Decoupled Transactions: Low Tail Latency Online Transactions Atop Jittery Servers

Pat Helland
phelland@salesforce.com
Salesforce
San Francisco, CA, USA

ABSTRACT

Modern cloud data centers are busy places that share lots of resources. It is common for services to fluctuate in their responsiveness, sometimes becoming slow or very slow. Many distributed systems experience cascading slowness as one or a few slow servers (or their network) bring the entire system to its knees.

Non-transactional work copes by using idempotent retries bypassing the laggards. For transactional databases, it's not so simple. This paper sketches a design for a distributed database providing responsive snapshot isolation transactions even when some of its servers and connections stop or, more perniciously, just slow down.

We present a thought experiment for a **decoupled transactions** database system that avoids cascading slowdown when a subset of its servers are sick but not necessarily dead. The goal is to provide low tail latency online transactions atop servers and networks that may sometimes go slow. Assume at most F recalcitrant servers in the database. Can we design a robust system that makes predictable progress while not waiting for F slow servers? Can we use these ideas for practical deployments in modern data centers with availability zones and today's expected operational challenges?

This hypothetical design explores techniques to dampen application visible jitter in a database system running in a cloud datacenter when *most of the servers* are responsive. This inevitably causes us to examine the nature of a database's knowledge of correctness and how that can exist without a centralized authority.

ACM Reference Format:
Pat Helland. 2022. Decoupled Transactions: Low Tail Latency Online Transactions Atop Jittery Servers. In *Proceedings of 12th Annual Conf on Innovative Data Systems Research (CIDR '22) (CIDR'22).* ACM, New York, NY, USA, 30 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Typically, online transactional databases are deployed with expensive dedicated hardware[1] in data centers owned by their enterprise.

It is desirable to run these solutions in cloud data center environments to avail ourselves of their tremendous advantages in flexible deployments and elasticity of resources. However, there are new challenges as these shared environments do not offer predictable response time to requests flowing across servers. In the cloud, responses exhibit probabilistic latency. Worse, the expected response time varies as the environment experiences pressure. Some servers will take noticeably longer while others provide their normal expected response time distributions.

Human facing *non-transactional* work provides a vibrant and responsive experience using *retries of idempotent operations*[2].

Can we provide high-availability low-latency responses from a high-throughput externally consistent[3] distributed database that tolerates jittery or sick servers? Can we know what happened in the past quickly with low tail-latency so we can briskly make changes in the future based on what happened? Can we rapidly protect against conflicting concurrent updates as we commit transactions? How the heck can we build a system without a centralized authority to remember what's happened and decide what should happen next?

It is not our goal to define a super-scalable SQL system.

We hope to scale to tens of servers with *predictable response time* while running in a *largely unpredictable environment.*

This thought exercise aspires to pry apart the cross-server dependencies in our imaginary system and push our minds to understand the nature of how state is represented in a system and how that state may be decentralized. Can we bound the internal dependencies within the implementation of a database so transactions can commit rapidly even when some components aren't responsive? It is hoped that this can empower future systems to blithely ignore performance irregularities in large data centers and just give prompt answers when *enough servers* participate.

1.1 Inspiration for This Work

In the cloud, we frequently measure the latency between a request and its response as one service invokes another. These response latencies are probabilistic and can be expressed as a CDF (Cumulative Distribution Function) showing the expected likelihood a response will be received by a specified time[4].

For example, an SLO (Service Level Objective) for a service may specify a 99.9% probability of responding within 2 milliseconds.

A system will typically have multiple dimensions to its SLO such as windows of time in which the target response latency is actually met. For example, 99.9% of the responses will be received within 2 milliseconds *at least 98% of the time.*

When a larger system is built using smaller services, the SLO of the smaller services can have significant impact on the resulting

[1] High quality servers, data storage in SANs (Storage Area Networks) [10, 43] and bespoke networks are common deployments for mission-critical databases.

CIDR'22, January 10-13, 2022, Chaminade, CA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

[2] See *The Tail at Scale* [20] for an excellent discussion about bounding tail-latency probabilities by retrying to an alternate server.

[3] I.e. snapshot isolation [13, 41] as seen external to a distributed database.

[4] See §12 (Appendix A: Building on a Jittery Foundation) for an in-depth discussion of the challenges we face within modern data centers.

system. When all goes well, the system itself offers its expected latency as each of its components meets their SLO for prompt service. When services within the system are less punctual, we can see impact on the larger system's responsiveness[5].

Retrying slow requests to an alternate server is a common technique to mitigate this [20]. Proactively sending two or more requests to different servers and accepting the first response dramatically lowers the expected latency for the first response.

In the DB world, we see latency bounding for log subsystems. One example is *Apache Bookkeeper* [7]. Log entries are appended by writing to N log-storage-replicas (called *Bookies*). When Q of N log-storage-replicas have acknowledged the receipt of the log entry, it is durable. Since N - Q log-storage-replicas may be slow, the expected and observed SLO for writing to the log is dramatically reduced. The database system is predictably faster with less variability.

AWS Aurora[46, 48] carries this further with replicated *AWS-storage-servers* for log replay and block creation. These AWS-storage-servers are effectively the lower half of the database. They are placed with 2 servers in each of 3 AZs (Availability Zones). When 4 of these servers have acknowledged the log entry, Aurora considers the log entry to be durable. Transactions having commit records in the entry may be confirmed to the waiting human. Even when AZ+1[6] failures have happened, the commit record can be read later. *Logging is fast even if 2 of the 6 AWS-storage-servers are slow.*

Still, major portions of the database run in a single server. If that server is jittery, humans will experience unpredictable delays.

Can ALL of the database be responsive atop unpredictable servers?
Can every aspect of a database be *decentralized* and *responsive* even when some of its pieces are not?

1.2 Applicability to a Broad Range of Systems.

This paper is about jitter-avoidance through the use of *quorum*. It investigates techniques to manage complex systems based on an alternate form of order (called *seniority*) that is largely decoupled from classic *happened before* partial-order messaging guarantees[35].

By sketching a design for a transactional database, we demonstrates the ***broad applicability of these principles*** *to many complex distributed systems.*
Paxos[36] provides *linear order* but it jitters We provide *partial order* and avoid jitter

1.3 Availability in a Complex System

The phone system over land lines offers amazing availability. Occasionally, a call is dropped but you can redial and connect.

Availability is defined as the opportunity to dial another call.

In transactional database systems, transactions may fail. Lock conflicts or other problems can cause work to be discarded. You don't want this to be common but once in a while is OK. When a transaction aborts, the application can restart the work and hopefully succeed. In a distributed database, restarted transactions may route to a different database server. If a single transaction gets stuck, the application can abandon it, retry, and probably succeed.

Our *decoupled transactions database* may occasionally timeout while doing a transaction. Even so, it remains open for new business.

1.4 Snapshot Isolation Guarantees

The goal for this design is to imagine a database that *runs many existing applications* with excellent *availability* and *responsiveness*.

Special attention is required to when, how, and with what guarantees the application sees and changes its data. *Snapshot Isolation* [13, 41] defines the database behavior that an application sees when running with concurrent work. It is arguably the most common isolation guarantee provided by modern commercial databases. In its basic form, snapshot isolation provides guarantees when reading and committing updates in a transaction:

- **Snapshot reads:** Each transaction gets a *snapshot time* as it begins and, as it reads data, sees updates from transactions committed before the snapshot time and its own updates.
- **Conflict detection prior to commit:** As a transaction is about to commit, the database will ensure that none of the updated records has been changed by other transactions since this transaction's snapshot time.

Snapshot isolation is extremely popular. Many existing applications depend on these guarantees for their successful execution.

1.5 What's Jitter?

Jitter mean behavior different than the expected norm. In particular, for response time varying from what's expected. In electronics, it means deviation from the circuit's expected clock timing.

In networking, *jitter* describes larger than expected variance in the delay moving through the network [17]. Servers may jitter for many reasons from *gray failure* to Operating Systems to Java VMs. See §12: (Appendix A: Building on a Jittery Foundation).

Within a distributed system, jitter comprises both *variability from the network* and *variability from the server processing a request*. From the standpoint of the client issuing a request, it can't tell the cause of the delay, just that the response is not as zippy as hoped.

If part of the cluster is not responding, those servers may or may not be doing work. For servers spread widely within or across data centers, an observer may see some servers responding quickly but not all of them. *The ones WE see responding quickly may not be providing prompt responses to other servers in the datacenter.*

What can we assume so we can make progress?

1.6 Quorum: Avoiding Snapshot Isolation Jitter

Snapshot isolation transactions can do independent changes to the database *as long as there are no conflicting updates and each transaction sees only snapshot reads*. What if these two things can be accomplished without stalling behind jittery servers?

We can avoid stalling[7] when we do work with quorums. The idea is to ask a collection of servers to do each operation and wait for only some of them to answer. Suppose we have a quorum of N servers and we only need answers from Q of them (where Q > N/2). When we get Q answers, we know that at least one server of the Q has already seen any single previous operation. By combining all Q answers, we can know about previous work. F is our maximum

[5] See the paper *Thinking about Availability in Large Service Infrastructures* [39] for great discussions of SLAs (Service Level Agreements), SLOs (Service Level Objectives), and SLIs (Service Level Indicators). The discussion also includes multi-dimensional SLOs.

[6] Tolerating AZ+1 failures means an entire AZ can be lost *at the same time* as one more server is down for other reasons [46].

[7] The probability of stalling drops *dramatically* when we don't wait for *every server*.

number of jittery servers[8]. If F < N-Q, we can learn what we need to learn while not getting stuck behind jittery servers.

We will propose that both *checking for conflicting updates* and *snapshot reads* can be accomplished with quorum operations.

Decoupled transactions: *Can enough servers do enough work?*
Healthy servers finish the job and ignore slow or dead ones.
Decoupled transactions: *Tolerate up to F jittery servers.*

1.7 Framing the Proposed Design

To avoid jitters, we minimize coordination and do it with special care. We only coordinate to *verify no transactional conflicts*, *keep track of very recent updates*, and *remember coarse grained information about persistent shared data*.

The big idea behind this design is to *maximize the read-only data* and ensure we can read it *while sidestepping jittery servers*. If all but the recent data is kept in shared storage in the cloud *and we can read it without jittering*, that solves a big part of the problem, the older portion of the *snapshot reads* aspect of *snapshot isolation*.

To avoid jitters while reading from shared storage, we must allow independent read-only access (once its no longer *recently changed*). Separately, we must find recent changes while bypassing jitter.

Finally, we allow multiple independent workers to make changes. Independent changes to the database can be correct if we focus carefully on the two requirements for *snapshot isolation*:

- **Snapshot reads** of committed data as-of the snapshot.
- **Prevent update conflicts** since the transaction's snapshot.

Decoupled transactions database uses a few basic concepts:

- **Record-versions:** Change to a SQL database can be represented by generating new *record-versions*.
- **Worker servers:** Worker servers can perform transactions independently by generating new record-versions. They have their own transaction logs.
- **Shared storage of older record-versions:** Older record-versions are organized in a shared storage for easy access by key within an LSM: Log Structured Merge [37, 40] system.
- **Newer record-versions are found in other workers:** After a worker commits a transaction creating new record-versions, the worker allows them to be read directly[9].
- **Snapshot reads** combine older record-versions with newer record-versions as needed.
- **Optimistic transaction updates:** Workers perform transactions optimistically, committing if there are no conflicts.
- **Conflict detection before commit:** Conflict detection of updates happens at centralized coordinator(s).

External application servers create database connections to one of a pool of worker servers. If an individual worker server becomes unresponsive, the app can time-out and try another worker server.

Snapshot reads and conflict detection use quorums.
They don't stall even when up to F nodes haven't responded.

[8]See §7.2 for a discussion of how the notion of F jittery servers can be practically applied to 3AZ data centers for various use cases within the database. Each of these use cases has its own definition of "F" derived from its particular requirements.
[9]Of course, the worker making recent changes *can get jittery*. For those rare cases, we can *rapidly* kill the worker and read its log. This is discussed below in §14.
In depth discussions of this are found in §15.3, §15.4, §15.7, §15.9 and §15.12.

1.8 Building a DB without a Central Authority

Databases thrive on ordered transactional updates seen by application developers. **Serializability** [14] and related forms of weaker isolation [13] provide applications a façade of loneliness. **Linearizability** [33] is a property of a single centralized object and its relationship to clients as they perform operations. What happens when that centralized object goes slow? Can we avoid using a central server to build our database?

If we truly believe that up to *F servers* can be abstaining from work at any time, any centralized adjudicator of truth may be in suspended animation. *Anything from any single server may be frozen and not able to make progress.* This leads to an existential quandary: *How did this database start and what is its source of truth?.*

What is the *Uncaused Cause*[10] that launches our database?

Our *decoupled transactions database* starts with a quorum of *catalog servers* describing the contents of the database, logs and their state of repair, as well as some stale notion of the ongoing state of servers comprising the database. *We assume their existence.*

When a sufficiently large subset of our catalog servers comes to life with their own partial knowledge of the system, we can reason about the database, its contents, and allowable changes to its future. No single centralized server has the truth or controls the future.

Decoupling transactions is not enough in a jittery world
We must also decouple the way we control the system.

1.9 A Sketch of What's Coming

The main part of the paper has 17 pages in these sections:

- **Decoupled Transactions Databases –** Sketches the pieces of our imaginary system and points out possible jitter-risk.
- **Seniority: A Disorderly Order –** Provides a framework to manage order and yet remain independent of jitter.
- **Logical Time & Transaction Commit –** Describes how transactions get *partial order* while avoiding jitter.
- **Quorum: Jitter-Free Fuzzy Visibility –** Looks at the essence of quorum: jitter avoidance with quirky behavior.
- **Confluence: Adding Clarity to Fuzzy Quorum –** We explain *confluence* and its use with *quorum* to control the state of a complex system.
- **Jitter-Free Snapshot Isolation & 3AZ –** How can our database avoid jitter when running over 3AZs? We review the major aspects of the database and how they avoid jitter.
- **Discussion –** We consider the impact of using quorum, confluence, and seniority on complex systems.
- **Related Work –** Some readers may want to start here to compare and contrast with earlier published work.

Finally, we have 12 pages of appendices covering:

- **Appendix A: Building on a Jittery Foundation –** Reviews assumptions about modern data centers.
- **Appendix B: Quorum's Subtle Challenges –** Surprises!
- **Appendix C: Jitter-Free Database –** How it works!
- **Appendix D: Jitter-Free Log-Repair –** Log repair details.

[10]Aristotle proposed that if everything has a cause, back in time we must eventually arrive at a *First Cause* [42]. Saint Thomas Aquinas introduced this into Medieval Christian theology in the 13th century as a proof of the existence of God. We introduce this notion to reason about the foundation and correctness of our system.

2 DECOUPLED TRANSACTIONS DATABASES

Decoupled transactions databases store data in shared storage. Five different types of servers do different jobs. Work happens by generating new record-versions and reading the correct record-versions for snapshot reads.

As data ages, it migrates to a key-value store in shared storage implemented as an LSM (Log Structured Merge System) [37, 40]. Our catalog maps key-ranges to stored data and its location in shared-storage. We check for conflicts before transactions commit.

Let's first look at this design without considering jitter. Then, we'll examine the risks for jitter in this design and their mitigations. We discuss the problems with *sharing status without a centralized authority*. Digging deeper into our *jitter risks*, we *taxonomize the kinds of status data we need to share.*

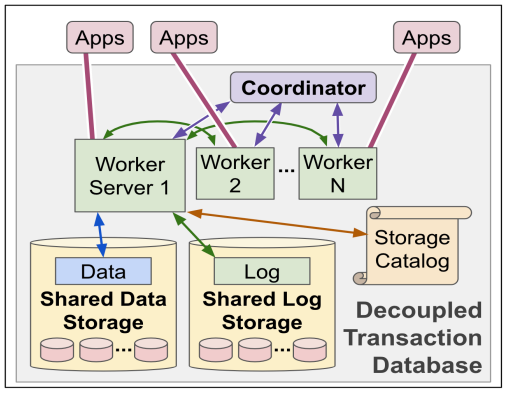

Figure 1: A *Decoupled Transactions Database* uses five types of servers to provide snapshot isolation transactions. Apps use DB connections to 1 or more worker servers.

2.1 Server Roles in Decoupled Transactions DBs

A **decoupled transactions database** has 5 server roles: *worker*, *coordinator*, *data-storage*, *log-storage*, and *catalog*. See Figure 1.

Worker servers are database servers. Incoming DB-connections feed in SQL requests. Execution, reads, and updates happen in worker servers. Workers *log to their own log* in shared storage.

Coordinator servers help avoid conflicting updates as transactions commit and help locate recent changes for snapshot reads.

Data storage servers hold replicas of data files with DB data.

Log storage servers hold replicas of the log extents used for logging. These include special mechanisms to optimize appending to the log and to accurately recover the end of the log after a failure.

Catalog servers track metadata for workers, log-storage and data-storage. They manage worker's log extents and replicas. They organize the data files used by the key-value store, describing the location of data extent replicas in shared storage.

2.2 Doing Work in a Decoupled Transactions DB

Each incoming transactions arrives to a worker server over a database connection. Transactions begin as-of a snapshot time. The worker reads **record-versions** (i.e., database records committed as-of the snapshot). It may also update records and commit its transaction. Record-versions created by transaction T_1 include their commit time. T_1's record-versions are visible to all subsequent transactions with a snapshot time greater than T_1.

In a decoupled transactions database, each record update creates a new record-version that layers atop earlier record-versions and is visible to transactions with later snapshot times. Record-versions older than a few minutes or so are in shared storage and are visible to all database servers in the database. Recent record-versions are found within the worker performing the transaction that created them. *Snapshot read semantics* are provided by reading the latest record-version that committed prior to the reader's snapshot time. Snapshot isolation depends on *conflict detection* and *snapshot reads*:

Conflict detection is provided by the coordinator. Before a transaction commits, a *permission-to-commit* request is sent to the coordinator to verifies that none of the proposed changes conflict with any other committed (or committing) transactions. If no conflicts are found, permission is granted.

Whereabouts of recent updates are obtained from coordinators at the beginning of each transaction. Each *whereabouts entry* describes a *possible recent update* made by a worker-server. These are guaranteed to cover all updates that commit[11] going back at least to the *snapshot time* for the transaction. Whereabouts cover both individual records as well as key-ranges[12].

