

Distributed Eventual Durability

by

Joseph Boulis

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Joseph Boulis 2025

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis is based on a manuscript submitted for publication. Joseph Boulis was the lead researcher for all of the algorithmic and experimental work presented in this thesis, and was the lead implementer of the prototype. Joseph Boulis was the primary contributing author to all sections of this thesis, with secondary contributions from Dr. Sujaya Maiyya and Dr. Kenneth Salem.

Abstract

We present the first design and implementation of a **Distributed *Eventually Durable*** transactional system. Durability latency often dominates transaction commit time in distributed databases, limiting performance for latency-sensitive applications. The Eventual Durability (ED) model (VLDB’24) addresses this by decoupling durability from commit, allowing transactions to commit quickly and persist in the background. So far, the ED model has been applied only to centralized databases. Extending ED to distributed, multi-shard settings introduces new challenges, as each shard persists changes independently potentially making partial changes of a transaction durable while other changes may be lost due to failures.

This work addresses these challenges effectively by first defining a new correctness criterion, **ED Snapshot Isolation** (ED-SI), which ensures that transactions only observe the effects of non-failed committed transactions. To enforce ED-SI, we develop a distributed commit protocol that builds on Percolator and adds validation to prevent reads from failed transactions. Our prototype of this system, called **ED Percolator**, deployed on AWS, supports both *fast* (speculative) and *safe* (durable) transactions to allow applications a latency vs. durability trade off while preserving formal correctness. Experiments show that ED fast transactions offer up to 7.6x speed-up over classical Percolator transactions, while ED safe transactions incur similar latency but abort significantly fewer transactions.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Sujaya Maiyya, for her unwavering support, encouragement, and guidance throughout this journey. Her mentorship has been central to both the completion and the quality of this thesis, as well as to my personal and professional growth. She provided invaluable feedback, challenged me to think deeply about my work, and encouraged me to step beyond my comfort zone, all while continually motivating me to strive for better. I am truly fortunate to have had her as my advisor.

I am also profoundly grateful to Professor Ken Salem for his exceptional patience, generosity with time, and insightful feedback. Through his one-on-one meetings, thoughtful guidance on countless drafts, and advice on presentations, he has helped me develop intuition for research and clarity in communication. His influence has had a lasting impact on the way I approach problems and present ideas.

I would also like to sincerely thank Professor Samer Alkiswany for the time and care he devoted to reviewing my thesis and for his incredibly insightful comments.

I am thankful to my friends in the lab, across ECE and MME, and overseas, whose companionship and support made my time at Waterloo both productive and enjoyable.

Finally, I am indebted to my family and my fiancée for their love, sacrifices, and unwavering belief in me. I could not have reached this point without them, and this thesis is as much theirs as it is mine.

Dedication

To my father Hany, my mother Mariam, my sister Youstina, and my fiancée Sara, whose love and support have been my foundation and strength throughout this journey.

Table of Contents

Author's Declaration	ii
Statement of Contributions	iii
Abstract	iv
Acknowledgements	v
Dedication	vi
List of Figures	ix
1 Introduction	1
1.1 Contributions	1
2 ED Model Overview	4
3 Challenges with Distributed ED	6
4 System Overview	9
4.1 Durability	9
4.2 Failures	11
4.3 Tracking and Managing Durability	11

5	Percolator Overview	14
6	Impact of Not-Yet Durable Writes	17
6.1	Durability Requirement Sets	19
7	ED Percolator Protocol	21
8	Correctness	27
8.1	ED Snapshot Isolation	28
9	Implementation	29
10	Experimental Evaluation	31
10.1	Write Transaction Latency	32
10.2	Read Transaction Latency	34
10.3	Resource Contention	35
10.4	YCSB	36
11	Conclusion	38
12	Related Work and Discussion	39
	References	41
	APPENDICES	45
A	Additional Proofs	46

List of Figures

2.1	ED Transaction Lifecycle. Transactions in green states are visible to others, those in red states are not.	4
3.1	An example depicting the challenges with ED	7
4.1	System Architecture	10
4.2	Epoch Rollover Protocol	12
4.3	Durable, In-Doubt, and Lost Epochs	13
5.1	Commit in Percolator (adapted from [24])	15
6.1	Percolator Failure Analysis. Transactions in green states are visible to other transactions, those in red states are not. Red arrows show state transitions that are triggered by failures of completed writes.	18
7.1	ED Percolator Protocol - Client-Side	22
7.2	ED Percolator Protocol - Store-Side	24
7.3	Changes from fig. 5.1d after a transaction commits (at both the primary and secondary) in ED Percolator.	26
10.1	Write Transaction Latency	33
10.2	Read-Only Transaction Latency	34
10.3	Abort Rates of ED Safe vs. Baseline Write-Only Transactions Across Client Concurrency under Uniform and Skewed Workloads	36
10.4	Comparison of YCSB Throughput and Latency	37

Chapter 1

Introduction

Previous work [18] showed how a centralized transactional system could be modified to support the Eventual Durability (ED) model. However, database sharding – a necessary feature of scalable data systems – results in a *distributed setting* with transactions spanning one or more shards. In this paper, we explore what it means to support the ED model in distributed databases and propose mechanisms to implement distributed ED transactions.

The distributed, multi-shard setting poses unique challenges for the ED model, particularly in handling *partial failures*. A failure of one shard may result in the loss of part, but not all, of a committed, visible, but not-yet-durable multi-shard transaction. However, under the ED model, committed but failed transactions must *disappear* atomically. Hence, we need a way to coordinate and propagate failure information across shards for a correct implementation of distributed ED transactions.

1.1 Contributions

In this paper, we explore how to extend the ED model to distributed settings by first defining a consistency criterion for distributed transactions under the ED model, and then proposing a distributed commit protocol that satisfies that criterion. Specifically, the paper makes the following contributions.

1. Correctness definition

We define *ED Snapshot Isolation* (or ED-SI), which adapts the classical SI definition for the ED model. Intuitively, ED-SI guarantees that each transaction will see a database snapshot consisting of the effects of committed transactions *that have not failed* as of the transaction’s start time (chapter 2 and chapter 8).

2. Distributed transaction protocol

Because the ED model allows committed transactions to subsequently fail, other transactions must somehow be prevented from reading values that were written by such failed transactions. We propose a distributed transaction protocol that achieves this. Under our protocol, remnants of failed committed transactions may remain in the database, but other transactions that inadvertently see them are prevented from committing by means of a commit-time validation step (chapter 7).

We base our work on the Percolator protocol [25] (chapter 5). Percolator implements a client-driven transaction model over a database sharded across BigTable tablet servers. By starting with an existing protocol designed for the classical transaction model, we aim to avoid reinventing the wheel and instead focus on what needs to change to implement the ED model. We expect that adapting other distributed transaction protocols to ED would require similar modifications.

3. Implementation of a multi-shard ED prototype

We implemented a prototype of a distributed ED transaction system, referred to as ED Percolator, deployed on AWS [2] (chapter 9). ED Percolator assumes a key-value database sharded across a set of independent K-V stores that durably store shard data. Stores are implemented using RocksDB [9] running over a persistent storage tier. The ED prototype offers applications two types of transactions: *fast* and *safe*. ED fast transactions are acknowledged at the ED commit point, before they are durable, allowing the application to progress quickly based on speculative durability. ED safe transactions are not acknowledged until the transaction durability point has been reached. These allow application to avoid speculation when necessary for critical operations, at the cost of higher latency.

4. Evaluation

Finally, we present an evaluation of ED Percolator and compare it with our implementation of classical (non-ED) Percolator (chapter 10). We show that ED Percolator is able to offer an effective latency/durability tradeoff to durability-aware applications. ED safe transactions offer latency similar to that of the baseline Percolator, while ED fast transactions are much faster because durability latencies are avoided. The experiments also show

that ED Percolator can reduce server contention, because it can release locks and other resources at the commit point, without waiting for durability.

Chapter 2

ED Model Overview

ED model is an alternative to the classical transaction model. Under ED, transactions may either abort or commit, as in the classical model. They are isolated prior to commit, and they become atomically visible at their commit point. Unlike classical transactions, ED transactions are *not* guaranteed to be durable at their commit point. Normally, committed ED transactions *harden* after commit, and eventually reach a durability point. Once durable, committed transactions are guaranteed to survive subsequent failures and remain visible. Alternatively, committed transactions may fail. A failed transaction *atomically* ceases to be visible as of some failure point. Thus, a transaction that fails is visible for the period between its commit point and its failure point, and is not visible after the failure point. fig. 2.1 illustrates the lifecycle for ED transactions.

Transaction consistency guarantees determine what a transaction "sees" when it reads the database. The distributed ED transaction protocol we present in this paper enforces

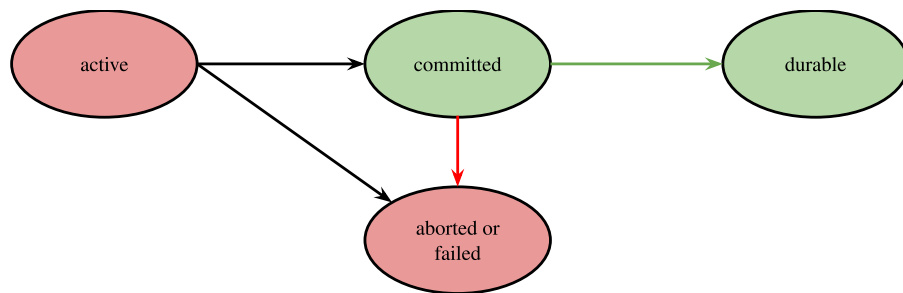


Figure 2.1: ED Transaction Lifecycle. Transactions in green states are visible to others, those in red states are not.

ED snapshot isolation (ED-SI), which is analogous to classical snapshot isolation. We need to define a modification of classical snapshot isolation for the ED model because ED introduces the possibility that visible transactions may cease to be visible, in the event of a failure.

Under classical snapshot isolation, transactions have a start point and a commit point, which are totally ordered. SI requires that a transaction T reads from a database snapshot that reflects all transactions that committed before T started, as if by running them in commit order. Classical SI also requires that concurrent committed transactions have disjoint write sets.

While section 8.1 defines ED-SI more precisely, intuitively ED-SI relaxes the classical definition by demanding that *each transaction read from a snapshot defined by all transactions that have committed **and have not failed** prior to the start of the reader*. This implies that if transaction T fails, all transactions that are read-dependent on T must also fail because transactions running after T 's failure must see a database snapshot without T 's changes.

