Eventual Durability of ACID Transactions in Database Systems

by

Tejasvi Kashi

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

Waterloo, Ontario, Canada, 2023

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

I am the sole author of this thesis with the exception of parts in Chapter 2.

Chapter 2 contains material from a manuscript in progress which I am co-authoring with Prof Ken Salem.

Abstract

Modern database systems that support ACID transactions, and applications built around these databases, may choose to sacrifice transaction durability for performance when they deem it necessary. While this approach may yield good performance, it has three major downsides. Firstly, users are often not provided information about when and if the issued transactions become durable. Secondly, users cannot know if durable and non-durable transactions see each other's effects. Finally, this approach pushes durability handling outside the scope of the transactional model, making it difficult for applications to reason about correctness and data consistency.

To address these issues, we present the idea of "Eventual Durability" (ED) to provide a principled way for applications to manage transaction durability trade-offs. The ED model extends the traditional transaction model by decoupling a transaction's commit point from its durability point – therefore, allowing applications to control which transactions should be acknowledged at commit point and which ones at their durability point. Furthermore, we redefine serialisability and recoverability under ED to allow applications to ascertain if fast transactions became durable and how they might have interacted with safe ones. With ED, users and applications can know what to expect to lose when there is a failure – thus, bringing back managing durability inside the transaction model.

We implement the ED model in PostgreSQL and evaluate it to understand the model's effect on transaction latency, abort rates and throughput. We show that ED Postgres achieves significant latency improvements even while ensuring the guarantees provided by the model. Since a transaction's resources are released earlier in ED Postgres, we expected to see lower abort rates and higher throughput. Consequently, we observed that ED Postgres provides an average of 91.25% – 93% reduction in abort rates under a contentious workload and an average of 75% increase in throughput compared to baseline Postgres. We also run the TPC-C benchmark against ED Postgres and discuss the findings. Lastly, we discuss how ED Postgres can be used in realistic settings to obtain latency benefits, throughput improvements, reduced abort rates, and fresher reads.

Acknowledgements

I want to express my heartfelt gratitude to my supervisor Prof. Ken Salem for his exceptional kindness, invaluable support, and guidance these past two years. It has been an honour and a privilege to work with him. He has been, and will continue to be, a source of inspiration for me to strive towards.

Secondly, I would like to thank Profs. Khuzaima Daudjee and Sujaya Maiyya for reading my thesis and providing valuable feedback and suggestions. I would also like to thank them for the questions and engaging conversations during the presentation.

These two years of grad school were challenging for a lot of reasons, and I could not have done it without my friends. They have supported me in more ways than I can recount here. I will definitely miss everything I did with them – impromptu plans, weekends, trips, playing badminton, table tennis, going for runs, and so much more.

I also want to sincerely thank the University of Waterloo for never leaving me wanting for anything. It has all the resources and facilities I could have asked for and more. The professors, the people, the events, and everything else here have all helped shape me into a better person.

Of course, I would be remiss not to thank my parents for their love, support and the countless sacrifices they have made for my sake.

Dedication

To my parents, who have made so many sacrifices over the years.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

A transaction's commit has been tied to its durability ever since the idea of ACID (Atomicity, Consistency, Isolation, and Durability) transactions was introduced more than fifty years ago [32] [26] [27]. According to the transaction model initially presented by Gray and Reuter [29], when a transaction *commits*, its effects are 'durable'. Indeed, committing a transaction may mean many things nowadays, but we will stick to the original definition for this discussion. Durability, in turn, means that the effects of the transaction can survive some failures.

A system might choose to protect itself against multiple kinds of failures. For instance, in a single-server, in-memory database, preventing data loss during power failures would mean writing a transaction's effects durably to disk. If the server irrecoverably fails, data written to disk is also lost. To protect against irrecoverable node failures, a standby server or a replica may be added to this single-server database setup. Now, making the transaction 'durable' would mean that its effects have to be written to the disk of the primary and to the memory or disk of one or more standby servers to survive node failures. In the case of a distributed database supporting distributed transactions, the transaction's effects may need to be durably recorded on multiple servers so that the transaction may survive the failure of one or more servers. Since disk writes/access times and network latencies are orders of magnitude higher than cache and memory access times [47] as shown in Table 1.1, it is easy to see how quickly the durability costs of transactions can add up when transactions need to survive progressively more complex and different types of failures.

Consider a concrete example using PostgreSQL that illustrates the increasing costs of transaction durability just described. Postgres allows synchronous standbys to be added to the primary node to enable failovers. The greater the number of synchronous standbys and the further away they are from the primary server, the higher the cost of making a transaction durable. We will show this cost of durability with a simple experiment that uses Postgres configured with different levels of durability. One primary Postgres node and one synchronous standby are set up, and we measure the latency of simple transactions, each of which performs one single-row update. We start the experiment with durability turned off, i.e., a transaction gets acknowledged even before its effects are written to disk.

Operation	Latency
CPU Instruction Execution	1 ns
fetch from L1 cache memory	0.5 ns
branch misprediction	5 ns
fetch from L2 cache memory	7 ns
Mutex lock/unlock	25 ns
fetch from main memory	100 ns
send 2K bytes over 1Gbps network	20,000 ns
read 1MB sequentially from memory	250,000 ns
fetch from new disk location (seek)	8,000,000 ns
read 1MB sequentially from disk	20,000,000 ns
send packet US to Europe and back	150 milliseconds = 150,000,000 ns

Table 1.1: Approximate times of various operations in a distributed system, from [47]

Then, we step up the durability level by forcing the transaction to the local disk before it is acknowledged. Following this, we increase the durability level even further by placing the synchronous standby progressively further away from the primary server – i.e., we start by having the standby in the same region and the same availability zone as the primary, and we go up to when the standby is in a distant region from the primary. The results are shown in Fig. 1.1.

We used `ping`[1] to measure the TCP round-trip time between the test client and the primary Postgres server. The results showed that the average round-trip latency is 0.162 ms. Based on this finding, and using the latency from the "no-durability" option (`us-east-1a, sync=off` in Fig. 1.1), we can estimate that the average per-transaction processing time on the server-side during the experiment for this specific transaction is 0.025 ms. The remaining latency experienced by the client can be attributed to network and disk latency. Using this information as a starting point, we can estimate the combined overhead of network and disk costs in subsequent configurations.