Snapshot reads must return the most recent record-version committed before the snapshot time. *Older record-versions* can be found in shared storage with an LSM-based key-value store (using both the catalog servers and the data storage servers). *Recent record-versions* can be found in the worker server(s) that updated the desired records. Coordinators provide guidance to locate recent record-versions within worker server(s).

2.3 Record-Versions and Their Unique Identity

SQL semantics need both *tables* and *indexes on these tables.*

Tables are implemented by creating record-versions with unique *primary keys* for each row by concatenating:

- *Table-ID* for the table, and
- *The SQL defined unique primary-key* comprising an ordered set of the table's columns.

The record-version for each row contains the contents of that row being stored in the table.

Indexes (also called *secondary keys*) are a set of record-versions whose keys concatenate:

- *Table-ID* for the table,
- *The Index-ID* within the table, and
- *The SQL defined non-unique secondary-key* comprising an ordered set of the table's columns.

The record-version for each unique or non-unique secondary key contains both the secondary key and the unique primary key it references. The combination of the secondary key and its unique primary key is, as a pair, unique.

Tombstone record-versions are identified by either a *primary key* or a *secondary key*. They denoting the deletion of the record as-of the transaction that generated the tombstone.

[11] *Whereabouts* will occasionally capture updates that do not actually commit. This is rare and does not impact correctness.

[12] Key-range whereabouts are beyond the scope of this paper.

Each record has a unique key, primary or secondary.

Primary records contain the contents of a row from a table. Secondary records hold the primary key of the row they index.

Record-version: an image of the new contents of a record, perhaps a *tombstone*. Each record-version is identified by both by *its unique record-key* and the *timestamp* of the transaction creating the new value. It may be a primary record-version or a secondary record-version.

2.4 Finding Record-Versions for a Snapshot

Implementing *SQL queries* uses both *exact-key probes* as well as *key-range scans*. This is true for both *indexes* using secondary-keys and *tables* using primary keys. To generate the correct value for a *snapshot read*, we need to combine older results from the LSM persistent storage with recently committed changes from the workers committing those changes.

Exact-key probes by workers: Worker-servers locate a record using its exact-key along with a *snapshot time*. It locates the latest record-version (if any) present within its *recent changes*.

Exact-key probes into the shared LSM: A search of the LSM locates a record using its exact-key as-of a *snapshot time*.

Key-range scans by worker servers: Workers can also support *key-range scans* over a key-range to locate the latest record-version (per key) within the key-range being scanned. Only records older than the snapshot time are returned. For each unique record-key, only the latest record-version before the snapshot time is returned.

Key-range scans of the shared LSM: These scan all files in all levels in the shared LSM that *might have a record-version within the key-range*. For each level, a *scanlet* produces a *per-level key-range result*. These *scanlets* are merged to produce an ordered set of record-versions within the key-range. Only records older than the snapshot time are returned. For each unique record-key, only the latest record-version before the snapshot time is returned.

2.5 Storage of Log and Data

Shared storage is accessible to all database servers. Different servers provide the needs for *data storage for the LSM* and *log storage for the workers' transaction logs*.

Data-files are written to *data storage servers* by workers. Data files are uniquely identified by a *data-file-id*, a 128-bit UUID. They are immutable and, once written may never be changed. Any worker server in the database may read a stored data file.

Workers write data files to *flush recently committed record-versions* to the LSM and *merge older parts of the LSM*[13]. The *storage catalog* manages metadata for the LSM to track these *data-file-ids* providing efficient ways to read a record-version by key.

Low-latency jitter-free reads of data files are possible by replicating the data files over a set of data-storage-servers using consistent hashing[51]. Other techniques exist, too.

Log-extents are written by appending *fragments* to *an extent* replicated over a set of *log storage servers*. Worker servers append transaction log records, contained in fragments, as they process transactions. Each worker has a private *log-window* comprising an ordered sequence of *log-extents*, each of which is replicated over a set of log-storage-servers.

Log storage servers, log-windows, and log-extents are specially designed with two goals:

- **Efficient appending of fragments** containing log records by a single worker server to its private log window.
- **Repair of a log-window after a failure.** This guarantees the durability of committed transactions and ensures the repaired log-window contents never change. See §15.9

2.6 Caring for Data in Shared Storage

Worker servers log updates (i.e., record-versions) and transaction commits to their per-worker log in shared storage. Recently committed record-versions are kept in the worker's memory and accessed as needed. Let's see how these older record-versions are managed.

Adding New Record Versions atop Shared Storage. Shared data storage receives new data files when new record-versions are *flushed* to the LSM. $\text{Time}_{\text{Flush}}$ delineates record-versions *old enough to be found in shared storage*. Record-versions newer than $\text{Time}_{\text{Flush}}$ remain in the worker server that created them[14].

Reorganizing Record-Versions in Shared Storage. Key-value stores may reorganize their data for easier access by key, incrementally rewriting key-ranges into new data extents to make reading by key easier (e.g., LSM merges). While the newer organization is better for reading records, the older one works fine, too.

Cataloging changes to shared storage. Changes to key ranges from *flush* or *reorganization* (e.g., merge) are tracked by the storage catalog along with the location of the data extents in shared storage.

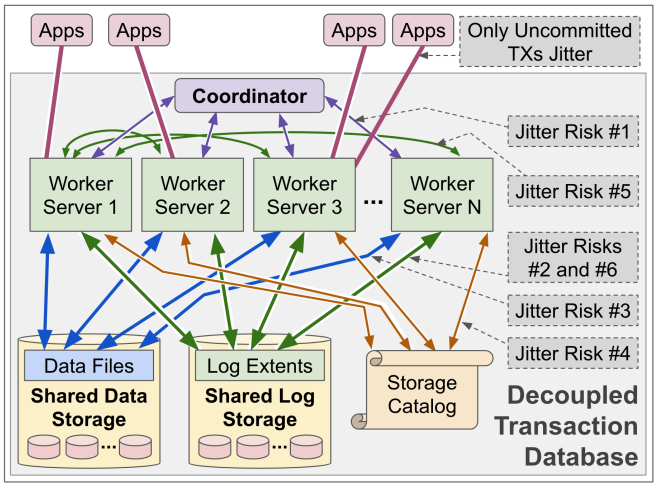

Figure 2: There are six major risks of jitter to address.

2.7 Where Are the Risks from Jittery Servers?

Transactional work comprises two things: *snapshot reads of existing record-versions* and *non-conflicting updates to records* (i.e., creating new record-versions). Without care, we risk that jittery servers can stall the database. See Figure 2.

[13] Merging of LSM data files reorganizes them for easy key-based record-version reads. This is sometimes called *compaction*.

[14] Flushing records from workers to shared storage is allowed to take a while. When complete, we can reclaim worker memory. Workers retain their record-versions until *after they are known to be in shared storage* and $\text{Time}_{\text{Flush}}$ is advanced. Because workers flush independently, at first they will have different $\text{Time}_{\text{Flush}}$ values.

Our definition of availability allows transaction abort. Jittery workers may block ongoing *uncommitted* transactions causing timeout, abort, and hopefully application retries.

The database must promptly accept new work including snapshot reads of committed record updates. *Stalled workers may not block access to committed updates and their record-versions.*

Uncommitted work may stall.
→ *Reading committed work must not stall.*

Consider six interactions in the database with risk for jitter:

- **Risk #1: Worker to coordinator:** Coordinators help with:
  - *Partial order of transactions*
  - *Avoiding conflicting updates*
  - *Learning about recent record-versions*

  They must not stall for jittery servers.
- **Risk #2: Workers appending to their log:** Must not be slow when writing log records or updating the catalog.
- **Risk #3: Workers reading data-files:** We must rapidly read data files even when storage servers go slow.
- **Risk #4: Worker to catalog:** Reading from catalog must not wait for the jittery servers.
- **Risk #5: Asking other workers for recent record-versions:** The target worker (or network) may be slow.
- **Risk #6: Reading a slow worker's log to side-step it.** If a worker can't respond about its recent updates, we look in its log. This, too, has risks of jitter we must avoid.

The crux of our jitter-risks involve *ordering work across services.*

2.8 Order without Dependency

Much of the work we do involves some form of order. A piece of work may interact with another and we must control that. We need order across these pieces of work and their interactions with other ongoing work in the system.

Let's decouple ***partial order of transactions*** from ***messages across servers***

Seniority: a form of *partial order* applied to transactions

Two types of order:

- **Happened before:** Messaging in a distributed system
- **Seniority:** Another type of *partial order*

Happened before is always jittery!
Messages between servers flow through single servers
Each server may possibly jitter...

Seniority of transactions can be jitter-free!

We can also use seniority for other things in the database.

Let's first consider jitter-risk #1: (worker to coordinator).

Coordinators support *snapshot isolation transactions* by:

- **Avoiding conflicting updates:** A transaction T_1 may not commit if it updates records changed by another transaction T_2 since T_1's *snapshot time.*
- **Locating recent updates:** Any subsequent transactions after T_1 commits with an earlier *snapshot time* must see all updates made by T_1.

Partial order of transactions is needed for *snapshot isolation.* For any two transactions T_1 and T_2, we must see one of:

- **T_1 is before T_2**
- **T_1 is concurrent with T_2**
- **T_2 is before T_1**

Can we provide *partial order of transactions* without risking jitter?

3 SENIORITY: A DISORDERLY ORDER

Our decoupled transactions database can be decoupled in execution because it uses *seniority* as a form of order. Seniority is largely decoupled from the messages that flow across servers.

Seniority is an attribute of *items.* The following *items* have seniority assigned to them:

- **Transactions:** Each is a separate item with its own seniority
- **Transaction logs**, extents, & fragments have seniority ranges
- **Flush's data files** have a range of seniority
- **Merged data files** have a range of seniority

Seniority uses quorum to know a ***fuzzy beginning*** and a ***fuzzy retirement*** of items in the system.

Since we have a decentralized system,
we cannot crisply know when things begin or end.

This section covers:

- **§3.1 (Seniority of Transactions):** This introduces *seniority of transactions*, briefly explaining their provenance[15].
- **§3.2 (Transactions: Items with an Exact Seniority):** Transaction have an *exact seniority* at a single *logical time.*
- **§3.3 (Seniority Ranges of Other Items):** Other items with seniority (e.g., log stuff and data files) use a range.
- **§3.4 (Seniority and the Lifecycle of Items):** Items have a lifecycle of birth, mid-life, retirement, and removal. Transitions in this lifecycle are not atomic but have fuzzy visibility (due to quorum) when changes happen. Seniority allows us to reason about the fuzziness and bound its duration.
- **§3.5 (Retirement of Items):** Here, we briefly summarize when various types of items retire from the system.

3.1 Seniority of Transactions

Transactions are assigned their *seniority* as they attempt to commit. Before committing its transaction, a worker must get *permission-to-commit* from a quorum of coordinator servers. See §4.

To commit: ***workers guess a transaction's seniority*** and ***coordinators confirm that seniority***

Seniority is a logical time within the database

Seniority of transactions is *partially ordered* in the database:

- **Seniority is assigned per worker:**
  - *Committing workers guess a seniority*
  - *Seniority is requested in permission-to-commit*
- **Seniority is partially ordered per coordinator server:**
  - *Each server locally processes permission-to-commit*
  - *Local processing by seniority order as guessed by workers*
- **Seniority is partially ordered across coordinator servers:**
  - *Quorum across coordinator servers*
  - *Partial order of transactions by seniority*

[15] A more in-depth explanation follows in §4:(Logical Time & Transaction Commit).

Transactions are partially ordered by seniority
A partial order of all transactions in the database

Assigning transaction seniority does not jitter

3.2 Transactions: Items with an Exact Seniority

A single seniority for is assigned to each transaction. The transaction commits at a *single logical time*. Other transactions have their own seniority at their own single logical time.

When two transactions have different seniorities, one is before the other. Occasionally, two transactions have the same seniority and *they are concurrent*. Transactions have a *partial seniority order*.

3.3 Seniority Ranges of Other Items

In addition to transactions, there are other items with seniority assigned as they are created. The seniority of these other items is derived from the transactions included within them. They are defined by the range's lower-bound and upper-bound of seniorities of the transactions captured within these items:

- **Log (fragments, extents, and log-windows):** Each has a range for its seniority. See §15.1.
- **Flush:** Each data files has a range for its seniority. See §14.2
- **Merge:** Each data file has a range for its seniority. See §14.3

Catalog manages log, flush, & merge item seniority
The catalog retires these items when no longer needed

Each of these stored items is *assigned its seniority as it is created.*

Extents and log-windows may not yet have a crisp upper bound for their seniority while receiving fragments for new transactions.

3.4 Seniority and the Lifecycle of Items

The lifecyle of items (including transactions) goes through stages. They are created, live within the system, retire, and have their resources reclaimed.

Items many be used across many parts of the database. Log-storage servers, data-storage servers, catalog servers, and coordinators all interact with items. Items are used across these boundaries of internal implementation. Their lifecycle needs to be coordinated.

Fuzzy visibility of changes. As with all changes within a distributed system, transitions are not atomic. For each of these changes, we see a *fuzzy window of visibility*:

- **Before the change begins:** No server sees the change.
- **During the change:** Some servers see the change and others do not. Some may see it appear, disappear, and reappear.
- **After the change:** All servers see the change.

Seniority controls the visibility of items' lifecycles

Quorum is used to share changes in an item's lifecycle

Controlling changes with quorum is both good and bad:

- **Good:** Quorum can be used to tolerate jitter
- **Bad:** Quorum shows *fuzzy visibility of changes*

Seniority is used to bound the fuzziness of visibility

3.5 Retirement of Items

Each type of item has different rules for its retirement:

- **Transaction T_1 retires when the latest of both:**
  - *Oldest system snapshot time advances* past T_1
  - *Flush time for T_1's worker advances* and T_1's in the LSM
- **Log related items retire when:** They aren't needed for *system restart* or *archive*.
- **Flush data file items retire when:** The changes in the flush are visible in the LSM.
- **Merge data file items retire when:** Newer merged *LSM data files* cover all its key-ranges.

4 LOGICAL TIME & TRANSACTION COMMIT

Seniority in our system comes from transactions and their commit time. Commit time is a *logical time* that advances without jitter and without any special hardware in servers or the network.

The **coordinator subsystem** comprises N_{Coord} *coordinator-servers* working in conjunction with *coordinator-clients within each worker-server*. See §2.1 and §5. Worker servers wait for Q_{Coord} servers to complete and advance work through the database. See §14.1.

Transaction commit happens with the help of a quorum of the coordinator servers checking for conflicting updates

Worker servers and *coordinator servers* have their own clocks that may be out of sync. **Logical time** is calculated at each server based on the time from their own clocks along with an adjustment to align the server's local clock to the system-wide *logical time*.

Each clock may drift independently and tight alignment is hard. Misalignment may be due to clock drift or network delays in communicating. We synchronize *logical time* for *transaction seniority*[16]:

- **Seniority of transactions** as they commit, and
- **Seniority of snapshots** and the transactions they see.

Logical time is independent at each server, both workers and coordinators. It never moves backwards but, as we shall see, sometimes its rate of advancement may be adjusted to better align with other logical times in the cluster. Each coordinator tracks its own logical time and independently processes requests at its local logical time.

4.1 Processing Permissions at a Logical Time

Before committing a transaction, the worker-server must ask for *permission-to-commit*. This will check that the pending transaction does not have conflicting updates with earlier transactions.

Each coordinator quorum-server waits to process an incoming operation until its *local logical time* aligns with the operation's requested future time. Each separate server processes operations at the logical time requested. If two requests are for the same logical time, they may be processed in either order. Consider two requests:

- Permission-Request P_j requesting time T_j and
- Permission-Request P_k requesting time T_k.

There are three possible execution orders for these two requests (based on the *local coordinator quorum-server's logical times*):

- $\mathbf{T_j < T_k}$: (P_j *is executed* ***before*** P_k)
- $\mathbf{T_j = T_k}$: (P_j *is executed* ***before or after*** P_k)
- $\mathbf{T_j > T_k}$: (P_j *is executed* ***after*** P_k)

[16] We avoid using the word *order*. Instead, we differentiate *seniority* and *messages between servers* (i.e., a *happened before*).

Requests arriving too late are rejected

Too late as seen by the local coordinator quorum-server

4.2 Workers See a Quorum as-of a Logical Time

Recall that the quorum for coordinators is calculated so that:

$$\mathbf{Q_{Coord} > (N_{Coord} / 2)}$$

Two quorums of size Q_{Coord} must share permission from at least one coordinator quorum-server. Before committing a transaction, workers receive *permission-to-commit* from a quorum of coordinator servers. This includes:

- **Logical time in the future:** A *potential commit time*.
- **List of updated record-version being changed:** Used to check for snapshot update conflicts back to snapshot.
- **Snapshot time:** The logical time for the snapshot check.

When a *permission-to-commit-request* is sent to N_{Coord} coordinator servers, the worker has selected a *logical time* and labeled all the permission requests for the transaction with that time. The transaction commits only when Q_{Coord} of N_{Coord} coordinator servers respond to the request. At that time, the *commit record* may be written to the worker's transaction log.

The *transaction commit time* is the same as the *permission-to-commit logical time.*

It is written in the transaction's commit record.

This becomes the transaction's seniority.

As a *permission-to-commit* happens at a coordinator server quorum, the *logical time of the transaction* lines up with all other transactions in the database. For any two transactions T_1 and T_2, they will share at least one coordinator server and align their logical times.

Seniority: A *partial order of items* in the database

Seniority is not the same as *happened before messages* across servers as defined by Lamport [35]

Seniority aligns with *physical wall-clock time* when a transaction receives permission to commit:

- *Each transaction commits at a single worker*
- *A transaction's worker may jitter in wall-clock time but aligns with other servers' clocks by receiving a quorum*

Partial order of transactions in the database: For any two transactions T_1 and T_2, they are *partially ordered within the entire database.* Exactly one of three possibilities exist:

- T_1 *is before* T_2
- T_1 *and* T_2 *are concurrent*
- T_1 *is after* T_2

4.3 Transactions & Conflict Check

Each *permission-to-commit* request includes both the list of updated records and the snapshot time used by the transaction.