Chapter 3

Challenges with Distributed ED

Ensuring correctness as defined by ED-SI in distributed multi-shard settings raises several challenges, as we discuss in this section.

Challenge 1: Transactions must fail atomically.

ED-SI states that a transaction T reads from a snapshot of the database reflecting changes from committed but not failed transactions prior to T 's starting point. This implies that if a committed transaction T_C fails to become durable, its changes must *disappear atomically* as of T_C 's failure point. Achieving this in distributed settings is challenging because of partial failures. As shown in Figure 3.1, a distributed committed transaction T_C wrote keys a and x residing in shards 1 and 2 respectively. Shard 2 fails *after* the commit point but *before* making the changes to x durable. A subsequent transaction T_R reading a should not read the value written by T_C because half of its changes are lost. But shard 1, likely hosted on a different machine, might be unaware of this failure.

A simple approach to propagate failures is to have the failing shard notify the other shards of its failure after it recovers. This solution breaks ED-SI because if the transaction T_R , starting after shard 2's failure, reads a and successful commits *before* shard 2 propagates failure, we would have exposed an incorrect snapshot of the database.

To safeguard against this, at commit time, T_R can communicate with all shards in the system to learn of any failures, and if so, abort T_R . However, this solution is costly, and also aborts transactions unnecessarily because failure of a shard i unrelated to T_R will cause its abort. Addressing these challenges requires ***an efficient way to propagate failures and a protocol to fail transactions atomically.***

Challenge 2: Dependent transactions of a failed transaction must also fail.

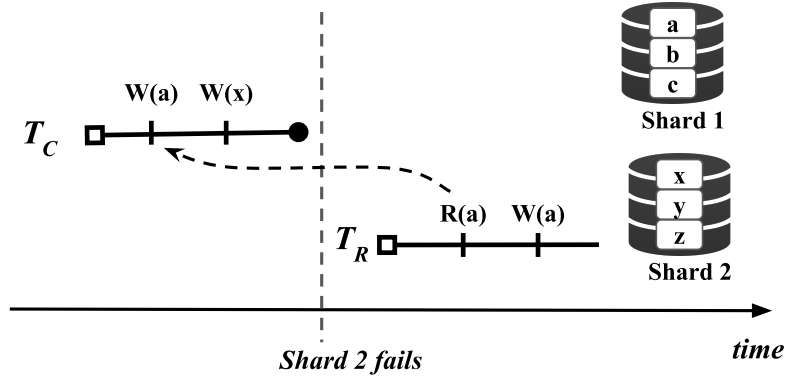


Figure 3.1: An example depicting the challenges with ED

Returning to the example Figure 3.1 consisting of a committed but failed transaction T_C , any committed transactions that are read-dependent of T_C must also *fail* (atomically) when T_C fails. In some cases, there can be a chain of dependent committed transactions $T_C \leftarrow T_1 \leftarrow T_2 \cdots T_i$ that must all fail upon T_C 's failure. Exposing changes from any such dependent transaction after T_C fails breaks ED-SI guarantees.

A naive approach to ensure correct ED-SI behavior is to track transactional dependencies, perhaps in a dependency graph, and associate each transaction with its dependency graph. When a new transaction T_{i+1} commits, the system can traverse the dependency graph to detect whether any of the previous transactions have failed. If so, it can abort T_{i+1} . However, the dependency graph may be arbitrarily large, and traversing it may be expensive. Additionally, the system would also need a carefully devised mechanism to trim the dependency graph to avoid its perpetual growth.

Addressing this challenge requires ***a mechanism to compactly track transactional durability dependencies***.

Challenge 3: Detecting durable vs. failed transactions correctly.

So far, we have not discussed *how* to track whether a transaction eventually becomes durable or fails. This is important because enforcing correctness relies on the ability to identify a failed transactions and make its changes disappear. In addition, the system needs a way to identify which committed transactions have become durable so that it can offer durability-aware applications the ability to wait for durability when necessary, e.g., via safe transactions.

Let us say we mark a committed transaction as durable *if and when all its writes become persistent*, and otherwise we mark it as failed. This approach works for transaction T_C in

Figure 3.1 because a failure of shard 2 before persisting x renders T_C as failed. However, as we have already noted, ED-SI requires that a transaction fail if any of its read dependencies have failed. Thus, a transaction’s durability depends not only on the durability of its own writes, but also on those it has read.

Addressing this problem requires *an approach to track and classify committed transactions as durable or failed*. Note that classical transactional systems also need to track durability, but only prior to commit since durability is guaranteed at commit time; ED transactional systems need to track durability after commit.

Chapter 4

System Overview

Our goal in this work is to design a distributed key/value database system that implements the eventually durable transaction model.

We will assume that the key/value data are sharded across a set of independent *stores*, each of which provides a Get/Put/Delete interface to its shard. We use the terms "store" and "shard" interchangeably in this work. To implement transactions, we interpose a *transaction manager* between application clients and individual stores, as shown in Figure 4.1. The transaction manager (or TM) allows applications to execute transactions involving multiple keys, which may be located in different shards. The transaction manager has both client-side (TM/C) and server-side (TM/S) components, as illustrated in Figure 4.1. The TM's client side forwards operations to the appropriate shard, and acts as the coordinator for transaction commits. The server side of the transaction manager includes a set of procedures that interact with the local store, triggered by client requests. It is also responsible for tracking and managing the durability of writes to its shard, as will be described in section 4.3.

4.1 Durability

Each store is responsible for durably storing the keys in its shard. We are not concerned here with the strength of the durability guarantee provided by the stores, or the specific mechanism used by the store to achieve it. For example, stores might make updates durable by writing them to a local non-volatile storage device, like a disk, or use a replicated distributed log to provide a stronger durability guarantee. In a cloud setting, a store might rely on an underlying cloud storage service to provide durability.

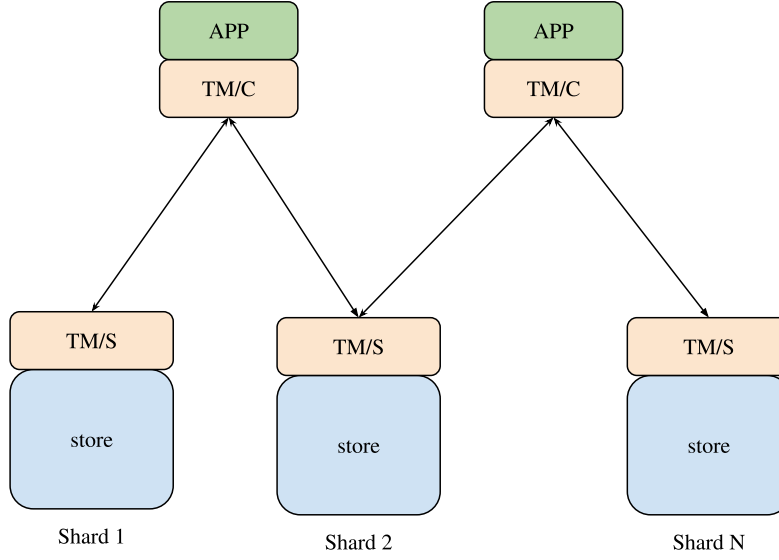


Figure 4.1: System Architecture

To support the ED model, we assume that stores acknowledge write operations quickly, *without waiting for them to be made durable*. The store then makes the writes durable asynchronously, after they have been acknowledged. To help their clients (TM/S, in our case) manage durability, stores expose a **Sync** operation, which blocks until all preceding writes at that store have been made durable. The store, not the TM/S, determines the order in which write operations become durable. The only assumption we make about durability order is that if the TM/S issues two write operations, w_1 and w_2 , *on the same key*, and w_2 follows w_1 , then w_2 does not become durable before w_1 .

We use this asynchronous durability model at the store for two reasons. First, we are trying to implement eventually durable transactions over the stores, and ED transactions become visible first and durable later. We want to take advantage of visible but not-yet-durable writes to allow ED transactions to commit quickly. Second, database and storage systems often work this way. For example, many file systems defer durability guarantees on write operations until a file is closed or explicitly synced by the application. Similarly, database storage managers that use logging for durability typically rely on explicit log flush operations to guarantee the durability of logged updates.

4.2 Failures

Stores may experience failures. The nature of the failure depends on the implementation of the store, and we are not concerned here with how the failure happens. Rather, we simply define *a failure as an event at a store that causes the loss of one or more completed (but not yet durable) write operations*. For example, a server might crash and recover, resulting in the loss of buffered write operations. Under this failure model, a completed write operation at a store can be in one of three possible states:

in-doubt: In-doubt writes are visible but they are vulnerable to loss during a failure. All completed writes start in this state.

durable: Durable writes have been hardened and made persistent by the store. They are visible, and are never lost as a result of subsequent failures.

lost: Writes that have been lost as a result of a failure.

4.3 Tracking and Managing Durability

The TM's server side components (TM/S) are responsible for tracking and managing the durability of writes to their stores. They do this using an epoch-based approach. That is, at each shard the TM divides time into a sequence of disjoint periods, called epochs. Each epoch is uniquely identified by a pair (g, e) , where g is a *generation number* and e is an *epoch number*. A shard's generation number is incremented after each failure event at that shard. Epoch numbers increase monotonically within each generation.

Each TM/S tracks the current epoch at its shard, and associates each new write with the current epoch. It periodically terminates the current epoch, flushing all the pending writes, and begins a new one, using the epoch rollover protocol shown in Figure 4.2. Each TM/S uses its store to maintain a durable *generation counter*, whose value indicates the current generation number at that store. It also persistently maintains a *durability map*, which maps each generation g to the epoch number of the latest durable epoch for g . When a store fails, the TM/S at that store recovers the persistent durability map. The TM/S also increments its generation counter and resets the current epoch number to 1.

Using the generation counter and durability map at its shard, each TM/S can classify every epoch into one of three durability classes. fig. 4.3 illustrates a series of epochs spanning two generations. Epoch (g, e) is classified as **durable** if $d[g] \geq e$, where $d[g]$

```

// current gen and epoch numbers
int CurGen, CurEpoch
AdvanceEpoch() {
    beginCriticalSection()
    prevEpoch = CurEpoch
    CurEpoch += 1
    endCriticalSection()
    // wait for writes to complete
    Await(prevEpoch)
    Sync()
    // update the durability map
    Put(DurMap, CurGen, prevEpoch)
    Sync()
}

```

Figure 4.2: Epoch Rollover Protocol

represents the current value in the durability map for generation g , for example, epoch (5, 2) in the figure. The epoch is classified as **in-doubt** if $d[g] < e$ and g is the current generation, such as epoch (6, 2) in the figure. It is classified as **lost** if $d[g] < e$ and g is not the current generation, such as epoch (5, 3) in fig. 4.3.