From Fig. 1.1, we see that the setting `sync=local` adds 0.79 ms of latency to the `sync=off` latency. This shows that making a transaction durable accounts for about 97% of server-side latency. On the client side, latency increases by about 522%. When the durability level is increased to remote flush, the overhead is almost 99% of the total transaction latency. In the final setting, where the synchronous standby is in a distant region, almost all of the transaction latency is due to network and disk overhead, which comes up to almost 100% of the total latency.

Therefore, making a transaction durable is expensive, and making transactions tolerant to different types of failures incurs increasingly higher costs. If we were to consider

[1] Using 64 byte ICMP packets

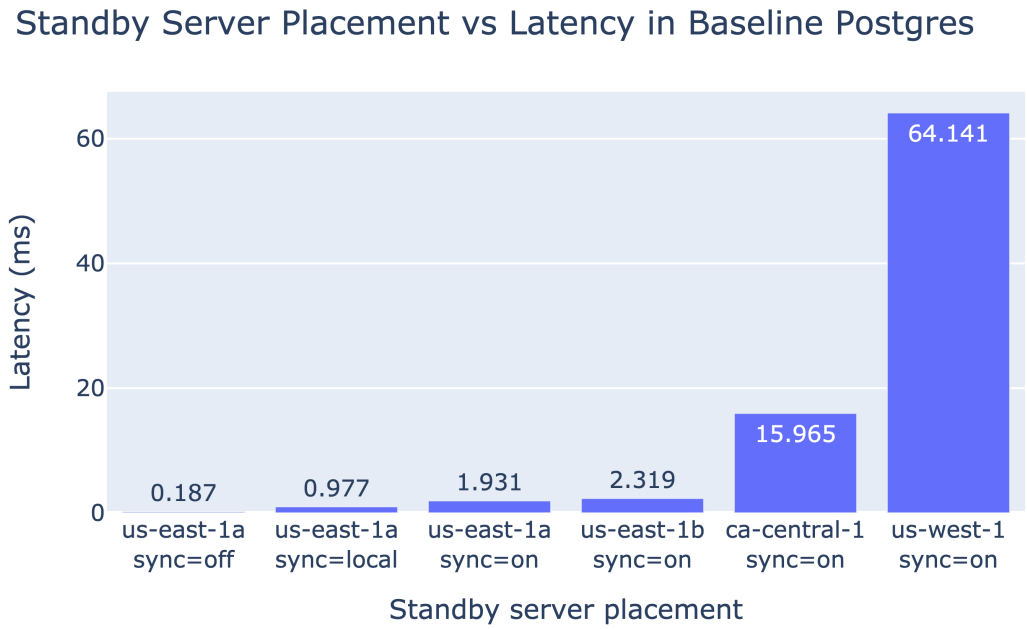


Figure 1.1: The Cost of Durability

this experimental setup as being representative of a multi-region, multi-standby Postgres configuration where there is one synchronous standby per data centre and possibly multiple asynchronous standbys, we can interpret the increasing latencies as rising costs of durability to make a transaction multi-data-centre fault-tolerant. This simple demonstration shows how quickly durability costs can add up as the cluster is configured to handle increasingly complex failures.

However, the manner in which systems have made durability tradeoffs over the years has not been particularly friendly to application developers. Since the traditional transaction model requires that a transaction be durable at commit time, databases cannot provide strictly-ACID transactions and achieve good performance at the same time. Hence, systems have relaxed this guarantee to achieve better performance, but, in doing so, they have pushed durability-handling outside the transaction model. With this approach, developers have the freedom to issue *synchronous* and *asynchronous* transactions, but there are still many uncertainties that they have to deal with while writing applications.

A common ad-hoc approach for avoiding durability costs is simply committing transactions without making them durable. E.g., using `synchronous_commit=off` in Postgres or using asynchronous commits in other systems. In systems using such approaches, of course, non-durable transactions can be lost on failures. Another problem is that applications are unaware of when and if the transactions become durable. If they try to read their own writes and do not see them, they can infer that a failure must have wiped out the in-memory, non-durable writes. However, if they see the writes, they cannot know if the records visible to them are durable or volatile. Applications have to work with this

uncertainty and trust that the database system will eventually make those records durable. However, suppose their subsequent decisions required the durability of those records, and they were lost due to a failure. In that case, applications have to untangle themselves from this mess without much help from the database system. We discuss more examples and case studies of ad-hoc durability handling in related work (Chapter 5).

The second issue is with the visibility of transactions. When both *synchronous* and *asynchronous* transactions are active in the system, applications cannot be sure whether they saw each other's effects. For example, *asynchronous* transactions may not be able to see the effects of *synchronous* transactions until they are durable. There is a secondary implication of this visibility issue as well. Because the *synchronous* transactions hold on to locks and resources for the entire duration of their durability, they may cause higher aborts and, thus, lower throughput under contentious workloads.

Finally, because of this ad-hoc approach, database systems may silently lose transactions, and applications would be oblivious to it until they attempt to read the database state. Furthermore, this can lead to a non-repeatable-read-like situation where an application sees that a record has changed before and after a failure despite the updating transaction having *committed*. Not only does this violate the traditional transactional model, but it also leads to uncertainties that application developers have to work around. Therefore, clearly, there is a need to address these issues formally.

1.1 Summary of Contributions

To deal with durability costs in a principled way and to provide a formal framework to manage these tradeoffs, we extend the traditional transaction model to introduce Eventual Durability in this thesis.

- Chapter 2 begins by providing a brief background on serialisability and recoverability. It then introduces the idea of Eventual Durability (ED) and compares it to the traditional transactional model.
- For a system to support durability tradeoffs, serialisability and recoverability are important transactional properties to guarantee. Therefore, we redefine serialisability and recoverability under ED by building on top of the classical model. We further illustrate how they can be leveraged to eliminate uncertainties described earlier.
- Chapter 3 discusses in detail how the ED model was implemented in Postgres. We identify the missing pieces and describe the changes made to implement these pieces in the transaction manager. We argue for the correctness of these changes and show that they uphold ED guarantees.
- Finally, Chapter 4 presents experimental results of ED Postgres and shows how ED reduces latency, contention, improves throughput and enables fresher reads.

Chapter 2

Eventual Durability

2.1 Background

Before presenting the Eventual Durability (ED) model, we will first provide some brief background on serialisability and recoverability.