As a new *permission-to-commit* request for transaction T_1 arrives at each coordinator server, it waits until the desired logical time to process it. The server then compares its updates against all previously processed transactions, T_2 where:

- $T_1 = T_2$ or
- $T_2 >$ Snapshot-Time(T_1)

For each record updated by transaction T_1, the coordinator server ensure that record was not previously updated by T_2. If there are conflicting updates, T_1 is *denied permission-to-commit.*

If both T_1 and T_2 attempt to get permission-to-commit with:

- **Same logical time**
  They both select the *same logical time* for permission, and
- **Overlapping records being updated**
  They fiddled with the same records

Then one or both transactions will be aborted[17].

4.4 Tracking Logical Time and Clock Skew

For each incoming *permission-to-commit* request into a coordinator, the coordinator remembers *its logical time when the operation's request was received.* The matching response to the worker includes the logical time (as seen by this coordinator quorum-server) when it received the request. Based on the information in the response, each worker can track the *logical time interval* for a one-way request to each of N coordinators. This interval is the difference between the worker's logical time (when it sent the request) and the coordinator's logical time (when it received the request).

Each interval fluctuates based on network transmission time and the respective logical times of the two servers. It is even possible to have a negative interval if the receiving coordinator's logical time lags behind the sending worker's logical time!

Each worker sees a different *logical time interval* for each coordinator. Workers keep statistics about the expected logical time interval based on recent experience for each coordinator. It doesn't matter if the logical time interval is due to clock skew or network transmission delays. Different coordinators have different delay intervals (as seen by each specific worker).

Using these expected intervals, a worker selects a future time to use for each new request for an operation. This future time is a gamble. It's best to have it arrive at Q or more coordinators *before the desired time* for the request. Unless it's on time at Q or more coordinators, the request must be retried with an even later time. This causes request latency to be increased, possibly slowing work.

4.5 Gradually Aligning Logical Clocks

Consider a system with W workers and N coordinators. Workers accumulate expected *logical time intervals* as they collaborate with N coordinators to perform operations. Each worker tells all N coordinators its opinion of the intervals it's experienced for all N coordinators. Each coordinator sees not only the interval for a single worker-to-coordinator but also how that worker perceives the logical time interval to the other coordinators. Each worker supplies each coordinator with all its observations.

Using this knowledge, a coordinator estimates how far out of alignment its *logical time* is with respect to the other N-1 coordinators. It can gradually increase or decrease the rate of advancement of its logical time to align with the others.

Existing systems such as Huygens [24] can align clocks in a datacenter within a few 10s of nanoseconds, albeit by instrumenting the NIC (Network Interface Card) for precision. Decoupled transactions needs less precision, perhaps 10s to 100s of microseconds.

[17] Each separate coordinator server processes its pending work at logical time in the request. Requests with the same time may be processed in any order. Some coordinator servers might process (T_1 before T_2) while others process (T_2 before T_1). At each server, the later conflicting transaction will not receive permission. Across the Q_{Coord} servers, if any denies permission, the worker client will abort the transaction.

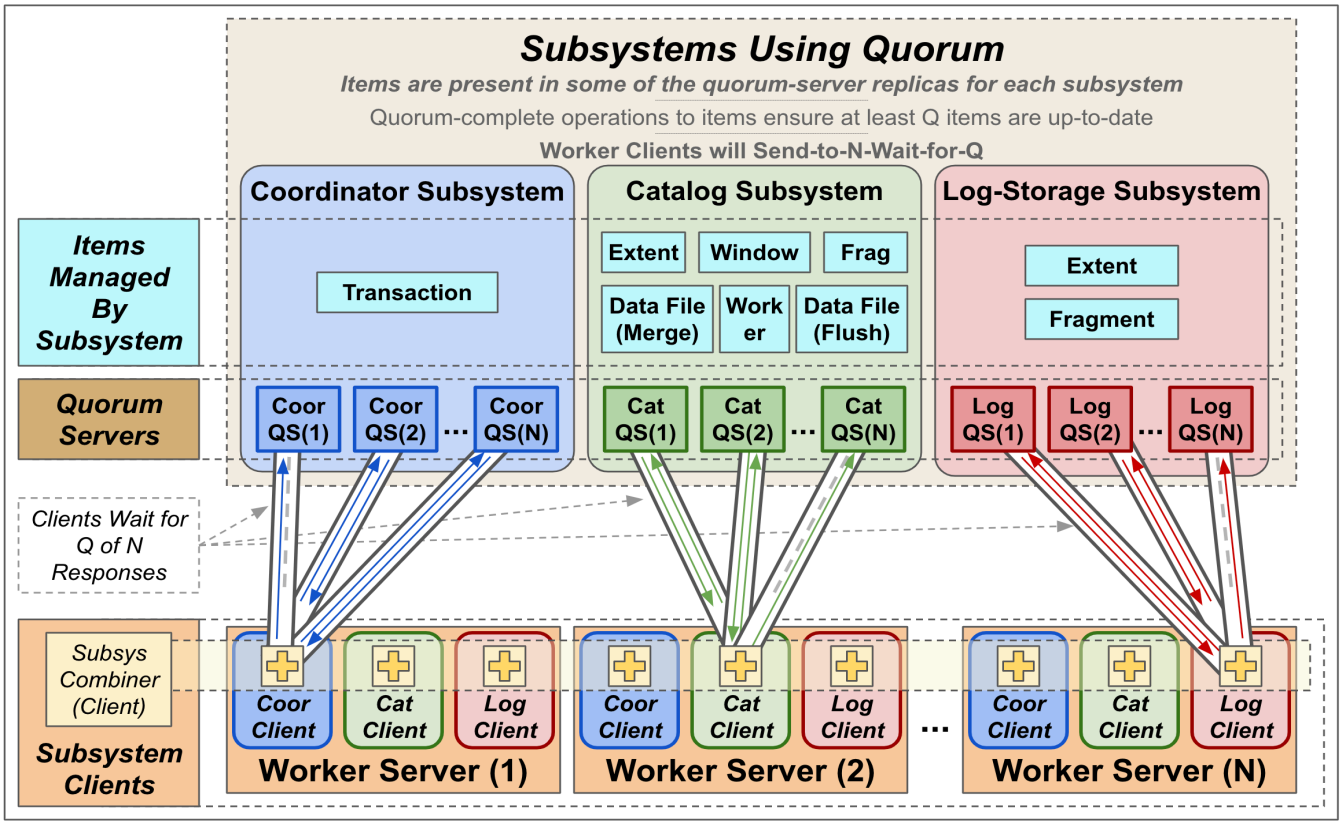

Figure 3: Coordinator, Catalog, and Log Subsystems use quorum to avoid jitter. Clients in workers send operations on different items to a quorum of subsystem services. Subsystem clients (in worker servers) combine quorum results to respond to operations.

Clock skew does not cause incorrect behavior. Partial order is guaranteed by messages that *happened before* [35] each other. When coordinators' logical times have large skew, some coordinators may receive requests for operations too late. The operation still may become complete-quorum if Q of N coordinator process it on time. Either the N coordinators' jitter resistance tolerates this or the worker is forced to select a time farther in the future, impacting the latency of operations. For this reason, our decoupled transactions database behaves best when the coordinators have a small skew between their logical times.

5 QUORUM: JITTER-FREE FUZZY VISIBILITY

Let's finally look deeper at **quorum and the behavior it provides**. Quorum has some subtle and confusing behaviors. We design with it only because it masks the jitter we are working so hard to avoid.

> **See §13: (Appendix B: Quorum's Subtle Challenges) for more details about quorum.**
> *It describes some of the race conditions seen by messages across quorum-servers. These can show some surprising results if they are not considered when designing a system.*

We look first at the subsystems and items we change with quorum operations. The difference between normal messaging and quorum over sets of operations is discussed. Combining quorum results at the client has special semantics we describe as *included operations*. Finally, we summarize the fuzzy visibility guarantees we get when using quorum.

5.1 Quorum: The Lay of the Land

We have three subsystems needing jitter-free support for items. Each item represent ongoing parts of the database and has its own special lifecycle. See Figure 3 and Table 1.

Subsystem	Item Type	Comment
Coordinator		*Transaction Support*
	Transactions	Conflict Check & Whereabouts
Catalog		*Durable Data and Worker State*
	Workers	Status of workers & their flushes
	Flush Files	Per-worker flushed data file
	Merge File	Merged data in LSM.
	Log-window	Per-worker log (many extents)
	Extents	Extent within a log-window
	Fragments	Fragments within Extents *(only used during log-repair)*
Log Storage		*Durable Transaction Logs*
	Extents	Physical extents in log-storage
	Fragments	Fragment replicas in storage

Table 1: Subsystems Using Quorum to Store Items

Worker clients have special code for each subsystem and item type within the subsystem. **Subsystem client combiners** have the semantics to manage quorum by *combining the results* from individual quorum-servers.

When a subsystem-client (within the worker-server) works on an item in a subsystem, it issues N requests to the subsystem and awaits Q responses. These Q responses (from Q quorum-servers) are then *combined* to get the answer needed within the worker-server.

5.2 Operations on Items Using Quorum

Quorums provide challenging behavior to both clients and servers.

Operations are special: *They're not simple read and write operations to data values.* They must cope with weird quorum behavior:

- **Subsytem quorum-servers see weird things:** Quorum-servers each see only *some of the successful operations.*
- **Subsystem clients (inside workers) see weird things:** Servers may give very different answers to an operation (because they've seen different things).

The remaining topics of this section will deal with some of the surprising facts of life when using quorums. Section §6 talks about building clients and servers to cope with this weird behavior.

5.3 Happened Before vs. Quorum Set Included

Partial Order is a *happened before* relationship across operations[18]. Each quorum-server sees requests from workers and issues responses based upon operations that have *happened before* at the individual subsystem's quorum-server. Operations flow across servers inside our distributed database using messaging.

Quorum is used in this paper to describe a set of N *quorum-servers*[19], each of which performs a defined set of operations. A client issues N requests for an operation and awaits responses from Q servers where Q > N/2, tolerating jitter for up to N-Q responses.

Operations across a quorum don't happen atomically. Individual quorum-servers may be very far behind[20]. They become visible to other clients in surprising and perhaps inconsistent ways. See §13. Since quorum visibility is not crisply ordered, we speak of *included operations* for a quorum, not *happened before*. When an operation *happens before* at a server it is *included* in later operations at that server and in quorums derived from those operations.

Quorum operations *include* other operations. We say *include* because two operations are not necessarily ordered when performed at a quorum of servers. Different quorum-servers may execute two operations in different order. The combined quorum result may show they both include each other.

Included operations at a single quorum-server. Each quorum-server processes operations one at a time[21]. This is a single quorum server's local perspective. At a single quorum-server, operations are ordered by *happened before.*

If operation $\mathbf{O_j}$ ***happened before*** $\mathbf{O_k}$ at a quorum-server,
then $\mathbf{O_k}$ ***includes*** $\mathbf{O_j}$
(from that quorum-server's perspective).

[18] See Lamport's *Time, Clocks, and the Ordering of Events in a Distributed System*[35]. The interested reader should examine the paper's *Partial Order* discussion and especially Figure 1. He says *events* where we say *operations.*

[19] Specifically, a set of coordinator-servers, catalog-servers, or log-replica-servers.

[20] Some quorum-servers may be extremely out-of-date, just like Rip Van Winkle. [52].

[21] This is a simplification. Concurrent operations at a single quorum-server may behave as if neither operation precedes the other and neither operation includes the other.

The *Coordinator Subsystem* comprises N_{Coord} *quorum-servers*

Example: *Included operation at one quorum-server $Coord_j$*

Assume the following:

- **$Coord_j$ includes T_1:** Permission-to-Commit operation for T_1 was processed earlier.
- **$Coord_j$ receives operation for T_2:** A *permission-to-commit operation* for transaction T_2 arrives at $Coord_j$.
- **T_2 conflicts with T_1:**
  - *Updating common records*
  - *T_1 is newer than T_2's snapshot*

$Coord_j$ included T_1's permission-to-commit operation
$Coord_j$ rejected T_2's permission-to-commit operation because $Coord_j$ included T_1!

Included operations at a quorum-server impact its processing of later operations

NOTE: A different server $Coord_k$ *might not include* T_1
$Coord_k$ will not detect the conflict between T_1 and T_2
$Coord_k$ may approve T_2's permission-to-commit operation

5.4 Quorum: Combining Operations into Sets

There are a few steps to performing an operation within a quorum:

(1) **Clients send operations** to quorum-servers via messages.
(2) **Quorum-servers**:
   - *Process the operations locally:*
     - Based on local state (i.e., already included operations). May do different things based on local state.
     - Each has seen a subset of the previous operations.
   - *Send back results:*
     - Results provide visibility to earlier arriving operations.
     - Results "include the earlier operations".
(3) **Clients combine the results** from Q of N quorum-servers.
   - *Normally includes all operations from each of them.*
     - Sometimes, results look for unanimous agreement.

Quorum combines sets of operations:
a new operation - is combined with - *a set of earlier operations.*
Earlier operations are *included* in the result.

5.5 Complete-Quorum and Included Operations

Complete-Quorums occur when Q or more servers respond and the client has *combined these responses into a unified result.* These combined results may include the effect of earlier operations sent to the quorum-servers. We say that earlier operations are *included in the complete-quorum.*

Client visible quorum responses and included operations: Suppose a client issues an operation O_k to N quorum-servers. When it receives Q responses to its request, they are *combined* into a single complete-quorum response called CQ_k. If any of the individual quorum-server responses includes operation O_j, then:

Complete-Quorum CQ_k includes Operation O_j.

Incomplete-quorum operations happen when less than Q quorum-servers have processed the operation. If a client hasn't received Q responses, it is not complete (at least yet). The client isn't sure that the operation has been seen by a quorum!

Incomplete-quorum-operations may be *intermittently included* in later operations, sometimes included and sometimes not included[22].

All quorum operations are *temporarily incomplete-quorum operations*. Complete-quorum is not achieved atomically. For a window of time, the operation is not yet complete-quorum.

Permanently incomplete-quorum operations may happen when a client attempts to obtain responses from quorum of the quorum-servers but the client crashes before sending all its requests, some of the servers reject the operation, or something got lost. Since the effect of this operation has not arrived at (or been kept by) at least Q of the quorum, not all subsequent operations will include the incomplete operation in their history.

We must cope with *intermittently included operations.*

5.6 Fuzzy Visibility: Quorum's Guarantees

Quorum provides some limited guarantees about *sets of operations*. It does so without any risk of jitter. What do we know about these guarantees and their limitations? What can we do with them?

Complete-Quorum Sets *include earlier* **Complete-Quorum Sets.**

Let's use: $\mathbf{O_x}$ and $\mathbf{O_y}$ → Operations "x" and "y"
$\mathbf{CQ_x}$ and $\mathbf{CQ_y}$ → Complete Quorums "x" and "y"

What we *do know* is:

IF ($\mathbf{O_y}$ *starts getting quorum* **after** $\mathbf{CQ_x}$ **completes**)
THEN $\mathbf{QC_y}$ **includes** $\mathbf{O_x}$**.**

At least one quorum-server processed both $\mathbf{O_x}$ and $\mathbf{O_y}$ since each were processed by Q quorum-servers.

Quorum CQ_x was *complete-quorum* before O_y started → Any overlapping quorum-server *must have processed O_x before O_y*

This is all you can know from multiple interleaving quorums.

What can we do with that?

A change to an item is communicated with a quorum operation to the subsystem. Consider operation O_x. As a worker client performs operation O_x, it goes through three stages:

- **$\mathbf{O_x}$ is not included (visible):** Before the operation starts.
- **$\mathbf{O_x}$ is intermittently included (visible):** It's incomplete.
- **$\mathbf{O_x}$ is included (visible):** Seen by all complete-quorum ops.

This is true both when the item is created and when it retires. It flutters into existence and out of existence.

[22]For example, an operation called *Max* accepts a new value each time it is called. Any *complete-quorum* operation will be included in the *Max* seen by subsequent quorum-complete operations. An *incomplete-quorum* will be *intermittently included.*

Example: Suppose the *Max* seen so far is 1,000. If a client performs *Max* operation O_m for a value of 1,000,000 and crashes when only K<Q quorum-servers have processed it, then operation O_m is an *incomplete-quorum-operation.* An average of K of N subsequent complete-quorums will include O_m in their combined response, resulting in a value of 1,000,000. The remaining (N-K) of N complete-quorums will not include O_m in the combined response, resulting in a value of 1,000.

6 CONFLUENCE: ADDING CLARITY TO FUZZY QUORUM

We use quorum to manage the state of items within our coordinator, catalog, and log-storage subsystems. Each of these has many items it manages and they each have well defined lifecycles.

As discussed above, quorum is both awesome and awful:

- **Quorum is awesome:** It's fast even when servers jitter.
- **Quorum is awful:** It's messy, fuzzy, and jumbled.

This section examines how we can make sense of our messy, fuzzy, and jumbled operations provided by quorum. We introduce *confluence*, a property of some computations where their outputs are independent of the order of their inputs. They offer *deterministic outputs* even with *non-deterministic execution.* With confluence, it's always OK to *combine the outputs.*

Next, we look at the limitations of confluence. Confluent programs can answer questions about what *does exist* but not about what *does not exist.* We examine the notion of *sealing a set of confluent operations* to provide *does not exist* computations.

Following this, we **combine both quorum and confluence**. We believe this is a *novel contribution* that allows us to *remain lively* and *still reason about does not exist.*

We see how changes to items in our subsystems gradually become visible to the rest of the system with *fuzzy edges* to these changes. We complete this section with an example of how we can cope with fuzzy transitions due to quorum.

6.1 Confluence of Operations on Items

Confluence is a property of some computations[32]. Components are confluent if they produce the same outputs for all orderings of their inputs. Confluent components compose and many replicas of the same component behave like one replica.

It has been proven that to be *confluent* (i.e., reorderable inputs), there must be *some logical abstraction of monotonic order*[3, 4]. In other words, we see a predictable (and reorderable) outcome of *outputs* to requests if and only if there's some way of looking at all the *inputs* as having some form of logical order.

Confluence (using seniority) is the lynchpin of this design. Quorum avoids jitter but creates a jumbled execution. Confluence (via seniority) resolves the confusion of the jumbled execution.