TM/S assigns each write to the current epoch and track the durability of writes at epoch granularity. That is, we say that a write belonging to epoch (g, e) is *deemed durable* if epoch (g, e) is **durable**. Similarly, writes are *deemed in-doubt* or *deemed lost* if they belong to **in-doubt** or **lost** epochs. Because the store, and not the TM, controls when and in what order writes actually become durable, the *actual* state of a write may differ from its deemed state. For example, writes belonging to the current epoch at each store are deemed in-doubt until the epoch is rolled over. However, some of those writes may have been made durable by the store prior to rollover. Although a write's deemed state may not match its actual state, the epoch rollover protocol ensures that two important relationships hold. We state them here without proof.

Lemma 4.1 *All writes that are deemed durable are **durable**.*

Lemma 4.2 *All **lost** writes are deemed lost.*

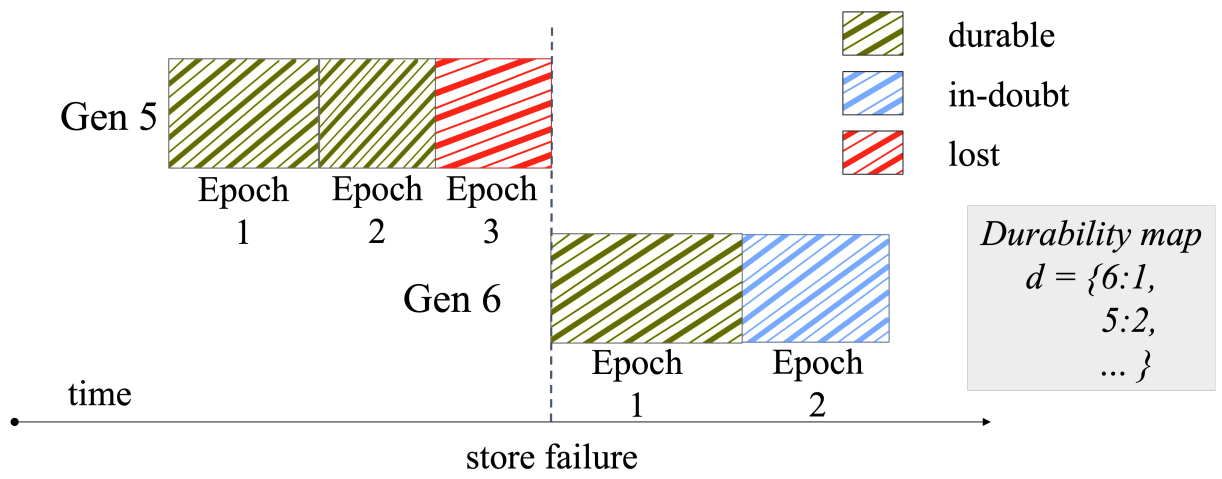


Figure 4.3: Durable, In-Doubt, and Lost Epochs

Chapter 5

Percolator Overview

Building a distributed ED transaction protocol needs two building blocks: (i) persistent stores that support fast writes, which we discussed in chapter 4, and (ii) a classical transaction protocol to be used as a starting point for our implementation, which we discuss in this section. Our ED transaction protocol builds on top of Percolator’s distributed transaction protocol [24].

Percolator uses a client-driven protocol to coordinate transaction execution. Each transaction is managed by a single client (e.g., TM/C in fig. 4.1), and that client buffers transactional writes locally until commit time. At commit time, it orchestrates the commit protocol by interacting with the underlying shards.

Percolator relies on a centralized timestamp service to generate two timestamps for each transaction: a start timestamp (T_s) and a commit timestamp (T_c). Percolator assumes a multi-version key-value storage system with each version of a key indexed by a timestamp. This multi-versioning using transaction timestamps enables Percolator to provide snapshot isolation.

To commit a transaction, the Percolator client chooses one of the keys in the transaction’s write set as the *primary key* for the transactions. It then uses the protocol illustrated in fig. 5.1, which shows the commit of a transaction with two keys, PK (primary) and SK (secondary). The client first writes the primary key’s value along with a *lock*, using the transaction’s start timestamp, as shown in fig. 5.1a. It then writes each of the secondary key values, along with a pointer identifying the location of the primary key’s lock, as shown in fig. 5.1b. At this point the transaction is said to be *pre-committed* with all keys durably written to their stores. But the transaction is not yet committed and thus, not yet visible.

<i>key,ts</i>	<i>value</i>	<i>lock</i>	<i>write</i>
PK, T_s	PKData	primaryLock	

(a) Primary Pre-Commit

<i>key,ts</i>	<i>value</i>	<i>lock</i>	<i>write</i>
PK, T_s	PKData	primaryLock	
SK, T_s	SKData	primary@PK, T_s	

(b) Secondary Pre-Commit

<i>key,ts</i>	<i>value</i>	<i>lock</i>	<i>write</i>
PK, T_c			data@ T_s
PK, T_s	PKData		
SK, T_s	SKData	primary@PK, T_s	

(c) Primary Commit

<i>key,ts</i>	<i>value</i>	<i>lock</i>	<i>write</i>
PK, T_c			data@ T_s
PK, T_s	PKData		
SK, T_c			data@ T_s
SK, T_s	SKData		

(d) Secondary Commit (asynchronous)

Figure 5.1: Commit in Percolator (adapted from [24])

Any new transaction trying to update either PK or SK will observe the locks, which is necessary to detect and resolve conflicts.

To commit the pre-committed transaction, the client atomically removes the lock at the primary key and writes a new record, $data@T_s$, to the primary at the acquired commit timestamp, as shown in fig. 5.1c. The record includes the value at the transaction's start timestamp, which serves as a pointer to the previously-written data, $PKData$, for the primary key. At this point, the transaction is committed and can be acknowledged to the application. It is fully visible, although the secondary key is still locked. Readers that encounter locks at secondary keys will follow the lock pointer to the primary key to learn the status of the transaction. The reader will read the secondary key value if the primary shows that the transaction is committed.

After primary commit, the client asynchronously commits the secondary keys and removes their locks, as shown in fig. 5.1d. These secondary commits do not change the status of the transaction, but they eliminate the need for subsequent readers to check the primary key to learn the status of the transaction.

Because the Percolator protocol is client-driven, a client failure during the commit protocol can leave a partially-committed transaction in the database. Percolator uses a lazy, reader-driven recovery mechanism to resolve such transactions. If a reading transaction encounters a locked key (PK or SK), it waits for a preset time for the lock to be released. After the timeout, it can attempt to abort the writing transaction by removing the writer's primary key record. If it finds that the writer transaction committed but the secondary lock was never removed, it completes the secondary commit on behalf of the original writer transaction.

In Percolator, the state of a transaction is determined by the state of the primary key. If the primary key is locked (fig. 5.1a and fig. 5.1b), the transaction is *active*. If the commit record shown in fig. 5.1c is present, the transaction is *committed*. If the primary key is neither locked nor committed, the transaction is *aborted*.

Chapter 6

Impact of Not-Yet Durable Writes

Each write in Percolator, including pre-commits and commits, is assumed to be durable once a store acknowledges it, which introduces potentially expensive durability waits on the critical path to a transaction’s commit point. In this work, we seek to avoid these delays by decoupling commitment from durability, as permitted by the ED transaction model (fig. 2.1). To enable this, we rely on the data stores’ ability to acknowledge writes quickly, *before* they are durable, as described in chapter 4. While this eliminates durability waits, it violates Percolator’s write durability assumption and introduces the risk that completed writes might eventually fail.

In this section, we present an analysis of the impact of such failures on Percolator. This analysis shows that simply running the original Percolator protocol over data stores that acknowledge writes before they are durable does *not* correctly implement ED transactions. We then use the results of this analysis to guide the design of our ED Percolator protocol (chapter 7). We expect that a similar analysis would be the starting point for adapting other distributed transaction protocols to support ED transactions.

We begin with a few definitions:

Definition 6.1 (Critical Writes) *The critical writes of a transaction are those writes required for the transaction to commit in Percolator. These include (i) the primary pre-commit (fig. 5.1a), (ii) the secondary pre-commits (fig. 5.1b), and (iii) the primary commit (fig. 5.1c). In contrast, the secondary commits (fig. 5.1d) are not critical, as they occur only after the transaction has already committed.*

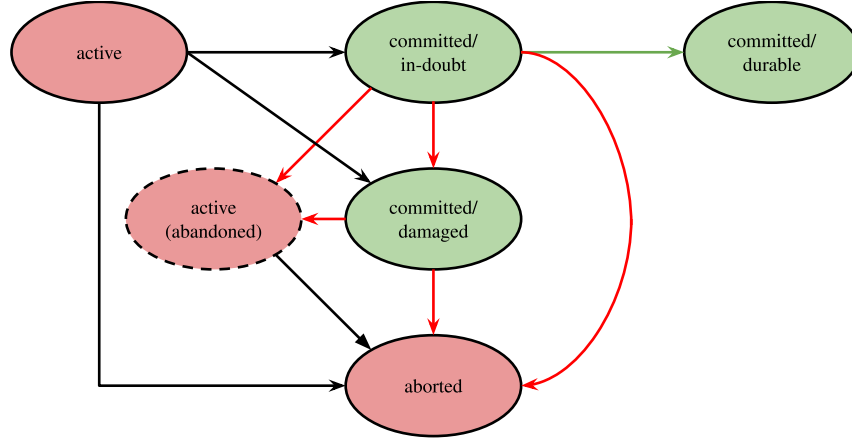


Figure 6.1: Percolator Failure Analysis. Transactions in green states are visible to other transactions, those in red states are not. Red arrows show state transitions that are triggered by failures of completed writes.