A database operation involves either reading or writing data items in the database. Read operations are denoted as $r[x]$, which indicates that the operation read x. A write operation is denoted as $w[x]$ indicating that the operation wrote to the value x. A transaction is a collection of one or more database operations that forms some logical unit of work. E.g., a transaction T could have:

$$T = r[a]\ w[a]\ r[b]\ w[b]\ c$$

This denotes that the transaction T first read the value a and then updated it, followed by reading the value b and updating it. In this notation, a sequence of operations in a transaction is always followed by one of two operations – c for *commit* or a for *abort*. Now, consider two transactions T_1 and T_2:

$$T_1 = r_1[a]\ w_1[a]\ r_1[b]\ w_1[b]\ c_1$$
$$T_2 = r_2[a]\ w_2[b]\ c_2$$

If T_1 and T_2 try to execute simultaneously, the database system has to make a decision on how to interleave their operations. A history H is some interleaving of concurrent database operations. E.g.:

$$H_1 = r_1[a]\ r_2[a]\ w_2[b]\ c_2\ w_1[a]\ r_1[b]\ w_1[b]\ c_1$$

Now consider this alternate history H_2:

$$H_2 = r_1[a]\ w_1[a]\ r_2[a]\ w_2[b]\ r_1[b]\ w_1[b]\ c_1\ c_2$$

While it may appear that there are only slight differences between H_1 and H_2, the results of a and b at the end of H_1 and H_2 are different. With this brief background, we now proceed to provide the classical definitions of serialisability and recoverability, followed by their renewed definitions under ED.

2.1.1 Serialisability

Bernstein et al.[11] define that a history "is serialisable if it produces the same output and has the same effect on the database as some serial execution of the same transactions". In our earlier example, H_1 is serialisable since its output will be equivalent to running T_2 first and then T_1. However, H_2 is not serialisable because its output will not be equivalent to any serial execution of T_1 and T_2 – i.e., running $T_1; T_2$ or running $T_2; T_1$ both give different outputs from running H_2. While there is a lot of complexity and nuance in serialisability, its realisation and its implications on systems, this simple definition should be sufficient to extend the classical idea.

2.1.2 Recoverability

Once again, we turn to Bernstein et al. [11] to define recoverability. A database system's recovery mechanism must ensure that the database state has all the effects of committed transactions but none of the effects of uncommitted or aborted transactions. In order to "undo" a transaction's effects, the database system has to restore the state on disk, as well as abort affected transactions. In the traditional transactional model, the database system guarantees that if a transaction is *committed*, it will not subsequently be aborted. Therefore, before a transaction commits, the system must ensure that all the transactions that the committing transaction, say T, has read from will not abort – or, in other words, have themselves committed. So, we say that a history H is recoverable if, for every transaction T in H that commits, T's *commit* follows the *commit* of every transaction from which T read.

2.2 The Eventual Durability Model

Traditional ways to deal with high durability costs have involved ad-hoc mechanisms that may or may not retain the transactional ACID guarantees. Many database systems have no notion of a transaction, do not reliably persist writes, and acknowledge requests while

writes have been accepted transiently in memory [39] [53][12]. Such ad-hoc approaches pose a legitimate risk to data, especially when developers do not have any control over which transactions should be acknowledged quickly and which ones should be acknowledged only after they are fully durable. Applications using such systems may have a limited notion of failure models, and, therefore, issuing write requests becomes a roll of dice for its users [51]. Systems like PostgreSQL provide synchronous and asynchronous commit options for transactions. However, users do not have any way of knowing when these asynchronous transactions become durable and how these transaction types interact with each other when run concurrently.

In this work, we introduce the idea of Eventual Durability to provide users with a principled foundation to manage durability trade-offs. The core idea of eventual durability is that transactions commit first and then become durable over time. We decouple a transaction's commit point from its durability guarantee while ensuring that execution histories are serialisable and optionally recoverable if the user desires. We also provide a realisation of the eventual durability model and set down its properties and guarantees. This model paves the way for applications to have fine-grained control over which transactions should commit quickly and which ones safely. Because of the recoverability property offered by ED, users can know what to expect during crashes.

Let us now take a closer look at the transaction processing model initially presented by Gray and Reuter [29]. Fig. 2.1 shows, in a simplified sense, the transaction lifecycle in traditional database systems. A transaction is initially in the *active* or *pending* state when it is still executing some read or write operations on the database. Once done, the user issues a `COMMIT`. At this point, the database system may perform some commit-time processing, and the state of the transaction is switched from either *active* or *pending* to *committed*. When the transaction commits, in addition to being guaranteed atomicity and serialisability, it is also guaranteed durability. In the traditional model, a transaction's *commit* is a promise by the database system that the effects of the transaction will not be lost from the system.

With the eventual durability model, we untether a transaction's ACID semantics from its durability guarantee. A transaction is serialised, atomically executed, and its changes are made visible to other transactions, but its writes may not be fully durable at commit time. This model is illustrated in Fig. 2.2. All transactions under the ED model commit first but become durable later. An important distinction from the traditional transaction processing model is that under ED, a committed transaction can be "lost" if it fails to achieve full durability.

There is more than one way for database systems to expose eventual durability to database applications. In this work, we expose eventual durability by allowing applications to choose between two types of transactions: "fast" and "safe". "Safe" transactions behave like traditional database transactions – they are not acknowledged until they have become durable. Fast transactions, on the other hand, are acknowledged as soon as they commit. All transactions – whether fast or safe – become visible after committing, just as in the traditional model.