We use three tricks to build *confluent operations*:

- **Disjoint:** If two operations have nothing to do with each other, they are reorderable.
- **Sent by a single client:** Operations sent from a single client can have an order defined by that client[23] that the existing seniority within the log established before the failure of the logging worker provides a monotonic seniority that allows confluent log repair by many competing servers.
- **Some intrinsic partial order:** Input operations have some semantic logical ordering[24]. That same logical partial order is present in the outputs of the confluent server.

[23]For example, when a single worker server writes to its private log-window, it is the single client. Its seniority is assigned by the logger. Even if the appended fragments are seen in different execution orders at each log-storage-server, log entries retain their seniority. Each log-storage-server may see its own order of the writes. Combining results from the quorum gives the correct answer. Furthermore, we will see in §15.1

[24]This is the *Logical Monotonicity* referred to by the CALM Theorem[32] (Consistency As Logical Monotonicity). For our system, this is *seniority.*

Seniority* provides our database with a *logical order

We have three subsystems using quorum (See §5):

- *Coordinators:* Control transactions using quorum and allow reorderable operations on them.
- *Log-Storage Servers:* Allow reorderable appending of fragments to tolerate jitter and improve performance.
- *Catalogs:* Register changes to persistent data files and extents while tolerating reordering of the operations.

***Seniority* provides monotonic order to their items:**

- **Transaction seniority → transaction partial order**
  - *Conflict detection (based on partial order)*
  - *Whereabouts (based on partial order)*
  - *Transactions retire (based on seniority)*
- **Fragment seniority → order of fragments in log**
  - *Parallelism:performance (reorder by seniority)*
  - *Replica holes from jitter (clean up by seniority)*
  - *Log repair of fragments (repair by seniority)*
- **Flush file seniority → order for each worker**
  - *Flushes are visible (based on seniority)*
  - *Flushes retire (based on seniority)*
- **Merge file seniority → merge order (per key-range)**
  - *Merges are visible (based on seniority)*
  - *Merges retire (based on seniority)*

6.2 Confluence: Deterministic Outputs with Non-Deterministic Execution

Executing operations happen at many different quorum-servers in a *non-deterministic order of execution*. Who knows when, where, and how often they will be run? We need to get correct behavior from our coordinator, catalog, and log-servers when *at least Q of N quorum-servers execute each operation at least once.*

Confluent operations produce deterministic outputs even when executed non-deterministically [1, 2, 32, 38]. Their outputs can be combined by clients to produce a unified set of responses, independent of their order of execution or how many times they executed.

6.3 Confluence: OK to Combine Outputs

Confluence is exactly the right property to use within quorums. You can mix-and-match the outputs from the quorum servers and, as long as one of the quorum-servers has an output, it will be passed on by the quorum-client.

Recall that a program is *confluent* if and only if it is *monotonic*, for *some logical abstraction of monotonic order.*

DEFINITION[32]: A program P is *monotonic* if:

for any input sets S, T where S ⊆ T,
P(S) ⊆ P(T)

In other words, if T's inputs *include* S's inputs, then T's outputs will *include* S's outputs.

A perfect match for quorum-clients combining results!

*Quorum-clients **include outputs** from quorum-servers!*

Seniority is our *logical abstraction of monotonic order*

6.4 Confluence and "Does Exist"

Confluent operations can answer questions about the *existence of something* but not the *absence of something.*

- **Does exist:** Describes a problem whose answer depends on the *existence of an operation.*
- **Does not exist:** Describes a problem whose answer depends on the *asserting that an operation does not exist.*

When a set of servers are performing confluent operations, they combine their results to get the complete answer. No server knows what has transpired at any other servers.

Clients don't know what's happened at all quorum-servers.

After receiving Q of N response, it knows what happened at Q but not at the other N-Q quorum-servers.

A client *does know* that an operation *does exist* if was successfully processed by Q quorum-servers. Since a set of executing confluent operations continuously grows by adding to the set, we can see if an operation *has already occurred* (i.e., it does exist).

6.5 Confluence, Sealing, and "Does Not Exist"

It's hard to tell when something *hasn't happened.* The knowledge is spread widely and you don't know the state of the other replicas. Alvaro [1] discusses two broad approaches: *sealing* and *windowing.*

Sealing input streams into confluent servers. When no more inputs are arriving, we can answer questions about what's not present in the inputs. Sealing requires coordination to close down the inputs. Shutting down the world can be disruptive. The seal operation is ordered. *Seal is not confluent!*

Windowing input streams into confluent servers. Input streams can be broken up into *batches.* Batches may be in some grouping other than a temporal order, allowing more flexibility and, perhaps, less disruption. Can we create windows of operations to know what's *not happening* in addition to *what's happening*?

Is there a gradual way to answer "does not exist" questions?

6.6 Combining Confluence and Quorum

Quorum confluence is the combination of both *quorum* and *confluence.* It can answer *does not exist* questions albeit with some fuzziness in it behavior during transitions in its state.

Quorum confluent solutions do not jitter: Quorum ensures they ignore up to F jittery servers.

Quorum confluent solutions understand *does not exist.* Items can be retired using their *seniority*:

- *Seniority is monotonic:* We use the logical time of commit.
- *Seniority can be retired:* As quorum-clients and quorum-servers track retirement age, retired items can be reclaimed.

It reliably advances retirement in a distributed subsystem.

Quorum confluence is both:

- **Live and fast** in the face of bounded failures, and
- **Can manage a complex distributed system** with does not exist.

To our knowledge, this is novel.

We argue that Paxos is a special usage of quorum confluence.
Paxos uses quorum & confluence, removing the *fuzziness* to provide *linearizable behavior* at *the expense of liveness.*

6.7 Seniority: Gradual Visibility and Retirement

As discussed in §5.6, operations are visible in a fuzzy fashion when using quorums. They change over stages:

- *O_x is not included (visible):* Before the operation starts.
- *O_x is intermittently included (visible):* It's incomplete.
- *O_x is included (visible):* All later complete-quorum ops see it.

When an operation O_x is *quorum-complete*, all subsequent *quorum-complete operations* O_y will include O_x.

We can control the visibility of changes *albeit with fuzzy edges.*

6.8 Example: Coping with Fuzzy Visibility

Let's consider an example showing the fuzzy creation of items within the *coordinator server* as it deals with *transactions.*

EXAMPLE: Fuzzy creation of transactions:

Consider two transactions:
T_1 & T_2 both trying to commit with seniority S_a

Two permission-to-commit operations are launched:

- Worker $W_1 \rightarrow T_1$ at S_a
- Worker $W_2 \rightarrow T_2$ at S_a
- T_1 and T_2 have conflicting updates.

It is fuzzy which of three outcomes will occur:

(1) **Outcome: T_1 commits and T_2 aborts**
- *T_1 arrives at >= Q coordinator servers first.*
- *T_2 arrives at < Q coordinator servers first.*

(2) **Outcome: T_1 aborts and T_2 commits**
- *T_1 arrives at < Q coordinator servers first.*
- *T_2 arrives at >= Q coordinator servers first.*

(3) **Outcome: Both T_1 and T_2 abort**
- *T_1 arrives at $Coord_x$ before T_2,* and
- *T_2 arrives at $Coord_y$ before T_1*

Any of these three outcomes is correct.

7 JITTER-FREE SNAPSHOT ISOLATION & 3AZ

This section *sketches how we avoid jitter* within the functionality of the database. We sketch enough here to understand the basic message of this paper.

We demonstrate that it is possible build a complex system like a distributed database that does not jitter
Quorum, confluence, seniority, and retirement make it happen!

We will first sketch how we can avoid jitter in our hypothetical database. After that, we will explain the required quorum set (i.e., the require size of N_{Coord}, N_{Cat}, and N_{Log}). Each of these subsystems has different availability requirements. We described the count and deployment of servers to tolerate an AZ+1 failure within a 3AZ environment.

7.1 Jitter-Free Pieces of the Puzzle

This section sketches the operations performed by our *decoupled transactions database.* At a high-level we discuss how they can avoid jitter and remain a robust system providing *snapshot isolated transactions* to applications. More details for each of these sections can be found in the appendices.

See **§14: (Appendix C: Jitter-Free Database)** and **§15: (Appendix D: Jitter-Free Log-Repair)**

Jitter-Free TX Commit, Begin, & Retire: Transaction commit and begin avoid jitter by using quorum to the coordinator subsystem. They align their *seniority of their work* by *proposing a seniority* that is *confirmed* quorum of coordinator servers. Transactions retire when they are older than the *oldest snapshot seniority* and their committed changes are successfully in the LSM.

→ *Transaction commit, begin, and retire do not jitter.* See §14.1.

Jitter-Free Flushing to the LSM: Flush is performed by each worker after it accumulates a lot of changes in its memory. By flushing these to the LSM, the new records can be read from shared storage. If the worker does get slow, there is risk that the flush itself can stall. In that event, other workers would notice the catalog says the integrated LSM merging of recent changes is stuck. The workers can kill the slow server and rapidly read its log and do the flush for the slacking worker.

→ *Flush can be rapidly performed by other workers*
→ *Catalog accesses uses quorum to avoid jitter.* See §14.2.

Jitter-Free Merging of LSM Data Files: Any (and all) workers contribute to the constant need to merge the LSM. Assignments for new merge data files are acquired by consulting the catalog.

→ *Merge does not jitter (idempotent & retried data file operations)*
→ *Catalog work uses quorum* See §14.3.

Jitter-Free Reading of LSM Record-Versions: Reading the LSM comprises finding needed data files, reading them from shared storage, finding records in the files, merging records from LSM levels, and returning the record-versions matching snapshot read rules.

→ *Reading from the LSM is jitter-free* See §14.4.

Jitter-Free Reading Recent Records: Done in one of 3 ways:
(1) *Direct message to other worker*
(2) *Direct message to log-following worker*
(3) *Fencing, repairing, & reading worker's log*

→ *The first two may jitter.*
→ *We fall back to jitter-free log repair and read.*
See §14.5, §15.13, §15.3, §15.4, §15.9, & §15.12.

Jitter-Free Log Quorum: The use of log quorum is both to have *enough replicas* to ensure durably logged data and to *know when you don't have enough replicas.*

→ *Logging to a quorum is jitter-free*
→ *Finding missing fragments with quorum is jitter-free* See §15.2.

Jitter-Free Liveness Check via Log: If worker W_a thinks worker W_b may be sick, it can verify its health or sickness. The catalog says where W_b is logging. W_a can ask the log-server replicas if W_b has been logging OK.

→ *Asking the catalog where W_b is logging is jitter-free*
→ *Asking the log-servers for W_b's log is jitter-free* See §15.3.

Jitter-Free Log Seniority (from Logger): Seniority is assigned to pieces of the log by the logging worker-server. This is important to ensure log-repair can be concurrent and jitter-free.

→ *Assigning seniority to log pieces is jitter-free.* See §15.1.

Jitter-Free Concurrent Log Fencing: Fencing worker W_a may be done by W_b using the catalog and log-storage replicas. We ensure many workers ($W_b \cdots W_n$) may concurrently fence W_a's log.

→ *Concurrent fencing in the catalog is jitter-free*

→ *Concurrent fencing in the log replicas is jitter-free* See §15.4.

Concurrent Repair of the Log: We ensure that the log of a sick and fenced worker W_a may be concurrently repaired by multiple workers ($W_b \cdots W_n$).

→ *Concurrent repair in the catalog is jitter-free* See §15.9.

Jitter-Free Retry to Log-Followers: Log-followers are an option to have another worker W_b follow the log replaying updates by work W_a. W_b *is usually almost up to date* and can supply recent record-versions from W_a's updates.

→ *Log followers may be jittery!* See §15.13.

→ *We fall back to jitter-free log repair and read.* See §15.12.

Bounding the Pain When a Worker Is Sick: All steps involved in detecting a worker W_a is sick, fencing their log, repairing their log, and reading their log to see any committed work they have done must be correct and fast when performed concurrently by multiple other workers ($W_b \cdots W_n$).

→ *Removing a worker & reading their log is jitter-free.* See §15.12.

7.2 Server Types & 3AZ Quorum Requirements

In designing a system to tolerate jitter, we must examine the needs of different *server types* as they continue operating in a jittery environment. Immutable storage, appending to logs, tracking system state, and checking for transaction conflicts each have their own unique needs. We first look at the work performed by each type of server and its required availability. Then, we examine the quorum requirements of each different server type.

Worker servers largely operate independently of each other. There are no considerations of quorum for workers.

Coordinator servers [25] provide light-weight knowledge about ongoing transactions, their possible conflicts, and their recently committed record-versions. This must be up and functioning even if we see AZ+1 jittery servers.

9 Coordinator Servers → 3 in each AZ.
Send to 9 Coordinator Servers → Wait for 5 Responses
AZ+1 Jittery Coordinator Servers (i.e., 4 total) → a Zippy Quorum (with 5 responses)

Data-storage-servers[26] store large immutable *data files*, each with a unique *data-file-id* (a UUID[53]). These data files are used to implement the LSM. We don't need to update existing data files and don't need a quorum[27].

Each *Data-File* is placed in 3 Data Storage Servers → 1 in each AZ
Write to 3 Data Storage Servers → 1 in each AZ (if possible)
Read from 3 Data Storage Servers → take the first response

Log-storage-servers support high-performance appending of a worker's transaction log records. Worker server logging is a challenging thing. Not only do we need to ensure correct repair of the log-window after a worker server crashes, we must do so even with AZ+1 jittery servers[28].

In contrast to *the coordinator* and *the catalog*, each log-window does not need to perform new changes after losing AZ+1 of its log-storage-servers. Rather, we view the log-window as read-only[29].

We ensure that *log-repair* can reliably repair the log-window's history without jitter[30]. If we do not insist a *log-window* can continue appending new log records, we can reduce the cost of each log. Some AZ+1 failures may cause the loss of unlucky worker servers whose log-window is now read-only. Other worker servers will continue to take traffic even if a few workers get stuck and die.

High-volume transaction logging → expensive replicas
Each Log-Extent → 6 Log Storage Servers → 2 per AZ
Append fragment: write to 6 → wait for 4 responses
Log Repair: (read >= 3 replicas) If 3 are missing → Fragment not committed → OK to end recovered log here
Only replicate log 6 times → Less expensive logging → ***Log extents may be* read-only *after AZ+1 failure*** → More complex log repair
See §15.2: (Jitter-Free Log Quorum)

Catalog servers describe where the database data is stored. They are responsible for tracking active worker servers, the set of data-storage-servers keeping our data files, the set of log-storage-servers keeping our data files, information about workers' logs, knowledge of the data extents used for various keys and levels within the LSM. See 5.1.

The state retained by the storage catalog must be preserved even if the database itself crashes. Hence, each replica of the catalog uses cloud-native storage to retain its state[31]. *The catalog is essential to the operation of a decoupled transactions database. It must continue operation with AZ+1 jittery weirdness.*

[25] See §4 for info about *Coordinators*, their role in the system, and how they avoid jitter.

[26] See §2.5 for a discussion of *data storage servers* and how they can keep immutable data-files while avoiding jitter.

[27] The data storage servers are designed to provide suitably large durability guarantees for data-files (minimizing data loss). They do not, however, need to manage crisp update semantics (since immutable data is not updated)[28]. For durability, data-storage-servers triple-replicate data-files, placing one replica per AZ. Reading a data-file from a single replica provides the correct contents. Writing new data-files must ensure three replicas are created, hopefully in separate AZs.

[28] See §14.5, §15.13, §15.3, §15.4, §15.9, & §15.12.

[29] This is the same as in *AWS Aurora*[46]. Aurora is designed to support *reading* a log after losing AZ+1 servers. Aurora cannot continue to *write* to the same set of its *storage servers* under these assumptions. This was not a design goal for Aurora.

[30] In Aurora, there is a single log-repair server, a common practice in database systems. Because *Decoupled Transactions* assumes we cannot rapidly select a *single centralized log repair server*, our design assumes idempotent repair by *one or more* worker servers. When any one completes repair, the log is permanently recovered.

[31] Each replica of the catalog must use cloud-native storage that has jitter correlated to the individual replica. For example, using storage in the replica's local AZ would not be correlated to failures and/or jitter within a different AZ.

Catalogs and coordinators are separately implemented:

- **Catalogs** must ensure that changes to the database state are recorded durably. These change are relatively rare and need not be as fast as the coordinator operations.
- **Coordinators** have ephemeral state that does not need to survive crashes of the database. They are faster and lighter-weight that catalog changes.

One catalog could reasonably support multiple databases.

9 Catalog Servers → 3 in each AZ.

Send to 9 Catalog Servers → Wait for 5 Responses

AZ+1 Jittery Catalog Servers (i.e., 4 total)
→ Quorum of 5 non-jittery responses

8 DISCUSSION

Distributed systems are complex and important for many uses[32]. To manage their resources, distributed system track when internal resources *do exist* and also when they *do not exist*. Knowing when something is *not in existence* is an essential part of a stable system. Until now, this required some centralized authority. This frequently uses a consensus based system such as Zookeeper[9] to accurately know its *centralized source of truth*. Centralizing is a risky proposition when servers and networks are jittery (or intermittently available) because this centralized authority can, itself, become intermittently available.

We introduce the notion of *quorum confluence* to allow us to know when resources *no longer exist* without the need of a centralized server. This knowledge is resilient and jitter-free.

Another pervasive challenge in distributed computing has been the rapid detection of server failure. We describe how to *rapidly detect and remove* servers from the system, assuming these servers log their work to a distributed log.

Quorum consensus is an important tool in this proposed *jitter-free database*. It can also be used to eliminate jitter from a broad class of distributed systems, including the root authority behind a datacenter and its components.

Not only are database systems complex to implement, they mask complexity from applications. This provides huge leverage to our systems as applications provide end user functionality with greater ease. Tolerating jitter is hard. Using a jitter-tolerant database can be transparent and easy.

8.1 Jitter-Free Knowledge of "Does Not Exist"

Quorum confluence allows systems to manage resources without jitter. This is, to the best of our knowledge, novel. It can dramatically improve the robustness and availability of distributed solutions.

Other systems have used quorum over distributed logs to find the tail-of-the-log after a crash. Many systems need *"does not exist"* to manage resources with more complex needs. Managing resources with lifetimes controlled by multiple owners poses different challenges. Combining both *seniority* and *confluence* allows systems to rein in some aspects of their work via *seniority* while allowing other aspects to continue ahead unabated by leveraging *confluence*.

