogServers` receive an empty message body. The log message header includes both the current and previous LSN obtained from the `Sequencer`, as well as the largest 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` apply non-durable redo logs from `LogServers` to an in-memory index. In the common case, this happens before any read versions that reflect the commit is handed out to a client, allowing very low latency for serving multi-version reads. Therefore, when client read requests reach `StorageServers`, the requested version (i.e., the latest committed data) is usually already available. If fresh data is not available to read at a `StorageServer` replica, the client either waits for the data to become available or reissues the request at another replica.[12] If both read time out, the client can simply restart the transaction.

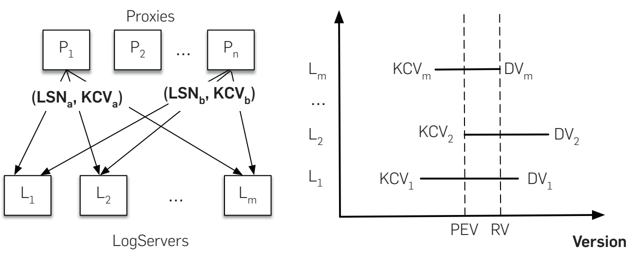
Since log data is already durable on `LogServers`, `StorageServers` can buffer updates in memory and persist batches of data to disks periodically, thus improving I/O efficiency.

Transaction system recovery. Traditional database systems often employ the ARIES recovery protocol.[20] During recovery, the system processes redo log records from the last checkpoint by re-applying them to relevant data pages. This brings the database to a consistent state; transactions that were in flight during the crash can be rolled back by executing the undo log records.

In FDB, recovery is purposely made very cheap—there is no need to apply undo log entries. This is possible because of a greatly simplifying design choice: redo log processing is the same as the normal log forward path. In FDB, `StorageServers` pull logs from `LogServers` and apply them in the background. The recovery process starts by detecting a failure and recruiting a new transaction system. The new TS can accept transactions before all the data in the old `LogServers` is processed. Recovery only needs to find out the end of the redo log: At that point (as in normal forward operation) `StorageServers` asynchronously replay the log.

For each epoch, the `ClusterController` executes recovery in several steps. First, it reads the previous TS configuration from `Coordinators` and locks this information to prevent another concurrent recovery. Next, it recovers previous TS system states, including information about older `LogServers`, stops them from accepting transactions, and recruits a new set of `Sequencer`, `Proxies`, `Resolvers`, and `LogServers`. After previous `LogServers` are stopped and a new TS is recruited, the `ClusterController` writes the new TS information to the `Coordinators`. Because `Proxies` and `Resolvers` are stateless, their

Figure 2. An illustration of RV and PEV.



recoveries have no extra work. In contrast, LogServers save the logs of committed transactions, and we need to ensure all such transactions are durable and retrievable by StorageServers.

The essence of the recovery of old LogServers is to determine the end of the redo log, that is, a Recovery Version (RV). Rolling back undo the log is essentially discarding any data after RV in the old LogServers and StorageServers. Figure 2 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, along with the LSN of the current transaction. Each LogServer keeps the maximum KCV received and a Durable Version (DV), which is the maximum LSN persisted by the LogServer (DV is ahead of KCV since it corresponds to in-flight transactions). During 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, 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 the 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 the 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 of the RV so that they can roll back any data larger than the RV. The current FDB storage engine consists of an unversioned SQLite[15] 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.

2.3. 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.[9] 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 LogServer results in a transaction system recovery.
- *Storage replication.* Every shard, that is, a key range, is asynchronously replicated to $k = f + 1$ StorageServers, which 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.

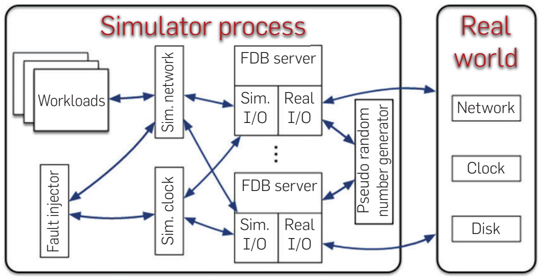
Note the storage team abstraction is more sophisticated than Copysets.[11] To reduce the chance of data loss due to simultaneous failures, FDB ensures that at most one process in a replica group is placed in a *fault domain*, for example, a host, rack, or availability zone. Each team is guaranteed to have at least one process live and there is no data loss if any one of the respective fault domains remains available.