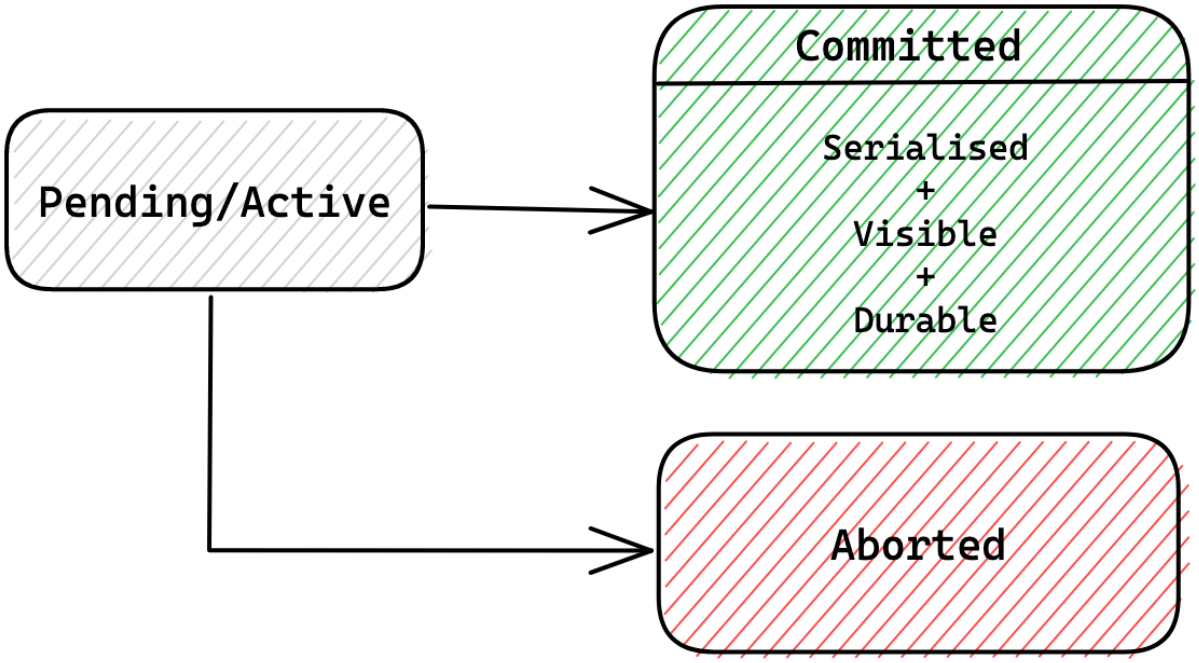


Figure 2.1: Traditional Commit Processing in ACID databases

Because a fast transaction's commit will be acknowledged before the transaction is durable, the application must be prepared for the possibility that a fast transaction will be *lost* – even though its commit has been acknowledged – in the event that it fails to become durable. Although fast transactions are not guaranteed to be durable when acknowledged, they retain all other ACID properties – i.e., they remain isolated and fully serialised with respect to other transactions. Under the ED model, applications can now choose to issue critical transactions in the "safe" mode and non-critical ones in the "fast" mode.

Furthermore, an important property that we guarantee, in addition to ED serialisability, is ED recoverability. We will formally define both these properties shortly, but in summary, ED recoverability states that a transaction cannot become durable until all of its dependencies have become fully durable. This property ensures that if a safe transaction, say T_s, happens to read the effects of a prior fast transaction, say T_f, which is not durable yet, T_s will not be acknowledged until both T_f and T_s are durable. This feature of interest in the ED model sets it apart from ad-hoc techniques that manage durability. In systems that accept commit requests and make transactions durable asynchronously, it is almost impossible for users to tell whether the data they are reading is durable or volatile. With ED, however, users can always be assured that if their safe transaction has read non-durable effects, they will all be fully durable when the database system acknowledges the safe transaction.

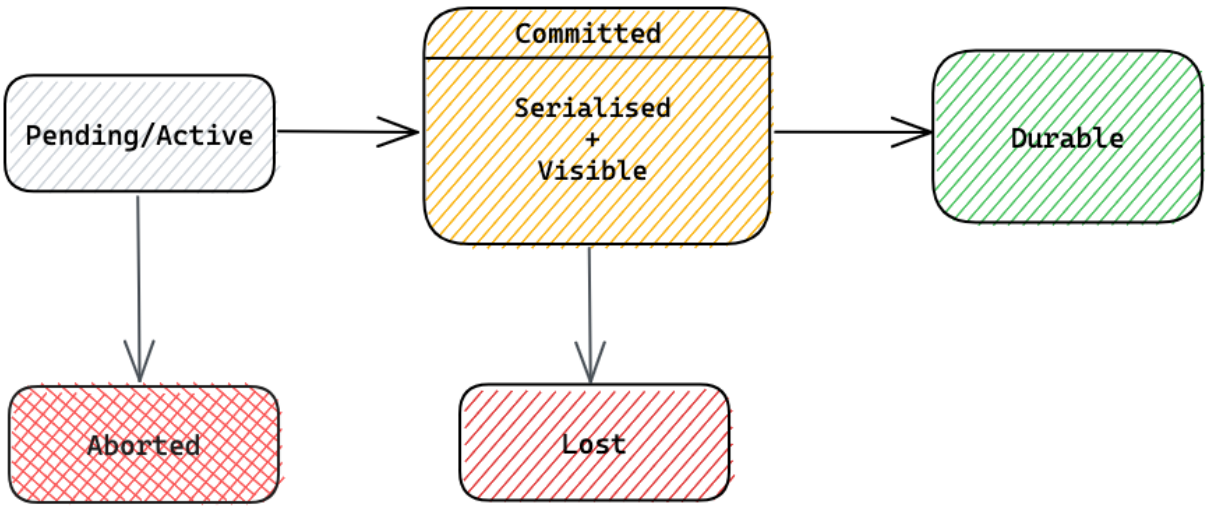


Figure 2.2: Proposed changes to the commit processing model in ACID Databases

2.2.1 Summary of Eventual Durability Properties and Guarantees

We will now summarise the set of properties and guarantees the ED model provides when realised using fast and safe transactions.

1. A transaction in a database system that implements ED can be tagged *fast* or *safe*.
2. All transactions in an ED system can be in one of five states at any given point – *active, committed, aborted, durable* or *failed*. The progression of states of a transaction can be one of the following:

$$Active \rightarrow Committed \rightarrow Durable$$
$$Active \rightarrow Committed \rightarrow Failed$$
$$Active \rightarrow Aborted$$

3. Transactions in the *committed* and *durable* states are visible, while others are not. Therefore, it follows that transactions that were once visible when they were in the *committed* state cease to be visible if they move into the *failed* state.
4. Fast transactions are acknowledged as soon as they commit. These are similar to *asynchronous* transactions offered by some databases but with key differences. Say, a fast transaction T_j reads from another fast transaction T_i and proceeds to commit. T_i later fails to become durable and is moved to the *failed* state. In such a situation, T_j can never become durable and will be moved to the *failed* state by the database. More generally, a transaction can become durable only if all of its dependencies have themselves become durable – should the ED implementation choose to guarantee ED recoverability.

5. A safe transaction is acknowledged only after it is fully durable in the database system. Sub-properties of safe transactions are listed below:

   (a) As soon as a safe transaction commits, it is immediately visible to other concurrently running transactions in the database system, even though the effects of the safe transaction are not durable. Depending on whether it is fast or safe, the reading transaction can decide to wait or return quickly.