Definition 6.2 (Read Dependencies) *The read dependencies of a transaction T are those critical writes that T depends on, either directly or indirectly. For example, if transaction T reads a key x , written by T' , which in turn read key y written by T'' , then T 's read dependencies include the writes to both x and y .*

Definition 6.3 (Durability State of Committed Transactions) *We say that a transaction T is **durable** if all of its critical writes and read dependencies are durable. We say T is **damaged** if at least one of its critical writes or read dependencies is lost. Otherwise, we say that T is **in-doubt**. An **in-doubt** committed transaction is intact, but at least one of its critical writes or read dependencies has not yet become durable.*

With these definitions in hand, we analyze failures of not-yet durable writes in Percolator, as illustrated in fig. 6.1. The diagram shows possible transaction states when failures are considered. This includes three commit states, each representing one of the possible durability states (definition 6.3) of a committed transaction. It also includes an "active (abandoned)" state. This represents active transactions that no longer have a client driving the transaction to commit (e.g., due to client failure). Percolator handles this with a lazy mechanism for other clients to drive such transactions to abort, as described in chapter 5.

The state transitions shown in the diagram include those driven by Percolator itself (in black), as well as those driven by failures of the transaction's critical writes or read dependencies (in red). For example, loss of one of a transaction's secondary pre-commit

writes or one of its read dependencies after commit will move the transaction from the `committed/in-doubt` state to the `committed/damaged` state. Loss of such writes prior to commit will result in the transaction moving directly into the `committed/damaged` state when the client commits it. Loss of a transaction’s primary commit write will revert a `committed/in-doubt` transaction back to an `active` state, from which it will ultimately abort. Loss of both the primary commit and primary pre-commit will move a transaction directly from `committed/in-doubt` to `aborted`.

By juxtaposing fig. 6.1 with the desired ED transaction lifecycle shown in fig. 2.1, we can see that running the original Percolator protocol with writes potentially failing after completion comes close to implementing the ED transaction model. Percolator’s `committed/durable` and `committed/in-doubt` states implement the ED model’s durable and committed states, respectively. Whereas the active and aborted states, in which a transaction is not visible to readers, implement the model’s active, aborted, and failed states. Transitions from green (visible) to red (not visible) states in the Percolator diagram implement the model’s transition from committed to failed state.

However, there is one clear problem, which is the `committed/ damaged` state in fig. 6.1. Because a secondary pre-commit loss or a read dependency loss puts a transaction in `committed/damaged`, its primary commit record could continue to exist. Thus, Percolator will treat it as a committed and hence visible transaction. This creates the risk that readers might see a partial transaction, violating the ED model’s requirement that transactions that fail must fail atomically (Challenge 1 in chapter 3).

Our ED Percolator protocol addresses this problem by treating `committed/damaged` transactions as failed, and hiding their effects from other transactions. To do this, ED Percolator needs a way to identify `committed/damaged` transactions, and distinguish them from other committed transactions. This is done by introducing a new type of metadata, called *durability requirement sets (DRS)*, which we introduce here before presenting the ED Percolator protocol itself in chapter 7.

6.1 Durability Requirement Sets

A durability requirement set describes a set of epochs (section 4.3), from one or more shards. Each element of a DRS is a triple (s, g_s, e_s) , where s identifies a shard, and g_s and e_s are its generation and epoch numbers from that shard. This serves to uniquely identify an epoch across the distributed system. As we will describe in chapter 7, the ED Percolator protocol constructs a DRS for each transaction, capturing all the epochs of its critical writes and read dependencies.

As described in section 4.3, the TM/S at each shard maintains a durability map and generation counter with which we can deem any epoch as **durable** , **in-doubt** , or **lost** . We can *evaluate* a DRS by classifying each individual epoch it represents:

Definition 6.4 (Evaluating a DRS) *A DRS $\mathcal{D}$ evaluates to **durable** if every individual requirement $(s, g_s, e_s) \in \mathcal{D}$ is deemed durable at s . $\mathcal{D}$ evaluates to **damaged** if at least one of its individual requirements is deemed lost. Otherwise, $\mathcal{D}$ evaluates to **in-doubt** .*

Since the durability metadata at each shard evolves as the shard advances its current epoch or experiences a failure, the evaluation of a DRS may change over time. Specifically, at DRS that evaluates to **in-doubt** will eventually evaluate to either **damaged** or **durable** as epochs are made durable or are lost at the shards.

ED Percolator associates a DRS with each committed transaction, and evaluates a transaction’s DRS to distinguish among the three committed transaction states shown in fig. 6.1. That is, if a committed transaction’s DRS evaluates to **durable** , ED Percolator considers the transaction to be in the **committed/durable** state. Similarly, if it evaluates to **damaged** or **in-doubt** , the transaction is in the **committed/damaged** or **committed/in-doubt** state, respectively.

Since ED Percolator tracks durability at the epoch granularity, its DRS-based determination of a committed transaction’s durability state is conservative. In particular, as noted in section 4.3, not all writes in an epoch that is deemed **lost** are necessarily lost. This may cause ED Percolator to conservatively treat an intact transaction as **damaged** . However, because of section 4.3, we can be sure that if a transaction has *actually* lost one of its critical writes or read-dependencies, its DRS *will* evaluate to **damaged** . Conversely, because some writes in **in-doubt** epochs might be durable, ED Percolator might conservatively treat a durable transaction as **in-doubt** . However, because of section 4.3, we can be sure that if a transaction’s DRS does evaluate to **durable** , its critical writes and read-dependencies are *actually* durable.

We define several additional operations on durability requirement sets, and use them in designing ED Percolator. *Reducing* a DRS, written $reduce(\mathcal{D})$, evaluates $\mathcal{D}$ and removes from it any epochs that are **durable** . *Minimizing* $\mathcal{D}$, written $min(\mathcal{D})$ removes redundant elements from $\mathcal{D}$. Specifically, if $\mathcal{D}$ contains two elements with the same store ID and the same generation number, minimization removes the element with the smaller epoch number. Because epochs in each generation are made durable in sequence, and because lost epochs are always the latest epochs in their generation, it is easy to show that $eval(min(\mathcal{D})) = eval(\mathcal{D})$. Finally, *merging* durability requirement sets $\mathcal{D}_1$ and $\mathcal{D}_2$ is defined as $min(\mathcal{D}_1 \cup \mathcal{D}_2)$.

Chapter 7

ED Percolator Protocol

Having described the durability requirement set and its operations, we now present the ED Percolator protocol. figs. 7.1 and 7.2 show the procedures implemented on the client side (TM/C) and on the server side (TM/S) of the ED Percolator transaction manager (fig. 4.1). The two procedures are adapted from the original Percolator pseudocode (Figure 6 in [24]).

```
class Transaction {
    struct Write { Key key; Value value };
    vector<Write> writes;
    int sts;    //start timestamp
    DRS readDRS; //DRS for read dependencies
}
Transaction():start_ts(GetTimestamp()){}
void Put(Write w) {writes.push(w); }
Value Get(Key k)
    v,drs = ReadKey(k, sts);
    readDRS = merge(readDRS,drs); //section 6.1
    return v;
```

```

bool Commit()
    Write p = writes[0]; // primary key
    vector<Write> secondaries = writes[1...];
    // reduce and evaluate readDRS section 6.1
    reducedDRS = {};
    for (r ∈ readDRS)
        switch Evaluate({r})
            case damaged:
                return false;
            case in-doubt:
                reducedDRS=merge(reducedDRS,{r});
    // primary prewrite
    res,d = PreWrite(p,p.key,sts);
    if (!res) return false;
    finalDRS=merge(reducedDRS,d);
    // secondary prewrites
    for (w in secondaries)
        res,d = PreWrite(w,p.key));
        if (!res) return false;
        finalDRS=merge(finalDRS,d);
    int cts = GetTimestamp(); //commit ts
    // primary commit
    res,finalDRS = CommitWrite(p.key,sts,cts,finalDRS,true);
    if (!res) return false;
    // secondary commits – asynchronous
    for (w in secondaries)
        CommitWrite(w.key,sts,cts,finalDRS,false);
    return true
}}

```

Figure 7.1: ED Percolator Protocol - Client-Side

```

// current failure timestamp at this server
int curFailTS // section 4.3

(Value,DRS) ReadKey(Key k, int ts)
    T = StartStoreRowXaction(k);
    // check for concurrent writes
    while T.Read(k,lock,[0,ts])
        // in-flight transaction has a conflicting lock
        BackoffAndMaybeCleanupLock(k);
        continue;
    // find latest write below given timestamp
    latest = T.Read(k,write,[0,ts]);
    if !latest return nil,{ }; // no data
    // read data and DRS of the writer
    d = T.Read(k,drs,[0,ts]);
    val = T.Read(k,value,[latest,latest]);
    return val,d;

(DurEval) Evaluate(DRS d)
    // evaluate using store's durability map
    // result is damaged,in-doubt,or durable
    return eval(d); //section 6.1

(bool,DRS) PreWrite(Write w, Key pk, int ts)
    // abort cross-failure transactions
    if (ts < curFailTS)
        return false;
    T,dreq = StartStoreRowXact(w.Key);
    // abort on conflicting write...
    if (T.Read(w.key,write,[sts,∞])
        return false;
    // ...or conflicting lock
    if (T.Read(w.key,lock,[0,∞]) return false;
    T.Write(w.key,data,sts,w.value);
    T.Write(w.key,lock,sts,pk);
    return T.Commit(),{dreq};

```

```

(bool,DRS) CommitWrite(Key k,int sts,int cts ,
                      DRS xactDRS,bool isPrimary)
T,dreq = StartStoreRowXact(p.key);
if (isPrimary) {
    // must hold lock on primary key
    // to commit primary key
    if (!T.Read(p.key,lock,[sts,sts]))
        return false;
    // add this write to transaction's DRS
    xactDRS=merge(xactDRS,dreq);
}
T.Write(k,write,cts,sts); T.Delete(k,lock,sts);
// write transaction's DRS to the store
T.Write(k,drs,cts,xactDRS);
return (T.Commit(),xactDRS);

```

Figure 7.2: ED Percolator Protocol - Store-Side

The ED client caches the **Put** requests issued by the application locally (fig. 7.1, chapter 7). It implements **Get** requests by sending **ReadKey** requests to the appropriate stores. We assume that clients understand the database sharding and are able to map keys to the stores that hold the keys' shards.