3. SIMULATION TESTING

Testing and debugging distributed systems is a challenging and inefficient process. This problem is particularly acute for FDB—any failure of its strong concurrency control contract can produce almost arbitrary corruption in systems layered on top. Accordingly, an ambitious approach to end-to-end testing was adopted from the beginning: 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 in the database, and determinism guarantees that every such bug can be reproduced and investigated.

Deterministic simulator. FDB was built from the ground up to enable this testing approach. All database code is deterministic and multithreaded concurrency is avoided (instead, one database node is deployed per core). Figure 3 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,[2] a novel syntactic extension to C++ adding async/await-like concurrency primitives with automatic cancellation, permitting highly concurrent code to execute deterministically. Flow provides the Actor programming model[7] 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.

Figure 3. The FDB deterministic simulator.



The simulator runs multiple workloads (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 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 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 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. Simulation injects machine, rack, and data-center 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 allowed 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, or choosing an unusual value for a tuning parameter, etcetera. This complements fault injection at the network and hardware levels. Randomization of tuning parameters also ensures that specific performance tuning values do not accidentally become necessary for correctness.

Swarm testing[14] 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 random subset of buggification points. We have open-sourced the swarm testing framework for FDB.[4]

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, randomized testing is embarrassingly parallel and FDB developers can and do "burst" the amount of testing they do before major releases, in the hopes of catching exceptionally rare bugs that have thus far eluded the testing process. Since the search space is effectively infinite, 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 cannot 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.

4. EVALUATION

This section studies the scalability of FDB and provides some data on the time of reconfiguration.

4.1. 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 remaining 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. Point reads and point writes are used together to evaluate 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 4 illustrates the scalability test of FDB from 4 to 24 machines using 2 to 22 Proxies or LogServers. Figure 4a 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 2946 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 of operations in a transaction boosts throughput. However, increasing operations further (e.g., to 1000) doesn't bring significant changes. Figure 4b shows the operations per second for 90/10 read-write traffic, which increases from 593k to 2779k (4.69X). In this case, Resolvers and Proxies are CPU saturated.

The above experiments study saturated performance. Figure 5 illustrates the client performance on a 24-machine cluster with varying operation rates of 90/10 read-write load. This configuration has 2 Resolvers, 22 LogServers, 22 Proxies, and 336 StorageServers. Figure 5a shows that the throughput scales linearly with more operations per second (Ops) for both reads and writes. For latency, Figure 5b shows that when Ops is below 100k, the mean latencies remain stable: about 0.35 ms to read a key, 2 ms 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 100 k, all these latencies increase because of more queuing time. At 2m Ops, Resolvers and Proxies are saturated. Batching helps to sustain the throughput but commits latency spike to 368 ms due to saturation.

4.2. Reconfiguration duration

We collected 289 reconfigurations (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 6 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 the timeout. However, client reads were not impacted. 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.

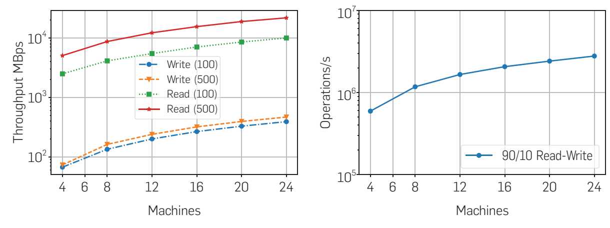
5. LESSONS LEARNED

This section discusses our experience and lessons of FDB.

5.1. 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. An added benefit of LogServers is that they are akin to witness replicas; in some of our multi-region production deployments, LogServers significantly reduce the number of StorageServers (full replicas) required to achieve the same high-availability properties. Further, operators are free to place heterogeneous roles of FDB on different server instance types, optimizing for performance and costs.

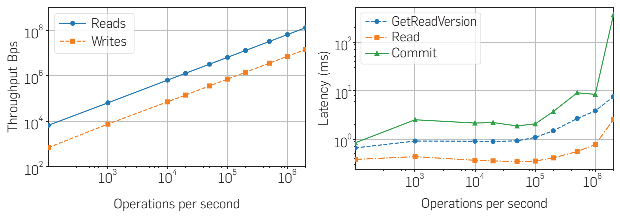
Figure 4. Scalability test.



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

(b) 90/10 Read-write.

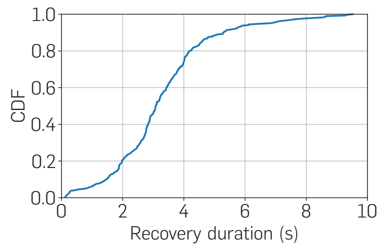
Figure 5. Throughput and average latency for different operation rates on a 24-machine cluster configuration.



(a) Throughput.

(b) Average latency.

Figure 6. CDF plot for reconfiguration duration in seconds.