Seniority can work cooperatively with *quorum* to control the *retirement* of parts of the system. It does so while ensuring other parts of the system remain lively and jitter-free.

While much research remains to be done, this appears to offer the hope of significantly better distributed systems in the future.

We get jitter-free retirement of distributed resources

8.2 A Jitter-Free Uncaused Cause

Managing data centers can be especially challenging, both when *booting them to life* and when *running under congested circumstances*. An existential problem is that of the *uncaused cause* or root authority of all truth in the datacenter. See §1.8.

Until now, the root authority of these systems built on *linearizable*[33] behavior. This inherently means it has intermittent availability as it jitters or locks up. While it is easier to reason about linearlizable behavior, data centers and databases are special things. Perhaps we should work harder to design this root authority to be highly available and not centralized.

Further research may yield
jitter-free & highly available management
of complex systems, including data centers

8.3 Flipping on FLP: Determining Dead vs. Sick

While FLP, the *Fischer-Lynch-Patterson Impossibility Result* remains undeniably true, we can build systems that sidestep the problem.

FLP[23] assumes that *all visibility into the server is via direct messaging*. For cases, like database worker-servers, where the server must *log its progress to a replicated log before responding*, we can see the health of the server by looking at its log.

The log uses quorum and we can check its progress without jitter. When (N_{Log} - Q_{Log} + 1) log-servers *respond to messages* saying no progress, we know it's not healthy[33].

We can ***deterministically know*** *if a server is* ***sick or dead***
in ***an asynchronous network***

8.4 Jitter-Free Snapshot Isolation

Snapshot isolation combines *snapshot reads* and *conflict detection*.

Snapshot reads can be jitter-free. Both *snapshot time* and its *whereabouts* can be found by a quorum of coordinator servers. Workers respond to RPCs with recent changes unless they are sick. If so, we read their changes from their logs. Quorum (both *catalog servers* and *log extents*) allows us to detect slow servers, fence and repair their log, and read committed record-versions if needed.

Conflict detection before commit is jitter-free. Prior to committing a transaction, the worker requests *permission-to-commit* from a quorum of coordinators. If a proposed update conflicts with any other transaction since its snapshot time, it will not commit.

Consensus means agreeing on the next new value across a set of servers. Each time a transaction updates record R_X, that is a form of consensus across the database cluster. Unfortunately, consensus has *liveness problems* and can't always complete in bounded time.

Snapshot isolation semantics ensure updates are not committed if modified since snapshot time. Attempting rapid updates to the

[32] As these systems become ever more important, we must ensure they are rock solid and don't flutter and flounder with *three sheets to the wind*[44] in stormy weather.

[33] Extent log-server replicas are placed to avoid correlated jitter. The probability of (N_{Log} - Q_{Log} + 1) log-servers jittering is *profoundly less likely* than 1-of-1.

same record(s) may thrash and see little progress. Over years, most applications atop these databases evolve to avoid this.

Decoupled Transactions ensures *liveness problems only impact the rapid updates to the same record(s).* In practice, mature snapshot isolation applications rarely fight over common records.

Rapid updates to a common record may see contention
The database remains jitter-free and open for business

8.5 What's New in Decoupled Transactions?

This hypothetic distributed database combines new ideas:

(1) **Jitter-free management of resources:** Complex systems can manage resources without centralized authority.
(2) **Partial transaction order via future time & quorums:** Pick a future transaction time and align with quorum.
(3) **Jitter-free conflict detection:** Possibly conflicting transactions are detected by at least one coordinator.
(4) **Jitter-free checking of server health:** For servers that log before responding, its quorum log shows its health.
(5) **Jitter-free concurrent log repair:** If we need to avoid jitter, we can'f select a single server to do anything, including log repair. Quorum based monotonic and idempotent repair of a distributed log empowers rapid visibility into committed work even within a sick worker.
(6) **Quorum-based catalog for log & key-value metadata:** A quorum catalog ensures jitter-free metadata. Jitter-free metadata and jitter-free storage means low tail latency logging, log-repair, and record reads from key-value storage.
(7) **Avoiding jitter using transactions, logs, & quorums:** We combine many aspects of transactions, log-repair, and quorums to build a transaction systems that avoids jitter.

9 RELATED WORK

This paper brings together thoughts from many different areas of research. We first look at responsiveness and why we don't trust timeliness in the datacenter. Next, we look at quorum and the ways it has been used in the past and how these contribute to our proposal. Finally, we contrast earlier work on confluence, its strengths and limitations, and how adding both quorum and retirement to confluence is so important.

9.1 Responsiveness and Lack of Responsiveness

A great deal of recent work has shown that modern cloud data centers are increasingly unpredictable. Responses to work are only probable in their timing. This thought process led us to simply believing that a slow answer can not be decisive. We must *ignore missing responses* and *act when enough responses have been received.*

Fault models for distributed computation lead us to conclude we can't know the state of other servers unless we actively get a message back from them. The absence of a response doesn't help!

Batch computation uses redundancy to bound latency. MapReduce (and other systems) leverage both timeouts and proactive redundancy to cope with *stragglers*[5, 6, 18, 21, 54].

Online work also uses redundancy to bound latency. Dean and Barosso [20] showed how to time-out and retry requests to ensure much better SLOs. Mogul, Isaacs, and Welch [39] describe many aspects of SLOs (Service Level Objectives) as well as the many challenges seen in responsiveness of deployed systems.

Gray failures are compounding our challenges. The recent work on gray failures [22, 27, 34] underscore that even when all is going well in your datacenter, that may not last. Increasingly, servers and networks may simply *go slow* rather than *failing fast.*

Servers and networks may be arbitrarily slow. These works, along with many others, made two things things very clear:

- *Messaging is not synchronous:* Unlike most classic first party data-centers, late responses from servers (and the networks connecting them) are probable and increasingly variable[34].
- *We can't use time-out to define failure:* Centralized leaders need rapid failure detect to somehow get a new leader. This is problematic when we cannot predict latency[30].

Omitting omissions[35]: We ignore missing responses! The absence of a response can be a *hint* but *it cannot be used to remove a server.* We combine:

- *Proactive redundancy:* Launch a lot of requests
- *Enough is enough:* Continue when enough responses come ... don't wait for special ones.

By combining quorum, confluence, seniority, and retirement we can make decisions with *enough responses*, not *a special set of responses.*

9.2 Related Work: Quorum

Quorum is the technique to know *when enough stuff has happened.* It allows us to move onto a new phase of some computation without stalling for jittery servers.

Quorum for distributed computing. Thomas[45] introduced voting algorithms for replicated databases. Gifford[25] extended this with weighted voting adding more flexibility. Each of these proposals provided consistent database updates and tolerated failures of a subset of the quorum of *storage servers.* These early (and seminal) uses of quorum ensured correctness of atomic and serializable consistent updates to replicated data, tolerating failures of up to (N-Q) of N replicas.

This usage of quorum depends on monotonically increasing version numbers assigned by the clients. Clients can compete for the largest version number and thrash retrying ever higher versions. These usages of quorum are resilient to the *failures in the storage* but do not address *liveness in the allocation of versions.*

Quorum for consensus. Lamport defined *Paxos*[36], a technique for safely gaining *consensus* across many servers. He also provided a proof for its correctness. Paxos ensures a single new value is agreed across a quorum of servers. This is a linear order of new values. Paxos is *safe* because any new agreed value will be seen by all participants. It is not *live* since agreeing on a new value cannot be guaranteed in bounded time.

Paxos leverages quorum and confluence but does so in a jittery fashion. Confluence comes from the *order of proposals within the Paxos algorithm.* Paxos uses both quorum and confluence but makes a different tradeoff than we do here. We choose to have *fuzzy state transitions* as items come into existence and retire. Paxos chooses to have crisp and clear transitions to new values at the expense of introducing liveness challenges[36].

[34] "Toto, I have a feeling we're not in Kansas, anymore!"[49]
[35] See §12: (Appendix A: Building on a Jittery Foundation).
[36] Jitter is one form of these liveness challenges.

Quorum logging and the tail of the log. Quorum has been used before to determine when things *do not exist* in a log. *Apache Bookkeeper*[7] and *AWS Aurora*[46, 48] each use quorum to know when a log append *does not exist*. Both of these solutions resolve *does not exist* for monotonic sequences originating at a single source of change[37]. They do not address the management of arbitrary resources shared within a system.

Seniority extends the notion of versioning. Rather than a single centrally assigned version number, seniority can be assigned in many ways. Each item (see §3) is gets its seniority [38] based on the rules for that type of item.

***Seniority* can be assigned using different techniques:**

- *Seniority assigned by quorum:* For example, the assignment of seniority to transactions (see §4)
- *Seniority from a single source:* For example per-worker log-windows and per-worker flush data files (see §14.2, and §15.1)
- *Seniority assigned in complex derived ways:* For example, the seniority used by merged data files as they coalesce record-versions from other data files in the LSM (see §14.3)

Seniority and retirement: Items can come and go. With retirement, asynchronous (and distributed) algorithms can determine the *retirement age* of items in many independent domains. These combine together to ensure a robust and jitter-free[39] state transition for managed items.

9.3 Related Work: Confluence

Confluence defines reorderable and jitter-free computation.

Confluence provides freedom from coordination. Coordination is not required for some classes of programs. This empowers predictable composition of parallel and out-of-order work. Confluence[1, 2, 29, 38] and the reorderability it provides are an important backbone of scalable computing. MapReduce, sort-merge, and much more are predicated on confluence.

Confluence requires logical monotonicity. Hellerstein and Alvaro clarify the type of computation we see when programs are *confluent* in *CALM: Consistency as Logical Monotonicity*[32]. They show that confluent programs avoid coordination and that confluence happens if and only if the program leverages some form of *abstract logical monotonicity*. Ameloot et al (2013)[4] and Ameloot et al(2016)[3] prove theoretically that coordination-free computation and confluence are the same. They also prove that confluence depends on *logical monotonicity based on some abstraction.*

Confluence allows answers to when *something does exist.* Confluence, by itself, only empowers some classes of computation. It can determine results based on existing facts without the need for coordination. Hellerstein and Alvaro state confluence programs can calculate what *does exist* but not what *doesn't exist.*

We extend confluence to include *does not exist.* By combining confluence with *quorum*, *seniority*[40], and *retirement*[41], we can show when items in the system *do not exist.* This does require coordination but that coordination is asynchronous and jitter-tolerant.

Consider the separate contributions of these mechanisms:

- *Quorum:* Tells you when something will be remembered.
- *Confluence:* Tells you what may be reordered.
- *Seniority:* Tells you a partial order across items.
- *Retirement:* Uses seniority to say *the item does not exist.*

The combination of all of these can be used to build a scalable and *jitter-free* distributed system. This includes managed items both *coming into the system* and *leaving the system.* Both *does exist* and *does not exist* are supported. While there is coordination across the members of the quorum, it is jitter-free (up to a certain limit of jittery servers in the system).

10 CONCLUSION

This thought exercise started to understand the nature of jitter and its implication on database implementation. What makes databases so susceptible to jitter? Each step of the way, there were more subtle challenges to conquer in the mental exercise of dissecting dependencies. Soon, it became clear that the challenges posed here are not specific to distributed databases.

Coordination[31] remains our biggest challenge in complex systems. When that coordination can be *fuzzy over time*, our systems become more robust.

11 ACKNOWLEDGEMENTS

Many people have helped tremendously in both developing these ideas and creating this paper. I am especially grateful to many of my friends and colleagues for openly sharing their befuddled expressions of confusion. Their lack of understanding forced me to dig into the essence of proposed solution. That dramatically clarified my thinking and, in turn, this paper.

I am grateful to: Shyam Antony, Vaibhav Arora, Phil Bernstein, Subho Chatterjee, Terry Chong, Allen Clement, Natacha Crooks, Dave DeHaan, Haiyan Du, Tim Eads, Thomas Fanghaenel, Shel Finkelstein, Joe Hellerstein, JV Jujjuri, Jim Mace, Jamie Martin, Mark Mears, Kaushal Mittal, Jacob Park, Bryan Pendleton, Akshay Manchale Sridhar, Brian Toal, Justin Wang, and Nat Wyatt for their thoughtful comments, engaging arguments, and feedback.

Special thanks to David Lucey for lending his networking expertise to help me articulate why messaging within data centers is largely a game of chance. Finally, Peter Alvaro has taught me a huge amount about many things. We've spent many hours discussing confluence while behaving as[42] two kids in a candy store.

[37] Log writes originate at one server and are written to a quorum of replicas.

[38] Seniority is comparable to the versions used by Thomas and Gifford, LSNs in Aurora's log and the entry-id in Bookkeeper's log (called a *ledger*).

[39] Assuming our assumptions for non-correlated jitter hold.

[40] Seniority is our form of *logical monotonicity of things entering the system.*

[41] Retirement provides *logical monotonicity of things leaving the system.*

[42] Being aware of the difference between a simile and a metaphor, this seems apropos.

REFERENCES

[1] P. Alvaro. *Data-centric Programming for Distributed Systems.* PhD thesis, University of California, Berkeley, Berkeley, CA, USA, 2015. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-242.pdf.

[2] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[3] T. J. Ameloot, B. Ketsman, F. Neven, and D. Zinn. Weaker forms of monotonicity for declarative networking: A more fine-grained answer to the calm-conjecture. *ACM Trans. Database Syst.*, 40(4), dec 2015.

[4] T. J. Ameloot, F. Neven, and J. Van Den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2), may 2013.

[5] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 185–198, Lombard, IL, Apr. 2013. USENIX Association.

[6] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, Oct. 2010. USENIX Association.

[7] Apache bookkeeper. https://bookkeeper.apache.org.

[8] Apache bookkeeper: Lac. https://bookkeeper.apache.org/distributedlog/docs/latest/user_guide/design/main.html.

[9] Apache zookeeper org. https://zookeeper.apache.org.

[10] R. Barker and P. Massiglia. *Storage Area Network Essentials: A Complete Guide to Understanding and Implementing SANs.* Wiley, paperback edition, 11 2001.

[11] J. Bartlett, J. Gray, and B. Horst. Fault tolerance in tandem computer systems. https://www.hpl.hp.com/techreports/tandem/TR-86.2.pdf, 1986.

[12] J. F. Bartlett. A nonstop kernel. *SIGOPS Oper. Syst. Rev.*, 15(5):22–29, Dec. 1981.

[13] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 1–10. ACM Press, 1995.

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

[15] M. Brooker, T. Chen, and F. Ping. Millions of tiny databases. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 463–478, Santa Clara, CA, Feb. 2020. USENIX Association.

[16] G. Chapman, J. Cleese, and E. Idle. Monte python and the holy grail: Not dead yet. https://www.youtube.com/watch?v=uBxMPqxJGqI, 1975.

[17] D. E. Comer. *Computer Networks and Internet (6th Edition)*, page 508. Pearson, 2015.

[18] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 21, USA, 2010. USENIX Association.

[19] F. Cristian. Understanding fault-tolerant distributed systems. *COMMUNICATIONS OF THE ACM*, 34:56–78, 1993. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.30.591&rep=rep1&type=pdf.

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

[21] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[22] T. Do, M. Hao, T. Leesatapornwongsa, T. Patana-anake, and H. S. Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[23] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[24] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, Renton, WA, Apr. 2018. USENIX Association.

[25] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM.

[26] J. Gray. Tandem tr 85.7 why do computers stop and what can be done about it? https://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf, 1985.

[27] H. S. Gunawi, R. O. Suminto, R. Sears, C. Golliher, S. Sundararaman, X. Lin, T. Emami, W. Sheng, N. Bidokhti, C. McCaffrey, D. Srinivasan, B. Panda, A. Baptist, G. Grider, P. M. Fields, K. Harms, R. B. Ross, A. Jacobson, R. Ricci, K. Webb, P. Alvaro, H. B. Runesha, M. Hao, and H. Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Trans. Storage*, 14(3), Oct. 2018.

[28] P. Helland. Immutability changes everything. *Commun. ACM*, 59(1):64–70, 2016.

[29] P. Helland. Don't get stuck in the "con" game: Consistency, convergence, and confluence are not the same! eventual consistency and eventual convergence aren't the same as confluence, either. *Queue*, 19(3):16–35, jun 2021. https://queue.acm.org/detail.cfm?id=3480470.

[30] P. Helland. Fail-fast is failing... fast! changes in compute environments are placing pressure on tried-and-true distributed-systems solutions. *Queue*, 19(1):5–15, Feb. 2021.

[31] P. Helland. I'm so glad i'm uncoordinated, February 2021. https://pathelland.substack.com/p/i-am-so-glad-im-uncoordinated.

[32] J. M. Hellerstein and P. Alvaro. Keeping calm: When distributed consistency is easy. *Commun. ACM*, 63(9):72–81, aug 2020.

[33] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[34] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 150–155, New York, NY, USA, 2017. Association for Computing Machinery.

[35] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[36] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[37] C. Luo and M. J. Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, Jul 2019.

[38] W. R. Marczak, P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Confluence analysis for distributed programs: A model-theoretic approach. In *Proceedings of the Second International Conference on Datalog in Academia and Industry*, Datalog 2.0'12, page 135–147, Berlin, Heidelberg, 2012. Springer-Verlag.

[39] J. C. Mogul, R. Isaacs, and B. Welch. Thinking about availability in large service infrastructures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 12–17, New York, NY, USA, 2017. Association for Computing Machinery.

[40] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[41] Oracle. Oracle database concepts 10g release 1 (10.1) chapter 13 : Data concurrency and consistency — oracle isolation levels, 2003.

[42] B. Reichenbach. Cosmological argument. In *The Stanford Encyclopedia of Philosophy (Spring 2021 Edition), Edward N. Zalta (ed.).* The Metaphysics Research Lab, Stanford University, 2021.

[43] J. Tate, P. Beck, H. H. Ibarra, S. Kumaravel, and L. Miklas. *Introduction to Storage Area Networks.* IBM Redbooks. IBM, December 2017.

[44] TheFreeDictionary. Three sheets to the wind. https://idioms.thefreedictionary.com/three+sheets+to+the+wind, 2021.

[45] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, jun 1979.

[46] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. 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, page 1041–1052, New York, NY, USA, 2017. Association for Computing Machinery.

[47] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. 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, New York, NY, USA, 2017. ACM.

[48] 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, page 789–796, New York, NY, USA, 2018. Association for Computing Machinery.