Upon receiving a **ReadKey** request, the store first checks for a lock on the requested key. If a lock is present – indicating a concurrent in-flight writer – the server delays and retries the read, hoping the writer will complete. If this fails, it may attempt to abort the in-flight writer before retrying (fig. 7.2, chapter 7), as in classical Percolator. Unlike the original protocol, ED Percolator returns to the client both the requested value and the DRS of the transaction that wrote that value. The client merges this DRS into **readDRS** to capture the durability requirements of all of the transaction's read dependencies (fig. 7.1, chapter 7).

To commit a transaction, the client first pre-writes the primary key, then pre-writes the secondary keys, and finally commits the primary key. This is the same general behavior as the original Percolator protocol, illustrated in fig. 5.1, but there are several important differences in our ED version.

First, as the commit proceeds, the client builds up a DRS (**finalDRS** in fig. 7.1) capturing all of the transaction's durability requirements. Specifically, the client adds to **readDRS**

a durability requirement for each of the transaction’s critical **Prewrite** operations (fig. 7.1: chapter 7 and chapter 7). The transaction’s final critical write is the primary commit. For this operation, the primary key’s store appends the write’s epoch to the DRS, which is the final DRS associated with the transaction (fig. 7.2, chapter 7). That store then returns the transaction’s final DRS to the client.

Second, ED Percolator confirms, at commit time, that the committing transaction does not depend on any transaction in the **committed/damaged** state; if so, it must abort the committing transaction. To achieve this, ED Percolator *reduces* and *evaluates* the committing transaction’s **readDRS**, which captures all of that transaction’s read dependencies. It does this by sending each requirement in **readDRS** to the appropriate store, as shown in chapters 7 to 7 of fig. 7.1. Requirements that evaluate to **durable** are discarded, as they are already satisfied, while those that are still **in-doubt** are retained in the reduced **readDRS**. If any requirement evaluates to **lost**, indicating the committing transaction depends on a **damaged** transaction, ED Percolator aborts the transaction to ensure ED-SI.

Third, ED Percolator prevents *cross-failure* transactions from committing (fig. 7.2, chapter 7). This prevents transactions from committing if they have a start timestamp before a store’s failure but may have read from the store *after* the failure. To achieve this, each store in ED Percolator obtains a *failure timestamp* from the timestamp service when it recovers from a failure. As we describe in chapter 8, all transactions that fail due to the store’s failure will fail atomically at the failure timestamp, necessary to ensure ED-SI.

Fourth, ED Percolator durably stores the transaction’s **finalDRS** along with the transaction’s commit record (fig. 7.2, chapter 7). fig. 7.3 illustrates the state of a committed transaction in ED Percolator. The ED protocol adds an extra metadata field, called **drs**, to the transaction’s commit record to store its final DRS. This is stored both at the primary and the secondary keys, during their respective commit phases. When other transactions subsequently read any of these keys, they read both the values and the associated DRS.

Recall Challenge 2 from chapter 3 which discusses the inefficiency of traditional approaches such as dependency graphs in tracking read dependencies for durability check. ED Percolator instead computes and stores a DRS for each transaction to track the durability of that transaction and its dependencies. Because of the way it is constructed, the **finalDRS** stored with each transaction is bounded in size by the number of database shards, regardless of the transaction’s size or the depth of dependency.

Lemma 7.1 *A transaction’s **finalDRS** has at most one durability requirement per shard.*

See appendix A.

<i>key,ts</i>	<i>value</i>	<i>lock</i>	<i>write</i>	<i>drs</i>
PK, T_c			$data@T_s$	finalDRS
PK, T_s	PKData			
SK, T_c			$data@T_s$	finalDRS
SK, T_s	SKData			

Figure 7.3: Changes from fig. 5.1d after a transaction commits (at both the primary and secondary) in ED Percolator.

For the sake of simplicity, fig. 7.1 describes the ED Percolator protocol as if DRS evaluations happen one requirement at a time, and pre-write and commit operations happen one key at a time. However, our prototype actually batches DRS evaluations, pre-writes, and commits, and piggybacks DRS evaluations onto **PreWrite** requests when possible. Indeed, this is the primary reason that the ED Percolator protocol defers checking of read durability requirements to commit time, instead of performing them at the time of the read. This is described in more detail in chapter 9.

The net result of these optimizations is that ED Percolator requires, in general, three synchronous rounds of communication to commit a transaction, like Percolator. The first round pre-writes and evaluates DRS requirements at the shard holding the transaction’s primary key. The second round pre-writes keys and evaluates requirements at all remaining shards touched by the transaction. These shards can be handled in parallel. Finally, the third round handles the commit of the primary key at its shard. Secondary commits require fourth round of messages, but, similar to Percolator, this round is asynchronous and does not impact transaction latency.

Chapter 8

Correctness

The ED Percolator protocol is designed to enforce ED snapshot isolation (ED-SI). Here, we define the ED-SI guarantee, and show that ED Percolator correctly enforces it.

Like the original Percolator protocol, ED Percolator relies on a timestamp oracle to assign timestamps to transaction start and commit events. Timestamps determine the order of events enforced by the protocol. In particular, a transaction with start timestamp t_s will "see" updates made by transactions with commit timestamps less than t_s .

For an ED protocol, we also need to order transaction failures with respect to other transaction events. This is because ED transactions cease to be visible *at the point at which they fail*. As described in chapter 4, committed transactions are damaged when one or more of their critical writes are lost, and writes are lost as a result of store failures. To order transaction failures, we order the store failures that cause them.

If a shard failure results in the failure of one or more transactions, ED Percolator ensures that those transactions will appear to fail atomically at the time of the shard failure. As described in chapter 7, when a shard fails, it obtains a failure timestamp before resuming handling of transaction requests. If the shard's failure damaged a transaction, ED Percolator will place that transaction's *failure point* at the shard's failure timestamp. That is, ED Percolator will ensure that the transaction appears to fail atomically as of the failure timestamp. The only exception to this is transactions that are damaged *before* they commit, e.g., losing a secondary pre-write before the primary commit. For such transactions, ED Percolator places the transaction's failure point at the same time as its commit point. We call these *failed-at-commit* transactions. Although they commit, they are never visible to other transactions.

8.1 ED Snapshot Isolation

ED Percolator enforces ED-SI, which is a variant of the classical SI consistency guarantee. An ED variant is needed because the ED transaction model introduces the possibility of transactions failing and ceasing to be visible after committing.

Definition 8.1 (ED Snapshot Isolation) *An ED transaction execution history is ED snapshot isolated if both of the following conditions hold for all transactions T :*

1. *If T commits, it has read from a database snapshot resulting from the execution, in commit timestamp order, of all transactions whose commit timestamp precedes T 's start timestamp, and whose failure timestamp, if there is one, does not precede T 's start timestamp.*
2. *For every transaction T' that is concurrent with T and whose write set overlaps with T 's, at least one of T or T' does not commit, or is failed-at-commit.*

Two transactions are considered concurrent if the intervals between their start and commit timestamps overlap. ED-SI is similar to the classical SI condition, which also constrains what transactions read, and prohibits write-write conflicts between concurrent transactions. Together, theorems 8.1 and 8.2, show that ED Percolator guarantees ED Snapshot Isolation. We state them here without proofs, which can be found in appendix A.

Theorem 8.1 *Under ED Percolator, each committed transaction T reads from a database snapshot resulting from the execution, in commit timestamp order, of all transactions whose commit timestamp precedes T 's start timestamp, and that was not "deemed failed" as of T 's start timestamp.*

Theorem 8.2 *For every pair of concurrent ED Percolator transactions, T_1 and T_2 , with overlapping write sets, at least one of T_1 or T_2 does not commit, or is failed-at-commit.*

Chapter 9

Implementation

We implemented a prototype system, written in Go, to execute distributed transactions. The prototype can operate in two modes: baseline and ED. In baseline mode, the prototype executes transactions using the classical Percolator protocol [25]. In ED mode, it implements our ED Percolator protocol.

The prototype consists of implementations of the client- and server-side components of the transaction manager (TM/C and TM/S in fig. 4.1). In the ED model, these components implement optimized versions of the algorithms shown in figs. 7.1 and 7.2. As was noted in chapter 7, the optimizations include batching of pre-write and commit operations and piggybacking of DRS evaluation with pre-write requests. The pre-write and commit batching optimizations are also applied to the baseline Percolator mode.

The ED model allows the transaction manager to expose a latency vs. durability tradeoff to applications, because of the distinction between transactions' commit and durability points. The ED model itself does not prescribe a particular application interface for exposing the tradeoff. In ED mode, our prototype does so by offering two types of transaction commit operations, *ED fast* and *ED safe*, as was done by Kashi et al. [18]. Both fast and safe transactions commit quickly and release the lock at commit time, the difference lies in *when* they acknowledge the commit to the application.

For a fast commit, the TM acknowledges the commit as soon as the transaction commits, without waiting to ensure its durability. Safe commits are only acknowledged after the transaction reaches its durability point, meaning that all of its critical writes and read dependencies are durable. Specifically, when a safe transaction's `finalDRS` evaluates to `durable`, the TM/C acknowledges its commit. The TM/C caches servers' durability maps

so it can perform DRS evaluations locally, without additional per-transaction server interactions. The TM/C regularly polls servers to obtain the latest durability map and update its cache. The application manages the latency vs. durability tradeoff by choosing whether each transaction will be fast or safe.

On the server side, our prototype uses RocksDB [9] to implement shard storage. RocksDB, in turn, relies on some underlying storage tier to provide durable storage of its database and logs. We experimented with two different storage tiers, offering different degrees of durability, as described in chapter 10.

The prototype uses RocksDB’s pessimistic transactions to enable per-key atomic shard updates. RocksDB’s transactions support atomic updates across multiple keys, but we rely only on single-key atomics in the prototype. We configure RocksDB to use manual flushing of its transactional write-ahead log. This means that RocksDB transactions complete quickly but are not guaranteed to be durable when committed. Instead, the transaction manager (TM/S) can explicitly issue RocksDB `FlushWAL` commands to enforce durability guarantees by forcing synchronization of the RocksDB log. This implements the store `Sync` operation expected by our system model (chapter 4). In baseline mode, TM/S issues a `Sync` after every request batch at server. Thus, all server requests are durable when acknowledged to the client, as expected by the original Percolator protocol. In ED mode, request batches are not durable when the server acknowledges them to the client. Instead, each TM/S uses a dedicated thread to continuously run the EpochRollover protocol (fig. 4.2) and update durability info. That protocol defines specific storage synchronization points, which are enforced using `Sync`.