Second, the decoupling design makes it possible to extend the database functionality, such as our ongoing work of supporting RocksDB[13] as a drop-in replacement for the current SQLite engine. Finally, many of the recent performance improvements are specializing functionality as dedicated roles, for example, 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.

5.2. 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[21] has deployed FDB for more than 0.5M disk years without a single data corruption event.

It is hard to measure 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.

5.3. Fast recovery

Fast recovery is not only useful for improving availability but also greatly simplifies software upgrades and configuration changes and makes them faster. The 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.[16] 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.

5.4. 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, for example, 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.

6. CONCLUSION

FoundationDB is 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, thus, allowing unusually quick recovery time and improving availability. Finally, deterministic and randomized simulation has ensured the correctness of the database implementation.

Acknowledgments

FoundationDB is still an ongoing effort. We thank Zhe Wu, He Liu, Dan Lambright, Hao Fu, Neethu H. Bingi, Renxuan Wang, Sreenath Bodagala, Yao Xiao, Aaron Molitor, our SRE team, and the Snowflake team for their continuous improvement to the project. C

References

1. CouchDB. https://couchdb.apache.org/.
2. Flow. https://github.com/apple/foundationdb/tree/master/flow.
3. FoundationDB. https://github.com/apple/foundationdb.
4. FoundationDB Joshua. https://github.com/FoundationDB/fdb-joshua.
5. Foundationdb storage adapter for janusgraph. https://github.com/JanusGraph/janusgraph-foundationdb.
6. Janusgraph. https://janusgraph.org/.
7. Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press Cambridge, MA, USA, 1986.
8. Bernstein, P.A., Hadzilacos, V., Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Boston, MA, USA, 1987.
9. Chockler, G., Malkhi, D. Active disk paxos with infinitely many processes. In *ACM PODC* (2002).
10. Chrysafis, C., Collins, B., Dugas, S., Dunkelberger, J., Ehsan, M., Gray, S., et al. FoundationDB record layer: A multi-tenant structured datastore. In *ACM SIGMOD* (2019).
11. Cidon, A., Rumble, S., Stutsman, R., Katti, S., Ousterhout, J., Rosenblum, M. Copysets: Reducing the frequency of data loss in cloud storage. In *USENIX Annual Technical Conference* (2013).
12. Dean, J., Barroso, L.A. The tail at scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80.
13. Facebook. Rocksdb. https://rocksdb.org.
14. Groce, A., Zhang, C., Eide, E., Chen, Y., Regehr, J. Swarm testing. In *ACM ISSTA* (2012).
15. Hipp, R.D. SQLite. 2020. https://www.sqlite.org/index.html.
16. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.D. Software rejuvenation: Analysis, module and applications. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing* (1995), 381–390.
17. Kung, H.T., Robinson, J.T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 6*, 2 (1981), 213–226.
18. Leesatapornwongsa, T., Hao, M., Joshi, P., Lukman, J.F., Gunawi, H.S. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *USENIX OSDI* (2014).
19. Lomet, D., Fekete, A., Weikum, G., Zwilling, M.J. Unbundling transaction services in the cloud. In *CIDR* (2009).
20. Mohan, C., Haderle, D., Lindsay, B.G., Pirahesh, H., Schwarz, P.M. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17*, 1 (1992), 94–162.
21. Shraer, A., Aybes, A., Davis, B., Chrysafis, C., Browning, D., Krugler, E., et al. Cloudkit: Structured storage for mobile applications. *Proc. VLDB Endow. 11*, 5 (Jan. 2018) 540–552.
22. Yabandeh, M., Gómez Ferro, D. A Critique of snapshot isolation. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys'12*, Bern, Switzerland 2012, 155–168.

Jingyu Zhou, Meng Xu, Bala Namasivayam, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Alec Grieser, Yang Liu, Alvin Moore, Xiaoge Su, and Vishesh Yadav ([jingyu_zhou, meng_xu, bnamasivayam, russell_sears, john_leach, daver, xind, agrieser, yliu68, alvinm, xiaoge_su, vishesh_yadav]@apple.com), Apple Inc., Cupertino, CA, USA.

Alexander Shraer (alexander@cockroachlabs.com), Cockroach Labs, New York, NY, USA.

Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, and Bhaskar Muppana ([alex.r.miller, evan.tschannen, steve.atherton, aj.beamon, g.muppana]@snowflake.com), Snowflake Inc., San Mateo, CA, USA.

Will Wilson, Ben Collins, and David Scherer ([will.wilson, ben.collins, david.scherer]@antithesis.com), Vienna, VA, USA.