[49] K. Vidor, V. Fleming, G. Cukor, R. Thorpe, N. Taurog, and M. LeRoy. The wizard of oz, 1939. https://www.youtube.com/watch?v=cMhrpapLTZM.

[50] P. Vosshall. Aws re:invent 2018: How aws minimizes the blast radius of failures. https://www. youtube.com/watch?v=swQbA4zub20, 2018.

[51] Wikipedia. Consistent hashing, 2021. https://en.wikipedia.org/wiki/Consistent_hashing.

[52] Wikipedia. Rip van winkle, 2021. https://www.wikipedia.org/Rip_Van_Winkle.

[53] Wikipedia. Uuid: Universally unique identifier, 2021. https://en.wikipedia.org/wiki/Universally_unique_identifier.

[54] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 29–42, USA, 2008. USENIX Association.

12 APPENDIX A: BUILDING ON A JITTERY FOUNDATION

As our cloud data centers get increasingly large, complex, and fast, we see many reports of systems and networks that just go slow in unpredictable ways. It's not surprising that this impacts the latency seen by servers trying to work with them. This section sketches some pervasive challenges that impact the latency seen within the datacenter and across AZs. We conclude by describing our simplified approach to the problem where we design assuming expect missing responses within a decoupled transactions database.

First, we consider the impact of a growing phenomenon called *gray failure* where networks and servers get sick and do not fail crisply. We look at how TCP's behavior can *amplify jitter* making response latency variable. Cascading timeouts and retries amplify the delays seen by requesting services. Networking across AZs is another source for variations in our response latencies. Then, we see even more onerous challenges with *partial observability* where different opinions can exist about which servers are alive!

Building on these challenges, we frame a possible path to a solution by considering how the notion of a *blast radius*[15, 50] can be applied to the set of *jittery servers* along with *failed servers*.

Examining different classes of failures in distributed systems leads us to realize the old *leader-based* solutions are inadequate given our assumptions. We the frame what we *can know*. Specifically, we know when we *have received a message* and we take action when we've *received enough messages*.

12.1 Gray Failures in Cloud Data Centers

Historically, most distributed systems assumed a **fail-fast** model [11, 12, 26]. Pieces of the system were either healthy or dead. Sick pieces were rapidly converted to dead pieces. Dead pieces were removed from the system and it moved forward without them.

In years past, *fail-stop* was a largely successful strategy when built using high-end hardware within constrained clusters and networks. While certainly not perfect, fail-stop worked well and reported problems were relatively rare. Lately, this has become problematic [30].

Gray failure [22, 27, 34] is a broad term used to describe how pieces of a system may not offer crisp and clear failure semantics. Instead, they may be healthy enough to squeak out an *I'm not dead yet!* [16] message even though they are not contributing their normal share of work. Even a single server going slow rather than failing can lead to devastating outages of the larger system [22, 27].

12.2 Networking and Retries Can Amplify Jitter

In 1974, TCP/IP was designed to maximize throughput over limited bandwidth, an excellent design tradeoff for that time. TCP/IP can cause dramatic increases in latency variability as seen by the communicating parties due to *error correction* based on timeout and retry, *datacenter routing* that sends packets through the same path through switches, and *congestion control* by slowing down TCP to avoid swamping the network. In addition, *cascading timeouts and retries* can wreak havoc on many distributed systems solutions [22].

12.3 Networking across Availability Zones

Cloud data centers are built on sharing. In addition to the challenges seen at a single site of modern cloud computing environments, there are additional challenges posed as we spread our tightly coupled solutions across different sites.

Availability Zones [15, 50] are an emerging approach to offering improved availability of tightly coupled work. By placing portions of a system in different buildings within a metropolitan area, *availability zones* provide sub-millisecond round-trip network communication while isolating parts of the system from many correlated failures. This can, however, introduce additional challenges.

Networking across availability zones can cause correlated delays in latency as well as availability. Availability zones are designed to be as independent as possible to maximize the chance that failures *within an availability zone* do not, in turn, become *failures across availability zones*. Cross-AZ network connectivity use different networking mechanisms with both a higher latency and reduced bandwidth. Not only is latency higher across AZs[43], the bandwidth dedicated to these links is lower than the bisectional bandwidth seen within an AZ.

Normally, cross-AZ traffic has sufficient capacity. Sometimes, congestion leads to packet drops on the cross-AZ links. Stress due to unusually high load can cause cross-AZ traffic to see high packet loss. As we've seen, packet loss leads to increased latency seen communicating over TCP[44].

12.4 Differential Observability in the Network

Differential observability in the network happens when there are inconsistent opinions about which servers can communicate within the network. Let the symbol "↔" denote two servers that see each other and can successfully have a TCP connection.

Consider three servers, S1 in AZ1, S2 in AZ2, and S3 in AZ3. Sometimes, we see the following pattern: (especially when traffic between AZ3 and AZ1 is high):

- **S1 ↔ S2:** Both S1 and S2 see each other.
- **S2 ↔ S3:** Both S2 and S3 see each other.
- **S3 NOT ↔ S1:** (S3 cannot see S1) and (S1 cannot see S3).

This can last for minutes or longer and leading to noticeable increases in response latency[34]. Difference of opinion about who's alive can be *very problematic* for performance[45].

12.5 Correlated Blast Radius: Failure AND Jitter

Most of us have seen wonderful discussions of *blast radius* and how it can be used to reason about correlated failures and improved availability. Similar to outages, *correlated jitter* has a blast radius[15, 50]. Jitter is frequently correlated to AZs, network topology, software upgrade and configuration topology, and sometimes power distribution topology[15, 50].

[43] It is common to see RPCs within an AZ in under 100μs while RPCs across AZs frequently take more than 400μs.

[44] I think of networking in a 3AZ environment as being similar to 3 cities separated by mountains and connected by a 2 lane highways. Cross town traffic works pretty well with many streets. Cross mountain traffic jams up easily under heavy load.

Travel from AZ1 to AZ3 may see dramatic slowdown when busy. AZ1 to AZ3 traffic may be clogged while AZ2 to AZ3 traffic and AZ1 to AZ2 traffic may be fast.

[45] Also, differential observability can cause correctness problems [22, 34].

Are we considering networks or servers? As a server waits for a response, it can't tell if the other server or the network in between caused the slowness. They both offer unpredictable latencies as they process the message. Servers may, indeed, be the underlying culprit. Or, it may be the network. It really doesn't matter!

When responses from healthy servers across healthy networks can take varying time to arrive [39], it is challenging to count on the meaning of a missing message.

Jitter and failure appear the same: **The response isn't here!**
Complete the job even if F nodes haven't answered.
It doesn't matter WHY they haven't answered: ***dead or sick****.*

Hence, we place our nodes in a quorums[46] based on *blast radius* concerns. We need to tolerate F nodes in total even when some are being restarted due to software upgrades at the same time as others are simply experiencing slowness.

12.6 Taking a Time-Out on "Time-Out"

We typically speak of *time-out* as a means to decide to do something different when a message response is delayed. *Timing faults*[19] happen when responses are later than expected. *Omission faults* happen when a late message never comes. Cloud environments offer *probabilistic SLOs*[47].

We don't really know the difference between:

- **Message is late** (*a timing fault*) and may arrive soon, and
- **Message isn't coming** (*omission fault*): better do something.

Determining the difference between these inherently involves trading-off a prompt decision against the chance of an incorrect decision. When messages were almost always timely, it wasn't so bad, just wait a few times the expected message delay and then give up. Now that messages behave like the US Postal Service, it's a bigger quandary. You either wait a *looong time* or risk frequent time-outs and their associated disruption.

As described above in §12.1, the *fail-fast* design pattern has a single centralized *leader server*. As long as the leader is responsive, this works great! When the leader is not responsive enough, there are two big challenges:

- **Deciding the leader is dead:** Just knowing when to give up takes much longer in an asynchronous system where messages delay unpredictable amounts.
- **Deciding *who* should be the next leader:** Selecting the next leader is a form of consensus. *Consensus* is the process of picking a single value across a set of distributed servers[36]. *Paxos* is certainly the most famous example of a consensus algorithm. We know that consensus cannot be guaranteed to complete in bounded time [23].

Centralized control is becoming less and less responsive. As our networking becomes less and less predictable, counting on *fail-fast* to have a single lively centralized decision maker becomes more and more challenging [30]!

Decoupled transactions → No central control!
All aspects of managing the database and its snapshot isolation are decentralized and quorum based.

Assume no more than F servers *appear to be sick or dead to this server*. It doesn't matter if they actually **are** sick or dead, just that the requesting servers perceives them as non-responsive. The remaining non-jittery servers are acting as if they are synchronous and meet their response time expectations.

All but F servers are responding promptly from the requesting server's perspective. This is a commonly observed pattern.

In §7.2, we map *"no more than F sick servers"* to the various types of servers used within our decoupled transactions database.

By considering the behavior we need from these server types, their placement in a 3AZ deployment of cloud data centers, and their implementation, we sketch how these servers and the services they provide can be robust even in the face of an AZ+1 outage.

12.7 "Omitting Omissions" as We Make Decisions

One interesting aspect of *fail-fast* is that is exists to convert *timing faults* (due to late message arrival) into *omission faults* (where we decide the message is never coming).

These *omission faults* are then used to decide we must be seeing a *crash fault* where no more messages will ever be coming.

Every decision our *decoupled transactions* database makes is based on an actual set of messages received.
We may *wonder* if a server is sick if messages are late. We don't *decide* to replace a server because messages are late.
Messages must have *Happened Before* [35] our decisions. Fortunately, we can decide with a *quorum of messages* each of which *happened before*.

Applying *happened before* to a quorum of responses does, however, have a number of challenges and subtleties!

[46] See §7.2 for a discussion of quorums and failures as applied to each of our DB's server types. This focuses on the requirements of each server type and how to tolerate an AZ+1 failure without disrupting the decoupled transactions database.

[47] See §1.1.

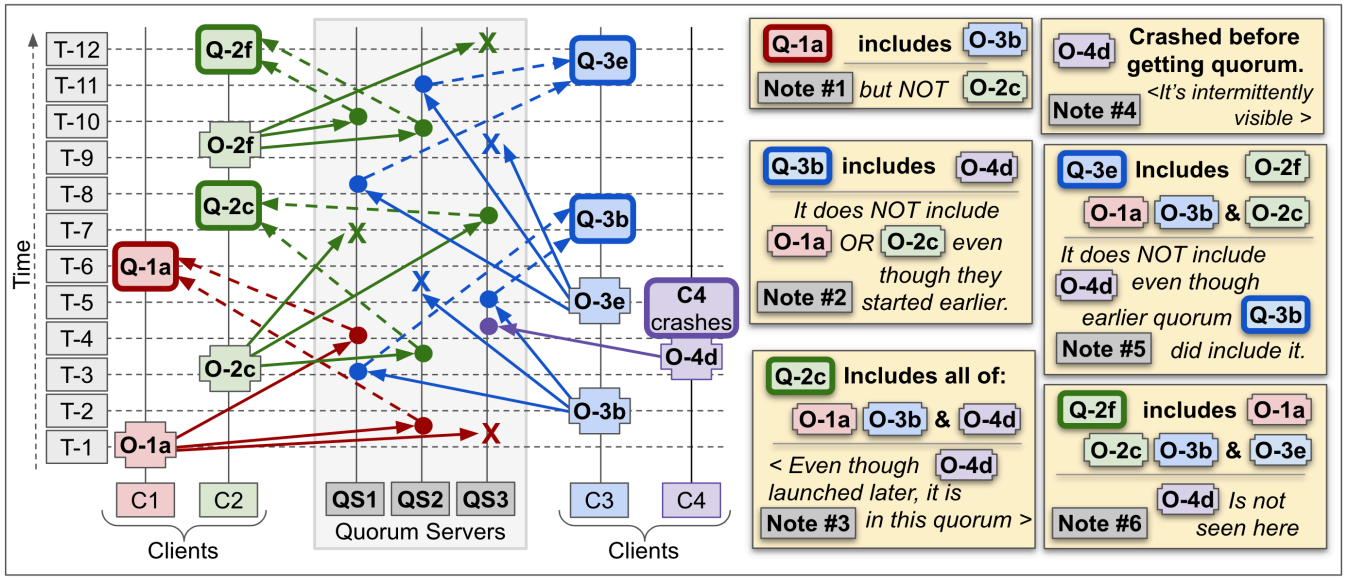

Figure 4: Lots of races can happen when multiple clients each try to do their operation using quorums.

13 APPENDIX B: QUORUM'S SUBTLE CHALLENGES

Quorum operations necessarily involve races and surprising orders of completion across the multiple quorum-servers in the subsystem. That is, after all, the point behind using quorum. Operations may execute in various orders and their quorum avoids stalling.

This section is placed in an appendix for interested readers.

It takes some work to study the cause of these races
Understanding this in detail is not essential to understanding the paper and its conclusions.

13.1 Quorum and Its Subtle Challenges

Quorum operations necessarily involve races and surprising orders of completion across the multiple quorum-servers in the subsystem. That is, after all, the point behind using quorum. Operations may execute in various orders and their quorum avoids stalling.

Consider Figure 4 for this discussion.

Define the following:

- $\mathbf{O_x}$ to mean **Operation** $\mathbf{O_x}$
- $\mathbf{CQ_y}$ to mean **Complete Quorum** $\mathbf{CQ_y}$
- $\mathbf{CQ_y \geq O_x}$ to mean $\mathbf{CQ_y}$ *includes* $\mathbf{O_x}$
- $\mathbf{CQ_y \not\geq O_x}$ to mean $\mathbf{CQ_y}$ *may possibly not include* $\mathbf{O_x}$

We see the following *weird behavior* with quorum:

- **Operations may be included even if launched later.**
  By the time a combined result from a complete-quorum CQ_k is returned from the quorum-client, it may include operations that started later than when O_k started.
  - $CQ_{1a} \geq O_{3b}$. See Note 1.
  - $CQ_{2c} \geq O_{3b}$ and $CQ_{2c} \geq O_{4d}$. See Note 2.
- **Operations may be NOT included**, *even if they launched before the complete-quorum that does not include them.*
  Sometimes, a *complete-quorum* CQ_k visits other nodes as it gathers up its work and misses some ongoing operation O_j that started first.
  - $CQ_{3b} \not\geq O_{1a}$ and $CQ_{3b} \not\geq O_{2c}$. See Note 2.
- **Incomplete-quorum operations may be included.**
  This includes *temporarily incomplete-quorum operations* that are simply ongoing work. It may also include *permanently incomplete-quorum operations* from crashed clients.
  - $CQ_{1a} \geq O_{3b}$ (*temporarily incomplete* - Note 1)
  - $CQ_{3b} \geq O_{4d}$ (*permanently incomplete* - Note 2)
  - $CQ_{1a} \geq O_{3b}$ (*temporarily incomplete* - Note 3)
- **Operations may be included intermittently**.
  This is a common occurrence as two different quorum operations visit different servers. The same client may *initially see operation* O_j and later it is gone. See Note 5.
  - $CQ_{3e} \not\geq O_{4d}$ *even though* $(CQ_{3b} \geq O_{4d})$ *completed earlier.*
- **Two quorums may include each other's operations!**
  - $CQ_{3e} \geq O_{2f}$ See Note 5.
  - $CQ_{2f} \geq O_{3e}$ See Note 6.

  How surprising is that!

14 APPENDIX C: JITTER-FREE DATABASE

Here, we dive deeply into the details of the pieces needed for a *decoupled transactions database*. The goal is to motivate how each piece can be implemented without jitter.

For many readers, this may be too much detail, so it is in an appendix. Hopefully, this can sate the curiosity and/or skepticism of those wondering about how to use seniority, quorum, and confluence to make a robust jitter-free database.

We cover the following:

- **§14.1: (Jitter-Free TX Commit, Begin, & Retire)** How can a high-performance jitter-free transaction system run without centralized transaction management? We look at commit, begin (getting a snapshot and whereabouts), and retire transactions. This extends the discussion in §4.
- **§14.2: (Jitter-Free Flushing to the LSM)** How does *flush* work? Why is it OK to have a worker-server stall with its recent updates not yet flushed? How does the overall database avoid getting stuck for very long?
- **§14.3: (Jitter-Free Merging of LSM Data Files)** Our database sees all older changes using the LSM in shared storage. We need to curate this LSM, performing merges as needed, register these changes with our *catalog* all without jittering and effecting ongoing transactions?
- **§14.4: (Jitter-Free Reading of LSM Record-Versions)** The whole goal of keeping most of our data in read-only shared storage is to allow shared and jitter-free access. How does this work without jitter? How can we look into the LSM by key (or key-range) and find the data we need?
- **§14.5: (Jitter-Free Reading Recent Records)** Accessing very recently committed updates is a challenge in this system. Mostly, the worker committing these changes can respond with the new value for the record. This is, of course, a risk of jitter and not always a successful approach.

14.1 Jitter-Free TX Commit, Begin, & Retire

Transaction commit is performed by a quorum of *coordinator quorum-servers*. Prior to transaction commit, worker servers send a *permission-to-commit* operation to N_{Coord} quorum-servers[48].

When Q_{Coord} of N_{Coord} have responded, the *coordinator-quorum-client* in the worker-server *combines the responses*. This may (or may not) grant permission to the transaction. If so, it then writes a *transaction commit record* to the worker's log[49].

The coordinators' commit[50] of a transaction T_1 does 3 things:

- **Assignment of seniority to T_1:** Seniority is *guessed by the worker* and *confirmed by a coordinator quorum.*
- **Conflict checking prior to commit of T_1:**[51] Updates by T_1 are checked for conflicts against any transaction T_2 committed since T_1's snapshot time.
- **Whereabouts creation:** Each quorum-server remembers the updates done by the transactions it has permitted.

Transaction commit does not jitter

Transaction begin also uses *coordinator quorum-servers*. Worker servers send a *Get-snapshot-and-whereabouts* operation to N_{Coord} of the coordinator's quorum-servers.

Each *Get-snapshot-and-whereabouts* operation specifies:

- **Proposed snapshot-time:** A future snapshot-time is *proposed by the worker* and *confirmed by Q_{Coord}*
- **Last snapshot:** seen by this worker server
- **Last whereabouts:** seen by this worker server. Only newly committed whereabouts need to be returned to the worker.