   (b) Assuming the ED implementation guarantees ED recoverability, a read-only and safe transaction is acknowledged only after all its read-dependency transactions have become durable.

2.2.2 Serialisability under Eventual Durability

We will now formally extend the classical definitions of serialisability and recoverability under eventual durability. The eventual durability model introduces two new states for a transaction, namely, *durable* and *failed*. This is in addition to the allowed states in the traditional transactional model – *active, committed* and *aborted*. When dealing with serialisability under ED, it is important to consider that even if a transaction is committed, it can still fail if it does not become durable. We account for this and extend Bernstein et al.'s [11] definition of serialisability herein.

In the classical model, a transaction is a set of database read-and-write operations that form some logical unit of work and is followed by either a c (commit) or an a (abort). For eventual durability, we introduce two new transaction history events, f (failure) and d (durability). In a history, a transaction's commit (c) may be followed by either d or f, but not both. d indicates that the transaction has become durable - we refer to it as the *durability point*. f, on the other hand, indicates that the transaction has failed. State transition events (a, c, d, f) mark the transaction's state progression, as described in Property 2 above.

A history over a set of transactions is an interleaving of the operations of those transactions. A committed projection $C(H)$ of a history H is a projection of all database operations and commit events from H, but not containing operations of *failed* or *aborted* transactions, and not containing any d events. Note that committed projections ($C(H)$) of ED histories are *classical* histories since they will never contain either of the two new events (d and f) that were introduced for ED histories.

Consider two ED transactions, T_1 and T_2 without any c, a, d or f events:

$$T_1 = r_1[x]\ w_1[y]$$
$$T_2 = r_2[y]\ w_2[z]$$

and two alternate histories H_a and H_b involving T_1 and T_2:

$$H_a = r_1[x_0]\ w_1[y]\ c_1\ r_2[y_1]\ w_2[z]\ c_2\ f_1\ d_2$$
$$H_b = r_1[x_0]\ w_1[y]\ c_1\ r_2[y_1]\ w_2[z]\ c_2\ d_1\ d_2$$

The subscript in each read operation indicates the version of the value being read. For example, $r_2[y_1]$ means that T_2 read the value y that was previously written by T_1. In H_a, T_1 fails after reading x and writing y; and T_2 becomes durable after reading y and writing to z. In H_b, both T_1 and T_2 commit and become durable. The committed projections of both H_a and H_b are as follows:

$$C(H_a) = r_2[y_1]\ w_2[z]\ c_2$$
$$C(H_b) = r_1[x_0]\ w_1[y]\ c_1\ r_2[y_1]\ w_2[z]\ c_2$$

With this, we can define ED serialisability as:

Definition 2.2.1 (ED Serialisability) *An eventually durable history H is serialisable if its committed projection, $C(H)$, is serialisable in the classical sense.*

In our example above, $C(H_a)$ is **not** serialisable, and $C(H_b)$ is serialisable. $C(H_a)$ is not serialisable because its output is not equivalent to the execution of T_2 from the initial state of the database. Had T_2 read from y_0, $C(H_a)$ would have been serialisable. On the other hand, $C(H_b)$ is serialisable because it is equivalent to running $T_1; T_2$.

Definition 2.2.2 (ED Recoverability) *An ED history is recoverable if* both *of the following conditions hold:*

1. *If T_2 reads from T_1 in H, then T_2 does not commit before T_1 commits.*
2. *If T_2 reads from T_1 in H, then T_2 does not become durable before T_1 becomes durable. That is, if $d_2 \in H$, then $d_1 \in H$ and $d_1 < d_2$.*

The first condition matches the classical definition of recoverability. It ensures that committed transactions, which are visible to the application, do not depend on transactions that have not committed yet, since such transactions might abort.

The second condition is specific to ED histories. Consider the following history, in which T_2 has read from T_1, like this:

$$H_c = w_1[x]\ r_2[x_1]\ w_2[x_2]\ c_1\ c_2$$

H_c is ED recoverable according to our definition, yet it seems "risky" because committed ED transactions can still fail. What if T_1 were to fail? The resulting history would no longer be ED serialisable since T_2 has seen the effects of T_1. This seems like exactly the kind of situation that recoverable histories should avoid! We define this history to be ED recoverable because that database system actually has an "escape" from this situation: if T_1 fails, it can force T_2 to fail as well. That is, it can fail and erase not only T_1 but anything that depends on T_1. However, this escape is available only if T_2 is not durable since durable transactions do not fail. This motivates the second condition in the ED recoverability definition: if T_2 depends on T_1, we cannot allow T_2 to become durable until we are sure that T_1 will not fail, i.e., until T_1 is durable. Note that this does not mean that T_2 *will* become durable, but simply means that for T_2 to become durable, T_1 must be durable.

2.2.3 Read-Only Transactions

We will conclude this section with a brief discussion of read-only transactions. Like all ED transactions, read-only ED transactions eventually fail or become durable after committing. Since read-only transactions make no changes to the database, this may seem like a distinction without a difference. However, the distinction is actually important.

If the database system guarantees ED serialisability, applications are guaranteed that a committed read-only transaction has seen a serialisable view of the database. However, committing a read-only transaction does *not* guarantee that the data it has read is durable. The read-only transaction may have read from earlier transactions that are committed but are not yet durable.

This is where the read-only transaction's durability point comes in. As long as the execution is recoverable, the second condition in Definition 2.2.2 demands that the read-only transaction's durability point occurs only after the data it has read is durable. Thus, the durability point serves to indicate when the reads are safe. Conversely, if the read-only transaction has read from a transaction that fails after commit, then the read-only transaction must also fail after its commit.

By exposing these events to the applications, the database system can provide them with valuable information for managing failure risks. In the subsequent chapters, we will discuss various ways in which the database system can do this.

Chapter 3

Eventual Durability in PostgreSQL

We will now describe the implementation of eventual durability in Postgres. We begin by looking at the general architecture of Postgres and then its streaming replication feature. Finally, we deep dive into the relevant modules needed for explaining the ED implementation. We then explain the changes made to Postgres and justify why they were made. Finally, we provide an argument for the correctness of the ED implementation in Postgres.