Chapter 10

Experimental Evaluation

This section investigates how eventual durability (ED) influences performance of a multi-shard distributed database compared to its non-ED counterpart. Specifically, we compare ED Percolator’s fast and safe transactions with classical Percolator, first on a suite of micro-benchmarks followed by a YCSB benchmark, which generates a richer workload.

Experimental Setup: We conducted experiments on Amazon Web Services (AWS) using three instances. One t3.large EC2 instance (2 vCPUs, 8 GiB memory) is used as the client machine, hosting the experiment driver application and the client side of the transaction manager (TM/C). Two m5ad.xlarge instances (4 vCPUs, 16 GiB memory each) are used to implement stores, each managing a shard of the test database. One of those two instances also hosts the timestamp oracle.

We experimented with two different storage tiers: Elastic Block Store (EBS) and Elastic File System (EFS). These provide the durable storage for the shard servers. EBS replicates the shard within a single AWS Availability Zone (AZ), while EFS replicates across multiple AZs within an AWS region. Both storage tiers ensure that durable changes will survive a failure and restart the store’s instance. EFS additionally ensures that the database will be durable even in the event of the loss of an AZ. Unless otherwise specified, experiments use EFS.

For all experiments, the transaction manager (TM/S) was configured to use a retry delay, randomly selected between 0 and 2.5 milliseconds when encountering a lock when reading, in both ED mode and baseline mode. In ED mode, the client(s) are configured with a polling interval of 250ms for updating their durability cache.

Each observed data point in this section is an average of three runs to account for the performance fluctuations incurred in a multi-tenant cloud. The number of keys per

transaction, the distribution of application workload (i.e., uniform vs. skewed), amount of read/write-only vs. read-write transactions can all vary per experiment. We state these for each experiment.

The experiments measure three quantities: (1) *end-to-end latency*, which is the total elapsed time between transaction start and commit acknowledgement, as measured at the client; (2) *throughput*, which is the number of successfully committed transactions per second; and (3) *abort rates*, which is the percentage of transactions started by a client that are ultimately aborted.

10.1 Write Transaction Latency

The goal of ED Percolator is to offer applications a principled way to trade durability for reduced transaction latency. The purpose of our first experiment is to test the extent to which this is achieved, in a simple setting.

In this experiment, a single client issues 10,000 write transactions sequentially against a database with 100,000 keys, sharded across the two servers. Each transaction consists of writes to two keys, chosen randomly from the keyspace using a uniform distribution. We report the mean end-to-end latency of these transactions.

We ran several versions of this experiment: (1) **ED Fast**, which runs in ED mode with transactions issued by the client using fast commits; (2) **ED Safe**, in which all transactions use safe commits; (3) **Baseline (sync=on)**, which runs in the baseline Percolator mode; and (4) **Baseline (sync=off)**, which runs in baseline mode, but without issuing any **Sync** operations. We ran each version using both storage tiers and summarize the results in fig. 10.1. The figure also breaks down the reported end-to-end latency to illustrate where the latency comes from.

We make several observations. First, by comparing ED Fast and ED Safe, we see that the ED mode is successfully offering a latency tradeoff. Safe transactions require about 9ms on EBS, while fast transactions complete in a little over 3ms. The savings is greater on EFS, which offers a stronger durability guarantee. The latency breakdowns show that both fast and safe transactions complete the various phases of the commit protocol in about the same amount of time, as expected. The higher latency for safe transactions stems from the client waiting for the committed transactions to become durable before acknowledging them to the driver application.

Second, we see that ED safe transactions have approximately the same latency as baseline transactions (with **Sync** on). Thus, there is little or no latency penalty to be paid

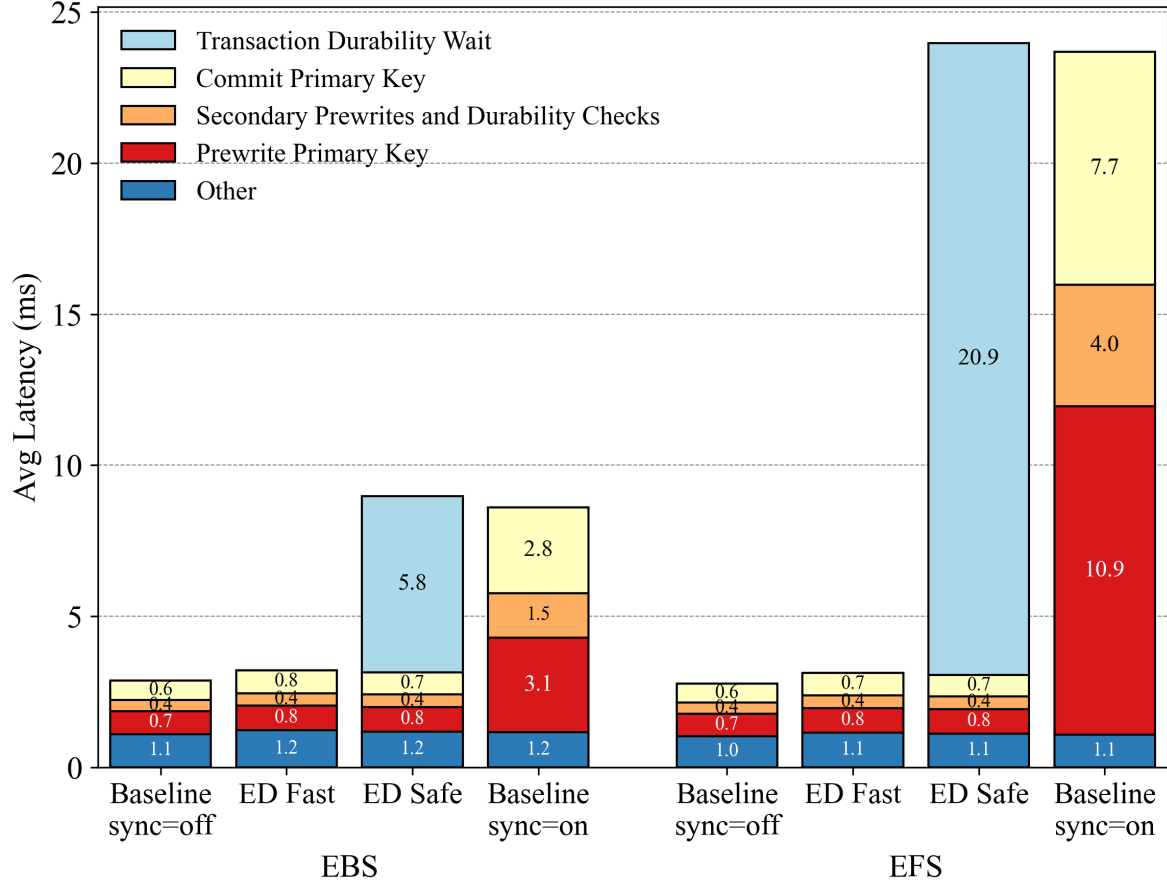


Figure 10.1: Write Transaction Latency

for the extra flexibility provided by the ED mode. However, the *source* of the latency is very different in the two cases. In the baseline, each phase of the commit protocol is relatively slow, because it requires a durable **Sync** after each phase. In ED mode, commit is reached quickly, without durability delays. Instead, safe transactions wait at TM/C until their changes have become durable before being acknowledged.

Third, we see that ED fast transactions have about the same latency as the baseline with **Sync=off**. Neither configuration experiences any durability-related delays. This comparison indicates that the ED mode is not introducing any significant overheads with respect to the baseline system.

10.2 Read Transaction Latency

With our next experiment, we seek to understand the performance of read-only ED transactions. We run an experiment in ED mode in which three concurrent clients issue requests to a database with 100,000 keys, sharded across two servers as in the previous experiment. Two clients issue read-only transactions, each of which reads two randomly selected keys from the database. One reader issues fast commits for its transactions, the other issues safe commits. The third client issues two-key write transactions, so that the database is not static while the readers are reading. We tested with both uniform and skewed request distributions, with the latter following a Zipf=1.05 distribution. We report the mean end-to-end latency for fast and safe read-only transactions, both with and without client-side durability map caching. For comparison, we also ran this experiment (two readers, one writer) in baseline mode.

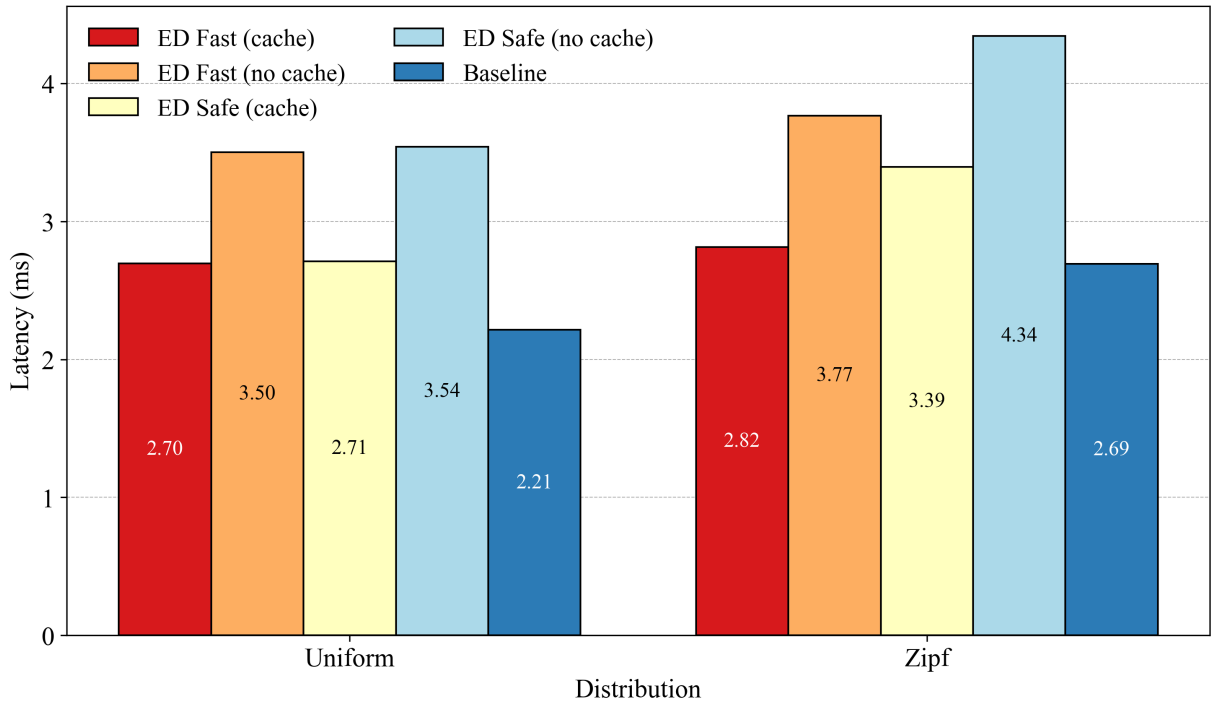


Figure 10.2: Read-Only Transaction Latency

The results are shown in Figure 10.2. Our first observation is that under the uniform request distribution, both fast and safe read-only transactions are slightly slower (about

0.5ms difference) than those of the baseline. This is the result of the additional commit-time check that is needed in ED mode to confirm that transactions have not seen the effects of failed transactions. Since this check is required *before* commit, it effects the performance of both fast and safe transactions. The results also show that this penalty would be substantially higher without client-side durability map caching, as the cache allows TM/C to evaluate a committing transaction’s DRS locally.