When of Q_{Coord} of of N_{Coord} have responded, the *coordinator-quorum-client* in the worker-server *combines the responses* to get a *snapshot time* [52] and a *set of whereabouts* [53].

Transaction begin does not jitter

Transactions retire when their seniority shows they are:

- **Older than the oldest snapshot** and
- **Flushed to the LSM**

Seniority of the oldest snapshot and *seniority flushed to LSM* are sent to the coordinator using *complete-quorum operations*. At least Q_{Coord} quorum-servers see the change.

As coordinator-clients send messages to coordinator-servers, they send these seniority values and the servers send them back. Seniority retirement values only move forward. Both clients and servers see them advance monotonically. Retired transactions are:

- *Scrubbed from each server as retirement seniority advances*
- *Scrubbed from combined results at the client*

Transaction retirement does not jitter

[48] See §4: (Logical Time & Transaction Commit) for more details.

[49] The presence of the commit record in the worker's log defines the moment of commit.

[50] The coordinator subsystem's quorum-servers perform commit without jitter.

[51] For any transaction T_2 with an identical seniority to T_1, if T_1 and T_2 have conflicting updates, either one or both of them must abort.

[52] Assuming the proposed *snapshot-time* arrives early enough at Q_{Coord} servers.

[53] Each quorum-server sends the *whereabouts* for the transactions it has permitted. Any committing transaction was permitted by Q_{Coord} servers. Grabbing *whereabouts* from Q_{Coord} servers means the *combined set of whereabouts* will include every update made by any transaction that might have committed.

14.2 Jitter-Free Flushing to the LSM

Seniority of flushed files: Periodically, each worker will *flush committed transactions to the LSM.* This is a long-running activity involving multiple This is accomplished in a number of phases:

- **Select seniority range of transactions to flush:** Each flush contains all committed transactions in a range.
  - *Lower bound:* Start at the previous flush's upper bound.
  - *Upper bound:* Pick a committed transaction.
- **Flush all transactions in the range:** Only ones committed at this worker. Write them to a data file in shared storage. Flushed data files are not yet visible to readers of the LSM.
- **Register the flush with the catalog:** Describes its worker and seniority range using a quorum operation.

Workers are allowed to jitter

It's OK if a worker slows down while flushing:

- **Recent changes are not yet visible in the LSM:**
  - Recent changes from all workers not seen in LSM
  - System slows a slight bit
- **Other workers get impatient:**
  - They remove slow worker
  - Rebuild its recent changes from its log
  - Flush them to LSM

Worker servers may stall → They're rapidly removed
Other workers extract the sick worker's changes
from the log and flush them to the LSM

Ongoing transactions (on other workers) see little impact

14.3 Jitter-Free Merging of LSM Data Files

Merge of LSM data files is periodically performed by any worker in the database. Workers *consult the catalog*[54] to see the part of the LSM that most needs reorganization. Workers wishing to help reorganize the LSM are given assignments to merge by:

- **Read a set of data files from the shared storage**
- **Extract records from these data files for input:**
  - *Records must lie within a specified key-range*
  - *Records must lie within a specified seniority-range*
- **Write a new data file (up to a maximum size)**
- **Register the new LSM data file with the catalog**
  - *Registered files have sorted records*
  - *Registered files specify their key-range and seniority-range*

The catalog subsystem comprises code running within *catalog quorum-servers* as well as code running within *catalog clients within each worker-server.* The state of the catalog subsystem is kept within a quorum of the catalog quorum-servers.

Merge is idempotent: *It may be retried until it succeeds*
Catalog handling of merge is jitter-free

Seniority of new data files supports retirement
New data files may have seniority within each key-range
Older data files aren't needed → Catalog may retire them

[54] By sending performing operations at Q_{Cat} of N_{Cat} catalog quorum-servers.

14.4 Jitter-Free Reading of LSM Record-Versions

Access to older record-version is done by *exact-key* or *key-range.* Locating them in shared storage comprises five steps:

(1) **Find the set of data files to read:** *Consult the catalog using Q_{Cat} of N_{Cat} responses*
(2) **Access the data files in shared storage:** *Data files are stored using consistent hashing → can be located and read across replicas without centralized control*
(3) **Locate the record-versions in the data files:** *Structured for key based lookup*
(4) **Merge across LSM levels**
(5) **Return the latest record-versions at snapshot time:** *The reading worker filters for snapshot semantics*

LSM data may be read without jitter
Each step avoids getting stuck waiting for any slow server(s)

The reading worker may get sick but
its dependencies (e.g., catalog and data files) avoid jitter

14.5 Jitter-Free Reading Recent Records

Whereabouts describe which worker-servers *may have recently updated records.* These are acquired by each transaction's worker at the beginning of the transaction. See §14.1.

Normally, recent record-versions are read from the updating worker-server by sending a direct message to the worker and waiting for it to return the requested update.

A direct message to read from another worker will request:

- **Read of exact-key or key-range**: The whereabouts describes the update that was granted permission to commit.
- **Read at an exact seniority:** The whereabouts specify the precise seniority of the possible update
- **The updated record-version may have not committed:** There is a chance the update did not commit. If so, the worker will respond no update for that key exists with that seniority.

Reading recently committed record-versions is the single biggest jitter-risk in this design

Fortunately, we only try to read record-versions from another worker when a whereabouts shows it is very likely to be there

Normally, recent changes will flush to the LSM
in shared storage within tens of seconds
Once there, they can be read without jitter

Also of use is the *Log-following server* (See §15.13)
Other than the very last few seconds since an update,
reading from it can be a rapid alternative.

Reading recent changes without jitter depends on rapid removal of a sick worker
This includes repairing and reading the sick worker's log

Sick workers can be removed in less than a second!
The hope is that the problem is rare and the repair is rapid

See §15.12: (Bounding the Pain When a Worker Is Sick)

15 APPENDIX D: JITTER-FREE LOG-REPAIR

In appendix D, we offer a deep dive of many aspects of log-repair and jitter free repair of data from a worker's log. We offer a guesstimate for the wall clock time to remove a worker and access its data. Also, we present a discussion of the value of having *log-following* servers keeping a close follow of each worker.

- **§15.1: (Jitter-Free Log Seniority (from Logger))**
  Later, in §15.9, we address *concurrent log repair by multiple worker-servers*. We must support concurrent log repair and avoid picking exactly one! By assigning order (or *seniority*) of the log pieces before failure, concurrent repairing workers can function independently.[55]
- **§15.2: (Jitter-Free Log Quorum)**
  How do we know the status of appended fragments in the log? How many replicas are needed for fencing, log-repair, and detecting a server is sick?
- **§15.3: (Jitter-Free Liveness Check via Log)**
  Here, we present an important observation. *Quorum logs don't jitter!* If a log can be read without jitter, we can detect a sick worker by examining their progress in the log. *This is an essential aspect to ensuring we have a jitter-free database!*
- **§15.4: (Jitter-Free Concurrent Log Fencing)**
  This section covers the details of *fencing the log of a sick worker*. This involves both steps in the *catalog* as well as steps in each of the *log server replicas* holding extent data that needs to be fenced. *Fencing the log* always precedes *repairing the log* so the fencing server gathers information needed for repair as it fences.
- **§15.5: (Log-Repair: Goals and Challenges)**
  Here, we describe the essential goals and challenges to providing jitter-free log repair.
- **§15.6: (Jitter-free Log Repair & Shared Storage)**
  This is an overview of many of the challenges to come in log-repair.
- **§15.7: (What Do We Know Before Log Repair?)**
  Before we can *repair a log*, we need to define our assumptions about what's available to us before launching the repair.
- **§15.8: (What Does It Mean to Be Ambiguous?)**
  After a crash, some of the most recently written log fragments may or may not survive repair. How does this happen? What does it mean for repair? Why will this result in a valid log state after repair?
- **§15.9: (Concurrent Repair of the Log)**
  How does log repair itself work? What steps are involved? How can it reliably complete even when there may be multiple competing worker-servers actively repairing the log?

  Multiple log repairers do have some challenges! Each may see a different set of log-server replicas. Some may see the presence of a fragment in the replicas they can reach. Others believe it doesn't exist in the log. When this happens, either outcome is OK but we *must pick and remember the decision about the presence or absence of a fragment. Picking exactly one decision warrants an entire appendix!*
- **§15.10: (Picking a Single Outcome Per-Fragment)**
  For ambiguous fragments, we must select an outcome to survive after log-repair. How can this be done without conflict, without jitter, and in bounded time?
- **§15.11: (Bulk Fragment Repair)**
  It turns out that 100s of ambiguous fragments can be repaired in just a few quorum operations. This can mean sub-second repair of the log!
- **§15.12: (Bounding the Pain When a Worker Is Sick)**
  Finally, we consider just how slow it would be to *decide a worker is sick*, *fence its log*, and *repair its log* to allow reading the state of committed record-versions. *We see that our use of quorum throughout the design can result in a surprisingly fast access to committed data through a worker's repaired log!*
- **§15.13: (Jitter-Free Retry to Log-Followers)**
  Log-following servers optimize reading of recent changes. While not seminal to avoiding jitter for recent reads, it is a practical and useful feature.

15.1 Jitter-Free Log Seniority (from Logger)

Seniority of items within each worker-server's log: Log records are created per worker containing records for *new record-versions*[56] and *transaction commit records*. Commit records have their transaction's seniority and appear in the log in commit (or seniority) order.

Items composing the log have ranges of seniority:

- **Fragments:** These have a range of seniority based on commit records contained in each fragment.
  - **Monotonically increasing:** Each fragment's range is larger than the previous one. Seniority of fragments has a low-order sequence field to increment if needed.
  - **Lower bound:** Greater than the last fragment's range.
  - **Upper bound:** Largest seniority in the fragment.
- **Extents:** Seniority range bounds based on its fragments.
- **Log-Windows:** Seniority range bounds based on its non-retired extents.

For both open extents and open log-windows (i.e., still receiving fragments), the upper bound of their seniority is not defined.

Seniority of log items is assigned by the logging worker. Log repair can be performed by multiple other worker servers as they follow the seniority established by the crashed logging worker.

[55] Picking a single log repairing worker-server would require a central authority. This cannot be guaranteed to complete in bounded time. So, we don't pick just one!

[56] Like most transaction systems, we these are labeled with a temporary identifier and later bound to a committing *seniority* (i.e., transaction-id) by a commit record.

15.2 Jitter-Free Log Quorum

Let's examining the replication requirements for logs. Log servers are expensive, each receiving many 100s of megabytes per second and ensuring low-latency durable appending to the log.

Log replication requirements:

- **Must read the log with F jittery servers:** Reading the contents of a log tells us the state of the database.
- **Don't need to write the log with F jittery servers:** We can send new log writes to a new extent on different servers.

Let's use the following terms:

- **N_{Log}:** Count of log-server replicas per extent
- **Q_{Log}:** Count of log-servers acknowledging quorum.
- **NQ_{Log}:** (NQ → Not Quorum)
  How many missing means we don't have quorum?

$$NQ_{Log} = (N_{Log} + 1 - Q_{Log})$$

NQ_{Log} tells us the following:

- **When is a fragment is *guaranteed to be durable?***
  If NQ_{Log} replicas don't have it, it's not durable
- **Is a worker is sick?**
  If NQ_{Log} replicas haven't seen recent work, the logger's sick
- **How many replicas to fence?**
  If NQ_{Log} replicas are fenced, the extent is fenced
- **How many replicas to read a durable fragments?**
  Read NQ_{Log} replicas → see durable fragments
- **How many replicas to read for accurate log-repair?**
  Read NQ_{Log} replicas → we can repair the log

We place log-servers to tolerate jitter
NQ_{Log} must be non-jittery

This allows **non-jittery reads** of durable log-writes

It does not ensure non-jittery writes to that log-extent.

See §7.2 for a discussion of this in a 3AZ environment.

15.3 Jitter-Free Liveness Check via Log

FLP (Fischer-Lynch-Paterson)[23] is rightly considered to be one of the most significant papers in distributed systems literature. It has dramatically shaped our solutions. FLP states that it is impossible to *deterministically* reach consensus in bounded time across a set of servers *with only a single faulty server.*

FLP assumes communication with a server is only via *asynchronous messages.* Today, asynchronous messages underly our cloud data center solutions[57]. Our assumptions about the probability of correlated jitter offer a way to *side-step the challenges posed by FLP.*

We are not challenging the correctness of FLP
We plan to side-step its assumptions

Consider database servers that log before responding. Our worker-servers are such servers. Before answering a message, they first log. If they can't log, they don't answer.

Assume log writes are sent to a quorum of servers. Similar to AWS Aurora[46], each server must commit a log write to 4-of-6 log replicas before responding to the message.

Assume that at most F of the servers jitter at once. These F servers are deployed so that all correlated problems impact fewer than F servers at a time[58].

You can check sickness without getting direct responses

The log knows the truth!

If (N_{Log} + 1 - Q_{Log}) log replicas respond to a message:
"that worker-server hasn't logged here for a while"
then you know the worker hasn't logged to Q_{Log} of N_{Log}
If it hasn't logged, it hasn't done any work and must be sick.

You know the server is sick
based on happened-before answers to messages

Not only can you know it is sick, you can:

- **Fence its log:** Kill it by stopping future work
- **Recover the log:** Ensuring accurate contents in the future
- **Read what is in the log:** Allowing reads of recent changes

Multiple workers may fence and repair a log. *We can't pick exactly one log repair server.* That would take a central authority. Instead, we must ensure that *when one or more log-repairing workers finish, the log will always have the same repaired contents.*

15.4 Jitter-Free Concurrent Log Fencing

A single log can be fenced by multiple workers concurrently:

- **Fence log-window in catalog:** Disallow new extents
- **Fence each extent (latest to earliest):** For each extent:
  - *Get catalog extent info:* Which were sealed? Where are each extent's replicas?
  - *Fence extent replicas:* For each replica:
    - Disable adding new fragments
    - Learn the largest fragment number in replica
  - *Fence enough replicas:*
    Fence at least: ((N_{Log} - Q_{Log}) + 1) replicas to prevent logging new fragments to Q_{Log} in the future
  - *Fence extent in catalog:* Record progress...
- **Record log-window is completely fenced in catalog**

Fencing a log-window does not jitter

Fencing may be done by one or more worker-servers
It is complete when any of them finishes fencing

[57] Indeed, message delivery time is getting less predictable as we improve other characteristics of the data center including higher bandwidth and lower latency. See §12.

[58] This is precisely the same arguments used to define *blast radius* boundaries[15, 50].

15.5 Log-Repair: Goals and Challenges

Let's review what we need to do when repairing the log. Log repair's job is to take a recently fenced log-window and ensure it can be read accruately. It is more challenging for this system than many databases because we can't have anything jitter. That means we can't have any centralized knowledge! That means we can't have a single log-repairing server.

Multiple log-repair servers adds some complications!

Accurate and repeatable log contents: The basic goal of log-repair is to pick any correct version of the log and before using it ensure we can make a durable and repeatable contents for that log. When logging to a replicated store, this does not happen by itself.

- **Committed work stays committed:** If Q_{Log} replicas have the fragment, it must remain in the log
- **No holes in the log:** The first fragment we omit from the log is the *tail of the log*. No later fragments may exist.
- **Ambiguous fragments may go either way:** It's OK to include them or exclude them. Once log-repair is complete, they must always be either in or out of the log.
- **Uncommitted work stays uncommitted:** Once the log is repaired, if it's not in the log it will never be in the log.

Possibly many concurrent log-repairers: We must support one or more log-repairers working on repair. There are subtle issues to clarify to ensure this is correct.

- **It just takes one to finish:** Once one or more completes log repair, it is complete
- **Each log-repairer may see different log-replicas:** They may be in different parts of the data-center. As they access log-storage replicas, log-repairing worker W_a may see different replicas of an extent than W_b. Furthermore:
  - *Fragments may differ:* As they were written, different replicas may see fragment F_x
  - *Durability guarantees may differ:* Some fragments made it to Q_{Log} replicas and others to fewer.
- **Log-repairers may see different states of a fragment:** This can happen when they see different replicas:
  - *Guaranteed durable:* Known to be at Q_{Log} log-server replicas. Log-repair reads NQ_{Log} replicas for each fragment.
  - *Ambiguous:* If fewer than Q_{Log} replicas exist, we may see different opinions. One repairing server may see the fragment present and another may see it missing.

We cannot jitter as we support multiple log-repairing servers! It is essential to recover the log in a small and fixed number of steps. If two log-repairing servers have different opinions of a fragment's disposition, we must resolve this without jitter. *We cannot have liveness problems as we repair the log!*

15.6 Jitter-free Log Repair & Shared Storage

Log repair has two big jobs: *pick the end of the log* and *ensure the durable end of the log doesn't grow or shrink.*

Fence before log-repair. Before we repair the log, we ensure it is frozen and cannot have new fragments added. See §15.4.

We can't add to log-servers for fenced log extents. When recovering a log, we don't ever add anything to the log-servers themselves. Before recovering the log, we fenced it. Fencing log extents in the log-servers means that nothing can be added to the extents! Hence, we toss more information into the catalog to implement log-repair.

Log repair adds facts to the catalog. After fencing, we don't modify log extents in the log-storage servers. We simply add information to the catalog! This includes some of the log fragments (the log records themselves)!

Picking the end-of-log is ambiguous. Acknowledged fragments are replicated to at least N_{Log}-F replicas and will be recovered. Depending on *which N_{Log}-F replicas are seen by log repair*, the end-of-log log may vary, including different sets of non-acknowledged fragments. The final length of the recovered log must be durably remembered before any recovered fragments are used to ensure the length doesn't shrink or grow if log repair is restarted.

Using one log repair server avoids end-of-log ambiguity.

Selecting exactly one server cannot be jitter-free.
Picking one server requires consensus
It can't be guaranteed to complete in bounded time[23]

Decoupled transactions recovers logs with >= 1 servers.
A quorum of catalog servers resolves end-of-log ambiguity and durably remembers the state of the recovered log.

15.7 What Do We Know Before Log Repair?

Logging workers write fragments to N_{Log} log-storage-replicas. They are *known to be durable* after Q_{Log} confirmations.