3.1 General Architecture of PostgreSQL

PostgreSQL is an open-source, object-relational database management system that has been in use and active development since 1987. It is one of the most popular relational database systems with rich documentation and community support. We chose Postgres to implement ED because it is already closely aligned with the model and needed fairly non-complex changes for a working implementation of ED. Postgres supports tunable `synchronous_commit` modes that were leveraged for our implementation. In this subsection, we will briefly look at the system architecture of Postgres to lay the groundwork for presenting the ED implementation later on.

Fig. 3.1 shows the system architecture of Postgres. Postgres uses a process-per-connection model and is organised as a multi-process system, with each process performing a specific set of tasks. For example, the back end process (also called the `postgres` process) handles queries from a client on a database. The background writer (`bgwriter`) and the `checkpointer` processes flush dirty pages from the buffer cache to disk, and the `autovacuum` process removes dead tuples from a table and compacts table size. The `postmaster` process is an orchestrator that spawns and kills other `postgres` processes as needed. It is responsible for initially accepting a connection request from a client and assigning a back end to handle the connection. There are many more background processes of Postgres that perform essential functions but are not relevant to the ED implementation.

Postgres maintains a buffer cache that contains copies of data on disk as pages in memory. Frequently accessed pages are loaded from the disk and kept in the cache, while

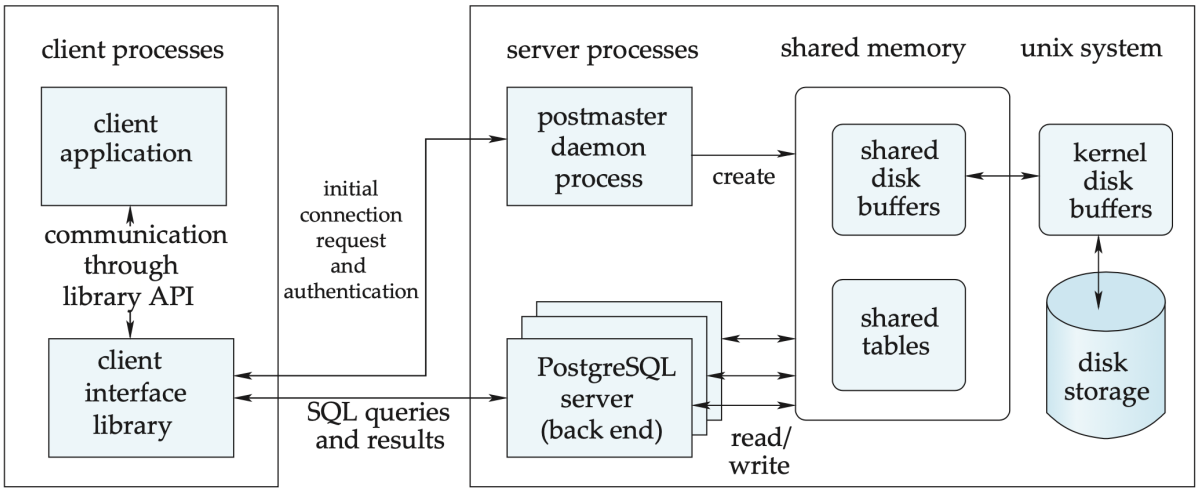


Figure 3.1: PostgreSQL System Architecture (from [52])

unused pages are routinely evicted. The database system uses write-ahead logging to record changes and provide a `REDO` mechanism for crash recovery. A background process frequently flushes dirty pages in the buffer cache to the kernel's disk buffers, eventually making their way to durable storage. It is worth mentioning here that Postgres uses shared memory to coordinate the functioning of various background tasks and is meant to be run on a single compute instance. Nevertheless, Postgres allows for a high-availability configuration involving two or more Postgres instances across different machines through its streaming replication mechanism, which we will describe in a forthcoming section.

3.2 Index of terms

We will now briefly define some acronyms and terms used in this chapter.

- `XLOG`: The `XLOG`, also called the Transaction Log or the Write-Ahead Log(WAL), is a record of changes made to the state of the database. We will interchangeably use `XLOG` and `WAL` in this chapter.
- `CLOG`: The Commit Log or `CLOG` stores the status of transactions. A transaction can be in one of the following states at any given time – `IN_PROGRESS`, `COMMITTED`, `ABORTED` or `SUB_COMMITTED`.
- `XID`: Also called the transaction ID and used interchangeably, it is a 32-bit integer that uniquely identifies a transaction and wraps around every 4 billion transactions.
- `LSN`: A *Log Sequence Number* (LSN) uniquely indicates the position of an `XLOG` record both in the in-memory and the on-disk copy of the `XLOG`. They are monotonically increasing numbers assigned to `XLOG` records when written to the in-memory `XLOG`.

- `WALSender`: The WALsender is a process on the primary server that ships new changes in the `XLOG` to standby servers when physical replication is enabled.
- `WALReceiver`: A WALreceiver is a process running on each standby server that connects to the primary server and receives `XLOG` entries from the WALsender when physical replication is enabled.
- `WAL_writer`: The `WAL_writer` flushes WAL entries from the in-memory `XLOG` cache to disk in asynchronous commit mode.

3.3 Path of a write query in PostgreSQL

Let us examine what happens when a user submits a transaction `T` that changes the state of the database. Assume the transaction has two operations – an update query followed by a `commit`. We will primarily look at the sequence of steps executed when the `commit` is issued. Fig. 3.2 illustrates this workflow.

We assume that the update query of the transaction received by the Postgres back end has made its way through the query processing engine and concurrency control and is waiting to be applied to the corresponding tablespaces. The very first thing the back end does when it receives the update query of the transaction is to extend the `CLOG` and mark the current transaction as `IN_PROGRESS`. Next, an exclusive lock on the buffer containing the data page in the buffer cache is obtained, and its usage count is incremented. The changes are made on the buffer (arrow #2 in Fig.3.2), and a corresponding `XLOG` record is generated. The `XLOG` record is inserted into the in-memory `XLOG` cache (arrow #3 in Fig. 3.2), which is stored as a ring buffer in memory. Once inserted, the record gets assigned an LSN, which is then updated on the corresponding page in the buffer cache. At this point, the update query is acknowledged by the Postgres back end and sent to the client.