The results are similar under the skewed distribution, except that we see that the latency for ED safe transactions is now somewhat higher than that of ED fast transactions. ED safe transactions are not acknowledged to the application until their durability point has been reached, and a read-only transaction is not durable until its read-dependencies are durable. Under a skewed distribution, a read-only transaction is more likely to have seen a recently-written update, which results in a durability wait. This is the source of the extra latency for the safe transactions. Fast read-only transactions do not make durability guarantees at commit time, so they do not experience such waits. Note that the increases contention causes a slight increase in latency for both ED and baseline transactions.

10.3 Resource Contention

ED safe transactions and baseline transactions both offer the same durability guarantee at commit time, and they have similar end-to-end latency. However, as shown in fig. 10.1, ED transactions commit quickly and then wait, while baseline transactions commit slowly. This means that servers can potentially release resources more quickly in ED mode than in baseline mode. In particular, for ED Percolator, servers release transaction locks at commit time.

To test this, we repeat the experiment from section 10.1, but with increasing numbers of concurrent clients, and using both uniform and skewed request distributions. This time, we measure the transaction abort rate in ED safe and baseline modes, as a function of the number of concurrent clients. The results are shown in Figure 10.3.

Under the uniform workload, which has little contention, abort rates are low for both ED safe and baseline transactions. However, under a skewed workload (Zipf 1.05), clients experience aborts due to lock conflicts, and the lock conflict rate is much higher in the baseline, because it retains locks much longer. A variety of early lock release techniques have been proposed for classical transaction systems, so this optimization is not new [7, 17, 11, 12]. However, this does show that ED systems will ”naturally” obtain the benefits of early resource release because of the separation of commit and durability in the ED model.

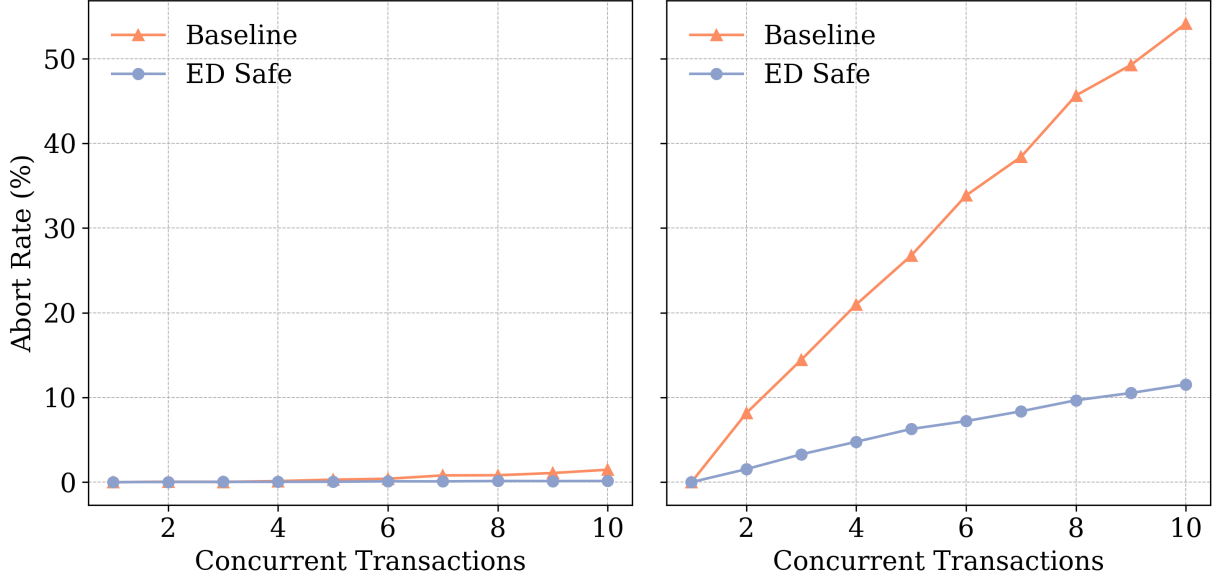


Figure 10.3: Abort Rates of ED Safe vs. Baseline Write-Only Transactions Across Client Concurrency under Uniform and Skewed Workloads

10.4 YCSB

We tested ED mode under a YCSB-derived workload to see whether ED Percolator’s latency/durability tradeoff holds under a richer workload than we have tested so far. In this experiment, each client runs transactions consisting of five operations. Operations are reads or writes, chosen according to YCSB [5] Workload A, which includes a 50:50 read/write mix and a skewed (Zipf 0.99) request distribution. Each client runs for one minute, and we measure the throughput and latency as a function of the number of concurrent clients. We used ED mode and ran the experiment once with fast transactions and once with safe transactions. The results are shown in Figure 10.4.

This experiment shows that ED Percolator continues to offer an effective latency/durability tradeoff under this workload. Fast YCSB transactions are about 5x faster than safe ones under low load, and about 2.5x faster under the highest load we tested.

We also tested the baseline system under this workload. Though it is not shown in fig. 10.4, we found that it had consistently higher latency and lower throughput than ED safe. However, the baseline’s performance was hurt by the relatively large number of **Sync** operations it performs, relative to ED mode. It may be possible to ameliorate this with

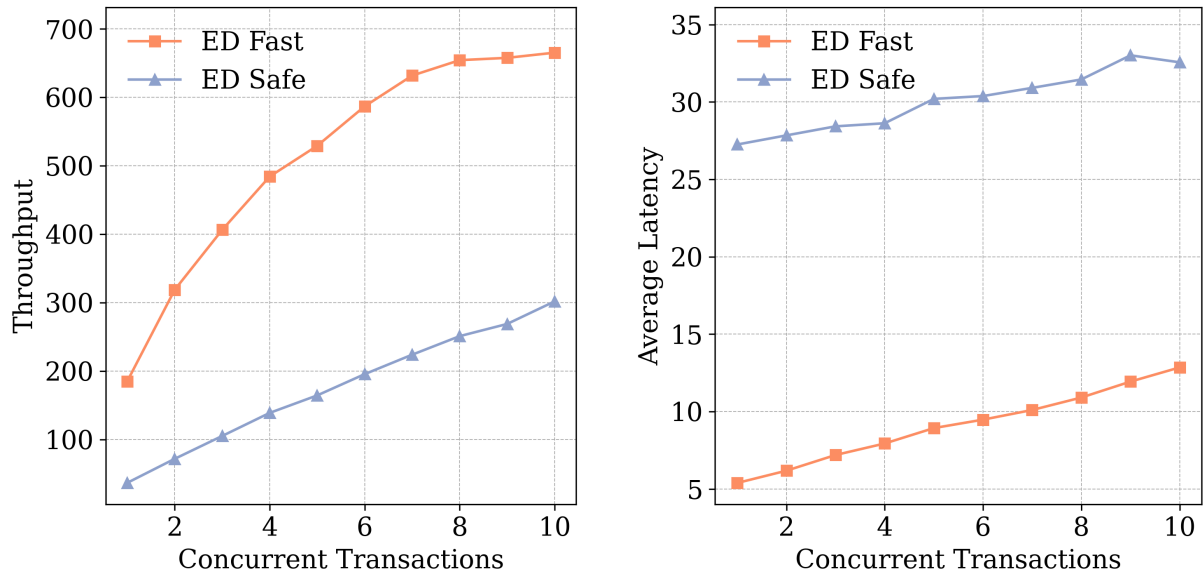


Figure 10.4: Comparison of YCSB Throughput and Latency

careful tuning of the underlying RocksDB system, but the ED system avoids this issue by syncing much less frequently (only at epoch boundaries).

Chapter 11

Conclusion

This work presents the design and implementation of a Distributed Eventually Durable transactional system. Eventual Durability (ED) model, which is an alternative to the classical transaction model, decouples transaction commitment from its durability and enables a system to commit *fast* while making changes durable in the background. This work extends the ED model (VLDB’24) to distributed, multi-shard databases. A committed multi-shard transaction, whose changes are visible to others, may fail due to a shard failure and cease to exist. To accommodate such histories, we propose *ED Snapshot Isolation* (ED-SI), which dictates that despite any failures to committed transactions, no transaction will see partial changes of a transaction. We design and implement *ED Percolator* that proposes mechanisms to guarantee ED-SI. Comparing the results of ED Percolator with its classical, non-ED counterpart, we observe that ED transactions offer up to **7.6x** speed-up while committing transactions. We believe that the building blocks of ED Percolator can be integrated with other distributed transaction systems.

Chapter 12

Related Work and Discussion

To support latency-critical applications, many database systems can be configured to sacrifice durability in exchange for lower transaction latencies. For example, PostgreSQL [29] and Oracle TimesTen [23] can be configured to use *asynchronous commit*, which acknowledges commit requests before the transaction is durable. These ad hoc techniques are effective, but their exact behavior and failure semantics are system specific, and it may be difficult for applications to distinguish which data are durable and which are not. In contrast, the ED model not only offers a latency/durability trade-off but also a system-independent basis for defining correctness.