Before log repair, each fragments is possibly:

(1) *Known durable:* Q_{Log} log replicas have confirmed responses
(2) *Ambiguous:* Possibly less than Q_{Log} fragment replicas exist
(3) *Missing:* No attempt was made to log the fragment

Last durable fragment (LDF): Fragments refer to the latest known durable fragment. *This provides log-repair a starting point.*

Before log-repair, the catalog for the log-window knows:

- **Log-window information:**
- **Per-extent information:**
- **Per-fragment information:** Empty before log-repair.

***Log repair*:** *Resolve the ambiguity of ambiguous fragments*

We must determine the last fragment in the log
All fragments in the log must be readable in the future

15.8 What Does It Mean to Be Ambiguous?

Each log-repairing server reads NQ_{Log} replicas[59]. See . (NQ stands for "Not Quorum") As described in

$$\mathbf{NQ_{Log} = (N_{Log} + 1 - Q_{Log})}$$

NQ_{Log} is the minimum number of replicas we must read.

Example: *AZ+1 jitter tolerance in 3AZ configuration.* See §7.2.

We would then see:

- **N = 6:** 6 total log-storage replicas.
- **Q = 4:** Only when we know 4 log-storage replicas are durable, is the write complete.
- **NQ = 3:** If we read 3 of these replicas, *we will see at least one of the 4 replicas written.*

[59] See §15.2

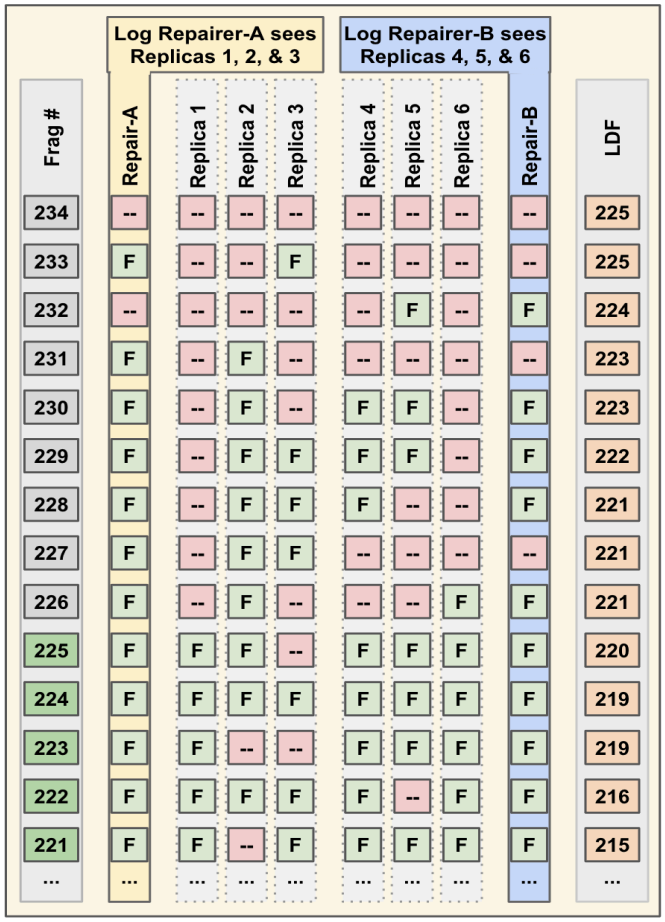

Figure 5: Monotonic state transitions to decide end of log

Note the following (using 3AZ numbers) (See figure 5.):

- **Fragments up to 225 are known durable:** There are at least 4 durable replicas.
- **LDF is per-fragment:** Each fragment includes LDF. This is the latest fragment that we know is durable on 4 replicas.
- **Log-Repairer-A sees replicas 1, 2, & 3:** It will certainly see at least 1 of 4 durable fragments.
  - *Fragments up to 231 are seen by Repairer-A:* If A is the only repairer, the log will end after 231.
  - *Fragment 233 is visible but not 232:* Unless another repairer sees 232, 233 will be discarded.
  - *Repairer-A sees LDF of 225:* Fragment 233 has LDF= 225
- **Log-Repairer-B sees replicas 4, 5, & 6:** It will certainly see at least 1 of 4 durable fragments.
  - *Fragments up to 226 are seen by Repairer-B:* If B is the only repairer, the log will end after 226.
  - *It also sees fragments 228, 229, and 230*
  - *It does not see fragments 227, 231, or 233 onward.*
  - *Repairer-B sees LDF of 224:* Fragment 232 has LDF= 224

15.9 Concurrent Repair of the Log

Databases normally assign a *single log repair server*. It is an idempotent operation and may crash and restart. As long as the single log repair server finishes, the log is repaired for all time.

> ***We can't pick a single log-repair service!***
> That requires a strongly consistent central authority!
> *That might jitter!*
>
> **We must allow *at least one* log repair to fix-up the log!**
> *It turns out that is quite challenging to do without jitter!*

Fragment-by-fragment repair: Starting at the *latest fragment in the log-window* found while fencing the log. In that fragment is an LDF, a known durable fragment on at least Q_{Log} replicas. From there, move forward through the log making fragments durable[60].

The final step is to update the catalog selecting an end-of-log. This should include all repaired *ambiguous fragments* up until there's a hole (or missing fragment) confirmed in the catalog.

> **Repairers read (N_{Log} + 1 - Q_{Log}) replicas** (See §7.2)
> *Each may see less than a quorum Q_{Log} replicas*
>
> **Big challenge:** Log-repairers may see different replicas
> ***Some see the fragment... Some see a hole!***
>
> **Either keeping the fragment or ending the log is OK**
> → Different log-repair views → an *ambiguous fragment.*
>
> *We must finalize* ***exactly one decision in the catalog***

We must ensure the following guarantees:

- **Preserve known durable fragments:** Once the logger saw a fragment acknowledged by Q_{Log} of N_{Log} replicas, that logging worker believed the fragment was durable and its transaction's committed. The database may have confirmed it is committed.
- **Leave no holes in the log:** A fragment is only durable when all earlier fragments are durable.
- **Finalize the contents of the repaired log:** *Ambiguous fragments (not confirmed by Q_{Log} replicas) may or may not survive:* It's OK to include it and it's OK to exclude it.

[60] Actually, each log-repairer can do parallel repair of fragments. Reading from replicas of the log-extent can fetch multiple fragments at once and sift through the set of fragments at each replica. Using this, a log-repairing worker can perform updates to confirm the presence or absence of each fragment, possibly competing with other log-repairing workers. See §15.11: (Bulk Fragment Repair) for an in-depth discussion of monotonic and concurrent log-repair.

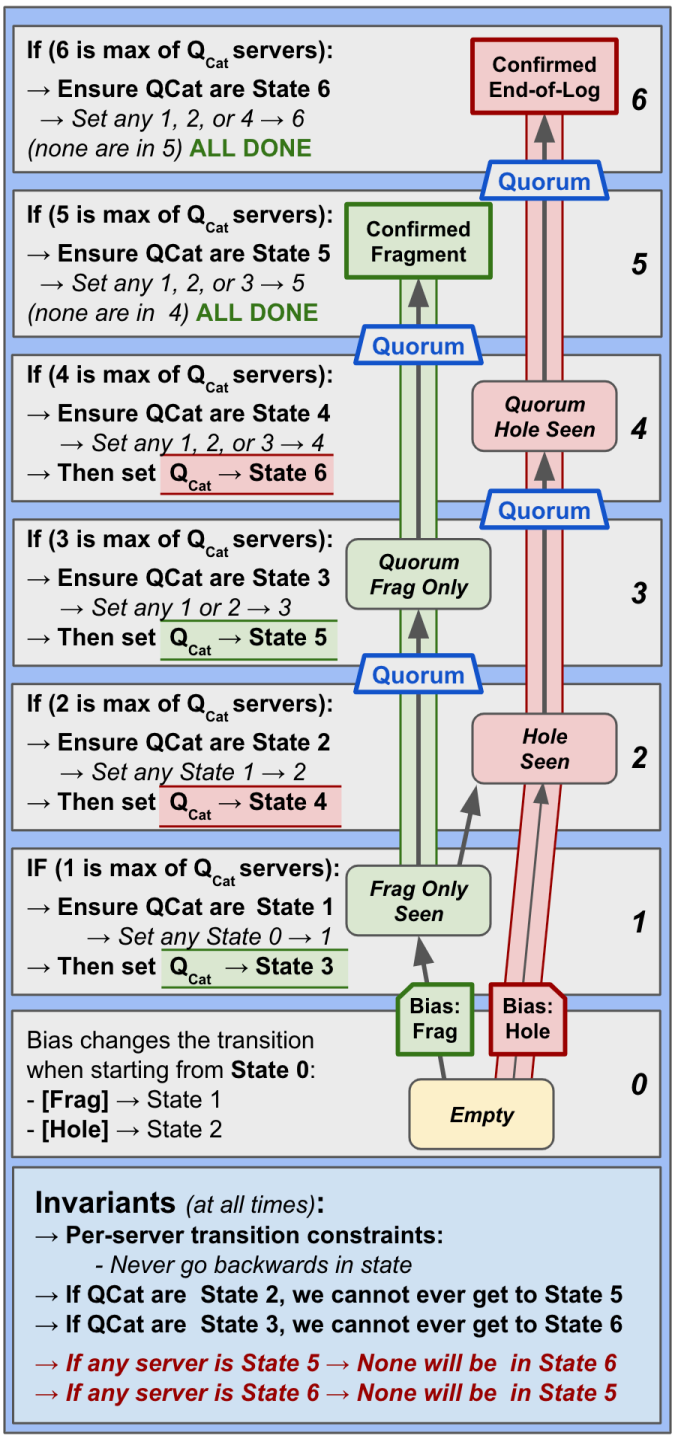

Figure 6: Monotonic state transitions to decide end of log

.

15.10 Picking a Single Outcome Per-Fragment

As repairers read each fragment, they may see:

- **The fragment is in at least one replica:**
  - *Try to include the fragment if there's no disagreement*
  - *Allow a hole if another repairer saw only holes*[61]
- **No replicas of the fragment:**
  - *Try to end the log here (unless another repairer disagrees*
  - *Allow the fragment if there's disagreement*[62]

Consider figure 6. For each fragment, the log-repairer will:

- **Bias for fragment:** The log-repairer either sees the fragment from the log or it doesn't:
  - *Fragment is seen:* Biased to include a fragment
  - *Fragment is missing:* Biased a hole
- **Update fragment in catalog, Step 1:**
  - *For a Fragment Bias: Send a "Frag-Seen" message*
    - Request: map from State-0 (Empty) to State 1 (Frag-Seen)
    - Returns an error if not in State-0
  - *For a Fragment Bias: Send a "Hole-Seen" message*
    - Request: map from State-0 (Empty) to State 2 (Hole-Seen)
    - Returns an error if not in State-0
- **Update fragment in catalog, Steps 2 and beyond:**
  - Read the fragment's state from at least Q_{Cat} replicas.
  - If there are different answers, send the largest state to at least Q_{Cat} replicas.
  - Advance to next state (See figure 6)

Achieving a consensus of the outcome for a single fragment takes no more than a six quorum interactions[63].

15.11 Bulk Fragment Repair

Each log-repairing server fences the log and gets a set of *ambiguous fragments*. The number is a function of the outstanding and unconfirmed log writes.

There are typically 100s (or fewer) ambiguous fragments
Assume:

- 100 mb/sec logging
- 32kb per fragment (average)
- 10 milliseconds to achieve Q_{Log}

Then: Average of 30 fragments not yet acknowledged to the logging server

- 100MB / 32KB is about 3,000 fragments per second
- 10milliseconds is 1/100th of a second
- 1/100th of 3,000 is 30

Typically about 100 or fewer ambiguous fragments.

[61] If this repairer saw a fragment and another saw only holes, it is ambiguous.
[62] If this repairer saw only holes and another saw a fragment, it is ambiguous.
[63] I'm sure this is not optimal. My goal is to establish that it is bounded and small.

A bulk log-repair of the ambiguous fragments:

- **Get ambiguous fragments from log:** The fence operation returns a bulk set of fragments.
- **Combine fragment:** all bulk results from the log replicas
- **Send combined request** to catalog as a bulk step 1
- **Perform bulk calls** to catalog quorum for all fragments
- **Call the catalog to seal** the log-window, ending at the first fragment in Step-5 (End-of-Log)

Log repair is both jitter-free and bounded

It is a fixed amount of work based on the number of ambiguous fragments and the number of concurrent repairers

Each step of log-repair is based on quorum operations
Log-repair is jitter-free

15.12 Bounding the Pain When a Worker Is Sick

Removing a sick worker and seeing their work can actually be quite fast. It involves the following steps:

- **Decide if the worker is sick:**
  - *Catalog:* Ask N_{Cat}, wait for NQ_{Cat}
    → Where is worker logging?
  - *Per active-extents:* Ask N_{Log}, wait for NQ_{Log}
    → Has logging happened lately?
- **Fence the worker's log-window:**
  - *Catalog:* Ask N_{Cat}, wait for NQ_{Cat}: Fence in the catalog:
    - *Fence the log-window*: Disallow fragments being added
    - *Describe active extents:* List of non-sealed extents
  - *Per active-extents:* Ask N_{Log}, wait for NQ_{Log}
    - *Fence the extent*: Restrict further fragments being added
    - *Largest fragment number:* Learn what was added
    - *Read recent fragments:* Fetches lates replica fragments[64]

    Fencing grabs most (or all) fragments needed for repair.
- **Repair each open extent:** We already fetched the recently written set of fragments from each replica.
  - *What's durable?*
    Find last known durable fragment (field in last fragment)
  - *What's ambiguous?*
    Everything after the last known durable fragment
  - *Disambiguate fragments in the catalog:*
    - *Package up ambiguous fragments into a catalog request*
    - *Send bulk fragments to a quorum of catalog servers*
    - *Send request to resolve fragments in catalog* [65]
    - *Define last fragment in extent*
    - *Seal active extent:* Confirm length in catalog
- **Seal log-window in catalog:**
  Describes repaired extents and end-of-log
- **Read the log:** Start at the end-of-log

Fast health-check, fencing, & repairing of worker's log!

Consider the following count of quorum operations:

- **Health-check:**
  - *Catalog:* One quorum request
  - *Per active extent:* One quorum request to at least NQ_{Log} replicas
- **Fence:**
  - *Catalog:* One quorum request
  - *Per active extent:* One quorum request to at least NQ_{Log} replicas
    Returns bulk status of recent fragment, typically all ambiguous fragments
- **Repair:**
  - *Catalog (Per active extent):* One quorum request per active extent to bulk repair ambiguous fragments (may be done in parallel for each active extent). Possibly up to 3 or 4 additional requests to resolve different opinions of ambiguous fragments[a]
  - *Catalog (Finalize log-window repair):* One final quorum request to declare end-of-log

Catalog operations for long-window repair:

- *Per log-window repair:* 3 catalog quorum operations[b]
- *Per repaired active-extent:*
  2 catalog quorum operations

Log operations (per active extent) for log-repair:
2 quorum operations to log replicas

Removing a worker and accessing its log is fast!
Each quorum operation should take less than 10 milliseconds
Estimate: 2-3 active extents to repair
Estimate: Less than 15 quorum operations in less than 150 milliseconds.
After that, reading the log will take additional time

[a] These are **very rough estimates**.
[b] Again, **very rough guesses!**

[64] This can be a hunch of what's likely to be needed for log-repair. Occasionally, more may need to be read for correctness.

[65] Resolving fragments in the catalog is complex and subtle if there is concurrent repair. See §15.10: (Picking a Single Outcome Per-Fragment) for a detailed discussion of how to use monotonic quorum operations to reliably resolve ambiguous fragments when differing views are seen by concurrent log-repairers. The explanation covers both a fragment-by-fragment sequence and then how to perform this with bulk requests for rapid log-repair.

15.13 Jitter-Free Retry to Log-Followers

One practical remediation for slow workers comes by adding a *log-following server*. **Log-following servers** process the logs of another worker. They are slightly behind in their understanding of changes made by the worker they follow. Because each read, either *exact-key* or *range-key* is as-of a specific *seniority*, there is no ambiguity about the correct answer. Either the log-following server *has the correct answer* or it knows that *it hasn't processed enough log to give the correct answer.*

Don't use log records unless they will survive log-repair. Log fragments are sent to N_{Log} log-servers. Only after Q_{Log} have confirmed the durability of the fragment on their replica do we know it will persist after log-repair. The worker itself tracks this durable quorum before responding to SQL work from the application. Hence, humans outside the database don't see committed work disappear following log-repair.

The worker also record the lowest fragment in the log-window that is *known to be durable across log-repair*. This is called LDF (Last Durable Fragment)[66]. As a logging worker receives asynchronous confirmation of fragment replica durability, it adds that information to each fragment written moving forward.

Keeping up with the worker. It is quite reasonable to assume a log-follower is probably much less than a second behind the worker it is following. Log-storage servers[67] support reads of the next fragment confirmed by LDF[68] To manage the jitter seen with a single replica of the log, we assume that the log-follower will follow multiple replicas of the log extent, hopefully Q_{Log} of N_{Log}.

Expected Lag of a Log-Follower. The log-server lag is largely dependent on the time it takes for the worker to know its fragment appends are durably replicated across Q_{Log} replicas. Many fragments are launched to log-server replicas to achieve logging throughput. If a logging server sees fragments durably after about 10 milliseconds, the LDF will lag about 15-20 milleseconds. Everything older than that is easy from the log-follower to have available to use as a backup to the logging worker.

Retries to *log-followers* reduce expected latency
Reduces expected tail-latency for most reads
It still does not completely solve the jitter-risk by itself

The very last part of a log cannot be used (yet)
Log records must be durable before used

Log records become *known durable* in two ways:

- **Logger says their durable:** *Later fragments* say *earlier fragments* are known to be durable.
- **We fence & recover the log:** Deciding what's durable

We can't use log records until they are *known durable*

The log's tail is hidden while a sick worker is alive
Fortunately, amount hidden should be less than the last second or so of normal work.

[66] See 15.7. This concept is present in Bookkeeper[7] as the LAC (Last Add Confirmed[8]. It is also present in Amazon's Aurora as the SCL (Segment Complete LSN)[47].

[67] I.e., one of the N_{Log} servers receiving fragments for the extent on an ongoing basis.

[68] Apache Bookkeeper supports *tailing reads* that only return fragments (called *entries in Bookkeeper*) known to be durable across Q_{Log} replicas][8].