After receiving confirmation that the update query was executed, the client proceeds to issue a `commit` for the transaction. The `commit` record follows a similar path as the update query did. The `commit` record is inserted into the in-memory `XLOG`. The commit request is not yet acknowledged to the client. Since, by default, `synchronous_commit` is set to *on*, the changes recorded in the in-memory `XLOG` cache are flushed to the on-disk copy of the `XLOG` (also called the WAL segment files) before the `commit` request is acknowledged to the user (arrow #5 in Fig. 3.2). It is worthwhile to note here that since LSNs are monotonically increasing, a flush of the `commit` record from the in-memory `XLOG` to the WAL segment files ensures that the `XLOG` records generated by the update query are also flushed to disk. Once the `commit` record is flushed to disk, the transaction status in the `CLOG` is updated to `COMMITTED`.

The transaction's changes are now recorded in the buffer cache, the in-memory `XLOG` cache and on disk in the `WAL` segment files. The `checkpointer` and the `background writer` processes of Postgres are responsible for eventually flushing dirty pages in the buffer cache to disk.

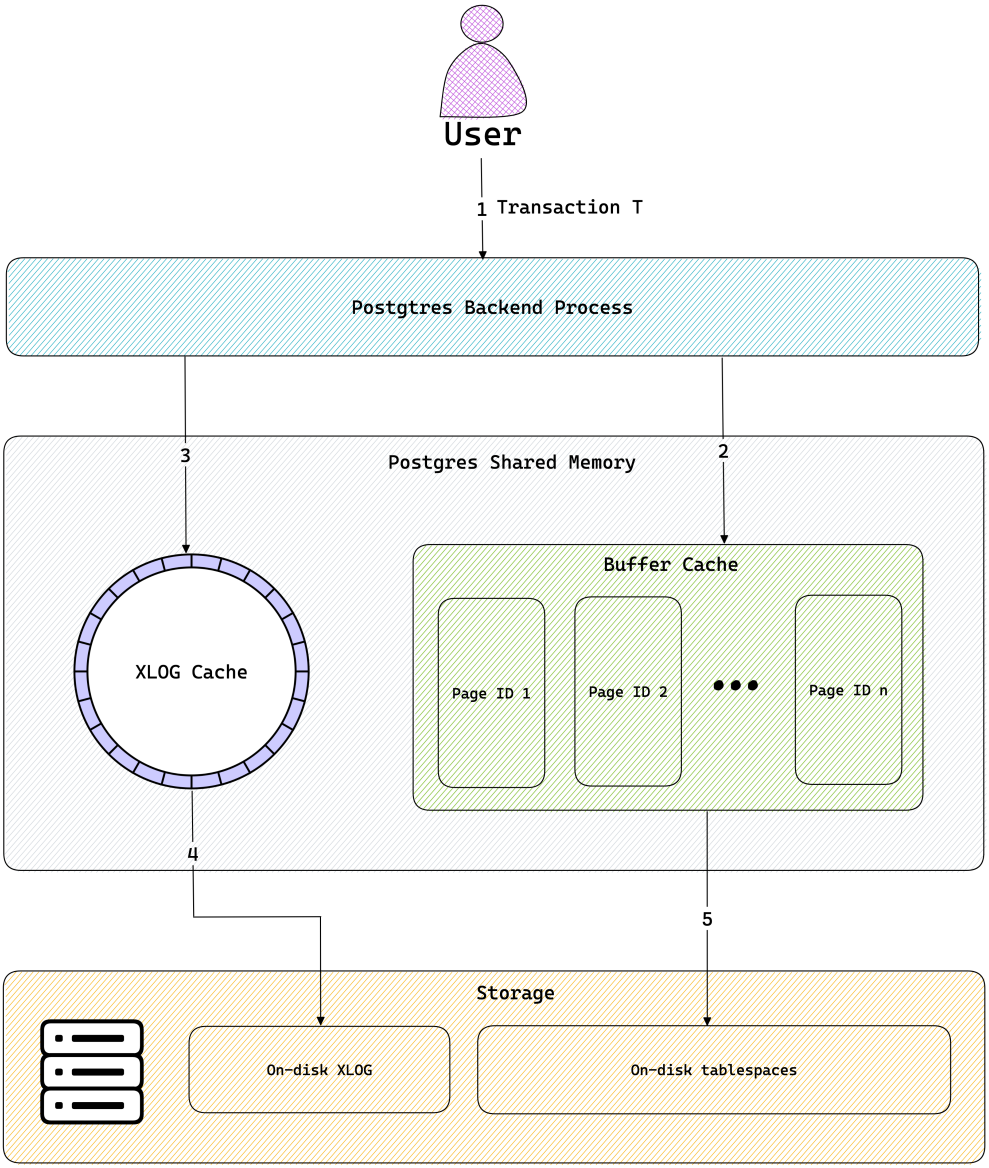


Figure 3.2: Path of a Transaction in PostgreSQL

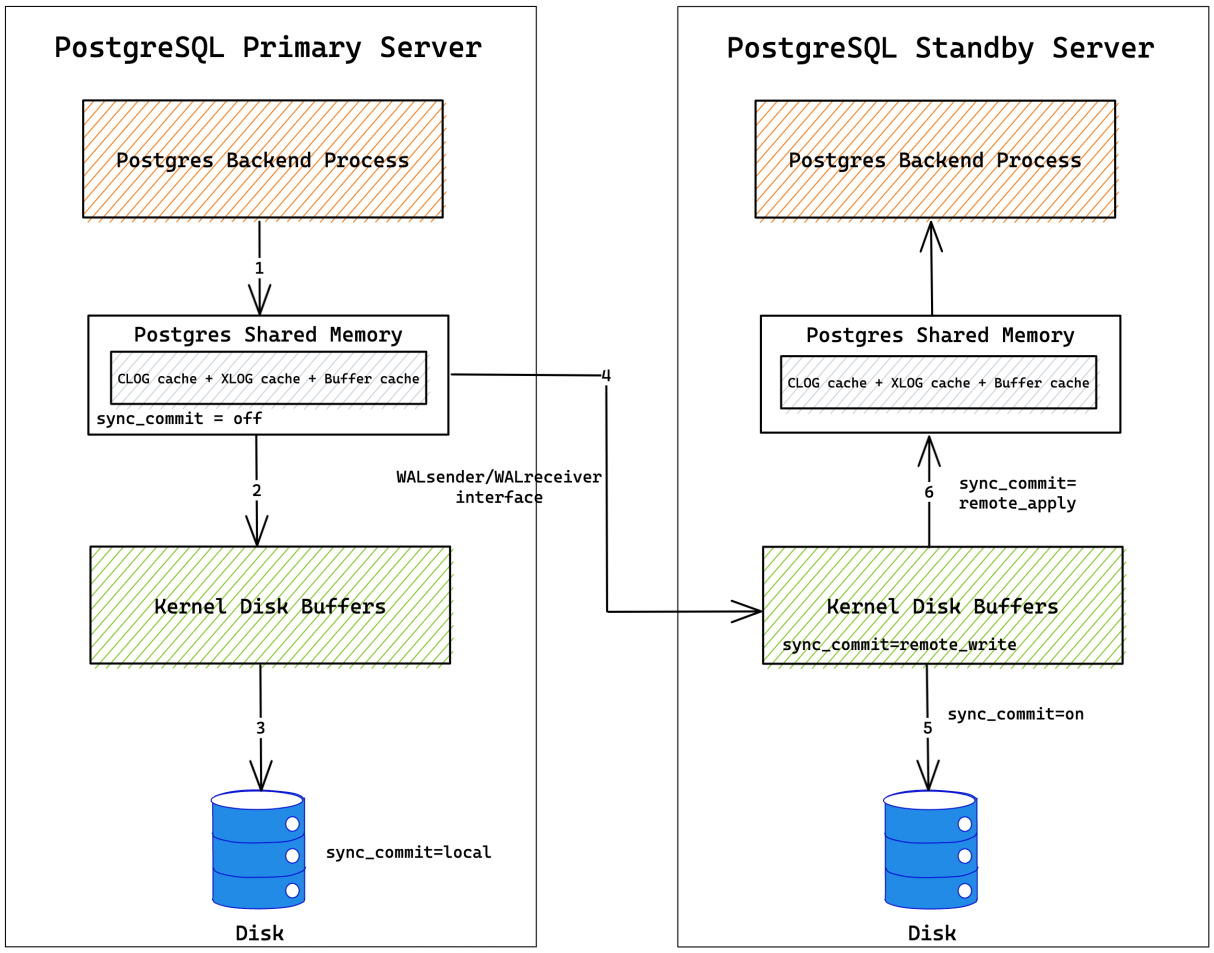


Figure 3.3: Streaming Replication in PostgreSQL

3.4 Streaming Replication in PostgreSQL

As briefly mentioned previously, Postgres supports streaming replication in synchronous or asynchronous mode to secondary servers. This feature, along with other techniques, may be used to set up a highly-available/fault-tolerant database cluster. Streaming replication may also be used to set up read load-balancing to reduce strain on the primary server. In the most straightforward configuration that uses streaming replication, there could be just one primary server and one standby server. However, Postgres also allows for complex structures with multiple synchronous standbys, cascading replication, and so on. The ED implementation in Postgres is integrated with replication, and therefore we will now look at streaming replication in PostgreSQL.

Fig. 3.3 illustrates the steps involved in the streaming replication process. The numbers on the arrows indicate the sequence in which the writes happen. In Section 3.3, we looked at how a transaction that changes the state of the database makes its way from the client to durable storage of the Postgres machine. This section will look at what happens to those

changes if streaming replication is enabled. In other words, how the changes recorded on the XLOG of the primary server get propagated to configured standbys.

As soon as a set of in-memory XLOG changes is flushed to the WAL segment files on disk, Postgres wakes up all WALsender processes to inform them that new data in the XLOG needs to be shipped out to the standby servers. Note that there is one WALsender process per standby server configured for streaming replication. The WALsender process is created when a standby server connects to the primary server in "replication" mode. The WALsender process wakes up when its latch is set by an XLOG flush. It consults the in-memory copy of the XLOG and begins sending XLOG records to subscribed WALreceivers. The WALreceivers on the standby servers receive these records and start writing to the kernel's disk cache. These changes are then flushed to the disk, and finally, they get applied to the in-memory shared buffers.