Our ED Percolator system builds on an epoch-based durability model. Several proposals offer different forms of epoch-based group commit [8] to improve the efficiency of transactions. FASTER [26], a centralized transactional K-V store, aims for high transaction throughput by using epoch-based commit to avoid the serial log bottleneck. In case of failures, transactions in uncommitted epochs may be lost. FASTER performs *concurrent prefix recovery (CPR)*, which provides a session-oriented guarantee about the resulting state. Similarly, COCO [22] implements a form of transactional group commit in a distributed setting, over shared data. It batches transactions into epochs and commits them all as a single unit, with the aim of boosting the efficiency of transaction processing. However, like other group commit implementations, this can come at the expense of increased latency, since transactions need to be formed into batches. In contrast, ED Percolator’s goal is to reduce latency.

DPR [20] extends CPR to a distributed setting, over sharded data. Applications establish sessions and perform sequences of atomic, single-shard operations within each session. Optionally, operations may be acknowledged once complete, without waiting for durabil-

ity. In case of failures, recently completed but non-yet-durable operations may be lost, but the system enforces a consistency guarantee that all dependencies of a surviving operation will also survive. However, DPR is not designed to provide multi-operation cross-shard transactions, which are the focus of our work.

In replicated databases, Li et al. [19] introduce RedBlue consistency, allowing applications to mix fast, locally-executed (blue) operations with slower, strongly consistent (red) ones, similar to how our API exposes fast and safe transactions. Saucr [1] reduces durability costs by adaptively deciding whether to persist updates in memory or on disk, depending on system availability, but still requires coordination with a majority of replicas before acknowledging updates. Whereas, Orca [10] lowers update latency by deferring durability until updates are read. The ED model adopts a similar idea for transactions, delaying a transaction’s durability until all its updates and dependencies are durable.

In sharded databases such as CockroachDB [28], Google Spanner [6], TiKV [16], and PolarDB [3], while implementations vary, commit protocols share foundational similarities with Percolator, which is based on Two-Phase Commit [14]. While systems like TiKV have minimal differences with Percolator, Spanner and CockroachDB introduce larger deviations. For example, unlike Percolator’s client-driven protocol, Spanner designates one shard as a transaction coordinator (similar to Percolator’s TM) to execute various transactional phases. Spanner and CockroachDB replace the centralized timestamp oracle with TrueTime or Hybrid Logical Clocks, using which, they achieve serializable isolation. CockroachDB introduces parallel commits [4], wherein instead of Percolator’s sequential phases (primary prewrite, secondary prewrites, and primary commit), it issues writes to shards in parallel and collects write intents. While these are ongoing, the client sends a commit intent to the primary key’s shard. Once all writes complete, the transaction can be considered committed.

Despite these variations, many of the techniques in ED Percolator can be adapted to these protocols. The first change is to add a durability marker (e.g., generation and epoch numbers) that each shard returns with its write. This marker should encapsulate failure information (similar to the generation number). The client or coordinator collects these markers and includes them in the commit request once all writes are complete. Since these protocols track transaction state at the coordinator or primary key shard, the durability markers can be stored with the transaction record. Finally, because commitment does not imply durability, the commit protocol must verify these markers to ensure the transaction reflects a valid database state. We hope the techniques and correctness arguments in this paper serve as a starting point for introducing ED to these systems.

References

- [1] Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {Fault-Tolerance}, fast and slow: Exploiting failure asynchrony in distributed systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 390–408, 2018.
- [2] Amazon Web Services, Inc. Amazon Web Services (AWS). <https://aws.amazon.com/>, 2025. Accessed: 2025-07-15.
- [3] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, Han Liao, Zilin Chen, Xiaojian Fang, Mo Chen, Chenghui Liang, Yanxin Luo, Huanming Wang, Songlei Wang, Zhanfeng Ma, Xinjun Yang, Xiang Peng, Yubin Ruan, Yuhui Wang, Jie Zhou, Jianying Wang, Qingda Hu, and Junbin Kang. Polardb-x: An elastic distributed relational database for cloud-native applications. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 2859–2872, 2022.
- [4] CockroachDB’s Parallel Commits. <https://www.cockroachlabs.com/blog/parallel-commits/>, 2025. Accessed Jul 10, 2025.
- [5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC ’10*, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.
- [6] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang,

- and Dale Woodford. Spanner: Google’s globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), August 2013.
- [7] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data*, pages 1–8, 1984.
 - [8] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data*, pages 1–8, 1984.
 - [9] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), October 2021.
 - [10] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Strong and efficient consistency with consistency-aware durability. *ACM Transactions on Storage (TOS)*, 17(1):1–27, 2021.
 - [11] Goetz Graefe, Hideaki Kimura, and Harumi Kuno. Foster b-trees. *ACM Transactions on Database Systems (TODS)*, 37(3):1–29, 2012.
 - [12] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 85–96, 2013.
 - [13] Jim Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB)*, pages 144–154. VLDB Endowment, 1981. Invited Paper.
 - [14] Jim N. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60, pages 393–481. Springer-Verlag, Berlin, Heidelberg, New York, 1978.
 - [15] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

- [16] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. Tidb: a raft-based htap database. *Proc. VLDB Endow.*, 13(12):3072–3084, August 2020.
- [17] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 24–35, 2009.
- [18] Tejasvi Kashi, Kenneth Salem, Jaemyung Kim, and Khuzaima Daudjee. Eventual durability. *Proceedings of the VLDB Endowment*, 17(13):4733–4745, 2024.
- [19] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making {Geo-Replicated} systems fast as possible, consistent when necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, 2012.
- [20] Tianyu Li, Badrish Chandramouli, Jose M Faleiro, Samuel Madden, and Donald Kossmann. Asynchronous Prefix Recoverability for Fast Distributed Stores. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1090–1102, 2021.
- [21] Wei Lin, Mao Yang, Lintao Zhang, and Lidong Zhou. Pacifica: Replication in log-based distributed storage systems. 2008.
- [22] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Epoch-based commit and replication in distributed OLTP databases. *Proceedings of the VLDB Endowment*, 14(5):743–756, January 2021.
- [23] Oracle. *Oracle TimesTen In-Memory Database*, release 22.1 edition, August 2023.
- [24] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.
- [25] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI’10, page 251–264, USA, 2010. USENIX Association.

- [26] Guna Prasaad, Badrish Chandramouli, and Donald Kossmann. Concurrent Prefix Recovery: Performing CPR on a Database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 687–704, 2019.
- [27] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 729–730, New York, NY, USA, 2012. Association for Computing Machinery.
- [28] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pages 1493–1509, 2020.
- [29] Postgresql 16.1 documentation, 2023.

APPENDICES

Appendix A

Additional Proofs

This section includes theorem proofs that were removed from the body of the paper because of length restrictions.

[7.1] A transaction’s `finalDRS` has at most one durability requirement per shard.

The reduction of a transaction’s `readDRS` leaves only durability requirements that evaluate to `in-doubt`. All `in-doubt` requirements for a given shard refer to that shard’s current generation number (section 4.3). Thus, merging reduced requirements into `reducedReadDRS` leaves at most one requirement per shard, with the shard’s current generation number. To generate the `finalDRS`, the ED Percolator client merges additional requirements into the `reducedReadDRS` to enforce the durability of the transaction’s critical writes. However, if any of these merged requirements are for the same shard as an existing requirement in the `reducedReadDRS`, they must also share the same generation number as the existing requirement. Thus, the merge operation will retain only a single requirement for that shard.

[8.1] Under ED Percolator, each committed transaction T reads from a database snapshot resulting from the execution, in commit timestamp order, of all transactions whose commit timestamp precedes T ’s start timestamp, and that was not “deemed failed” as of T ’s start timestamp.

ED Percolator’s read protocol (fig. 7.2), like Percolator’s, returns the latest *visible* version of a key prior to the reader’s start timestamp. In the absence of store failures, no committed transactions are deemed failed, and ED-SI’s read condition is equivalent to that of classical SI. The theorem holds because ED Percolator behaves like Percolator when there are no failures, and Percolator guarantees classical SI. However, failures introduce two new risks, which must be handled by ED Percolator:

Case 1: Reading from a damaged transaction

Consider a transaction T_w that is deemed failed before T 's start timestamp. Although T_w must appear to fail atomically at the time of the failure, undamaged parts of T_w may remain in the database when T starts. Allowing T to read from T_w and commit would violate the theorem.

Suppose this occurs, and T reads a surviving update from a deemed failed transaction T_w . In ED Percolator, T also reads T_w 's DRS. After all of its reads, during T 's prewrite phase, T evaluates T 's DRS, which incorporates T_w 's. Since T_w was already deemed failed prior to T 's start, T 's DRS must be deemed failed as well when it is evaluated. Since this will trigger ED Percolator to abort T , the theorem holds in this case.

Case 2: Failing to read from a damaged transaction

Consider a transaction T_w that writes x , commits, and does not fail prior to T 's start timestamp. Suppose T reads x , and T_w is the last transaction to update x . If T commits, the theorem requires that it must have read T_w 's version of x . However, suppose a failure at x 's shard after T starts causes the loss of T_w 's update to x before T issues its read. T cannot see T_w 's version of x , and will instead read some earlier version. This violates ED-SI because when T started, T_w was a committed and un-failed transaction. Therefore, ED Percolator prevents T from reading from a shard that has experienced a failure since T 's start timestamp (fig. 7.2, chapter 7).

[8.2] For every pair of concurrent ED Percolator transactions, T_1 and T_2 , with overlapping write sets, at least one of T_1 or T_2 does not commit, or is failed-at-commit.

Like Percolator, ED Percolator detects write-write conflicts during the prewrite phase of commit (fig. 7.2, chapter 7). However, in the ED protocol, it is possible for this check to fail if a prewrite is lost due to a failure before the check is made. Suppose that both T_1 and T_2 update a key x , and, without loss of generality, suppose that T_1 prewrites x before T_2 does. If a failure at x 's shard causes T_1 's prewrite to be lost before T_2 's prewrite of x , T_2 will fail to detect the conflict during its prewrite. However, in this situation, one of two things must happen. If T_2 's start timestamp is before the failure timestamp at x 's shard, T_2 will abort when it attempts to prewrite x , because of the cross-failure timestamp check (fig. 7.2, chapter 7). If T_2 's start timestamp is after the failure timestamp at x 's shard and T_1 commits, then T_1 's commit timestamp must also be after the failure timestamp, since T_1 and T_2 are concurrent (by assumption). In this case, T_1 's DRS must include a durability requirement for its prewrite of x , which has failed before T_1 's commit. Thus, T_1 is failed-at-commit, as it will be deemed failed by its DRS as of its commit point.