3.5 Commit Configuration

The synchronous_commit option specifies when a transaction's commit can be acknowledged to the client. Even when synchronous replication is not set up, this option changes when clients receive acknowledgements for their transaction commits. The options allowed by Postgres and their implications on transaction commit acknowledgements are described below:

- *off*: When synchronous_commit is set to *off*, commit requests are acknowledged as soon as they are written to the in-memory XLOG ring buffer (shown by the arrow #1 in Fig. 3.3), but before the XLOG records are flushed locally to disk or sent to any of the standby servers (if configured). Essentially, this option reduces transaction latencies by allowing them to be acknowledged much earlier than they would usually have been (since the default option is *on*). However, there is a risk that changes made by the transaction – which have been acknowledged as *committed* – may be lost if the primary server fails.

- *local*: When the option is set to *local*, the Postgres back end waits for the changes submitted by the client to be written to the in-memory XLOG cache and for the XLOG records to be flushed to the local disk (shown by arrow #3 in Fig.3.3). Transactions that use this synchronous_commit option can survive primary node restarts but not their complete failure. When and if the primary node restarts due to an issue, although the in-memory XLOG is lost, it can be recovered from the disk. However, if the primary node were to fail completely, then the transaction's effects would be lost.

- *on*: When there are synchronous standbys defined, the *on* option ensures that the back end process waits for the XLOG changes to be a) written to the in-memory XLOG cache b) flushed to the WAL segment files on disk c) written to the kernel disk cache

on the remote server(s); and d) flushed to the remote server(s)' disk This is shown by the arrow #5 in Fig. 3.3. This is the default option and ensures that transaction effects survive primary node failures.

- *remote_write*: The *remote_write* option is a slightly weaker version of the *on* option. With this `synchronous_commit` level, the back end waits for everything it would have waited for with the *on* option, except for the last step, i.e., flush of `XLOG` records on the remote disks (shown by the arrow #4 in Fig. 3.3). Using this option also ensures that transactions survive primary node failures.

- *remote_apply*: The *remote_apply* option is the most expensive among all the above options. With this option, the Postgres back end waits for everything it would have waited for with the *on* option and, in addition, waits for the `XLOG` changes to be applied to the corresponding pages in the buffer cache (shown by the arrow #6 in Fig. 3.3). Using this option ensures that transactions are visible to readers on standby servers as soon as they are acknowledged on the primary.

3.6 Eventual Durability in PostgreSQL – The Missing Pieces

As a quick recap, the ED model decouples a transaction's commit point from its durability. One way of implementing ED is by using two kinds of transactions – *fast* and *safe*. All transactions commit first and become durable eventually. Fast transactions are acknowledged at commit, and safe transactions are acknowledged at durability. The ED implementation in Postgres enforces ED recoverability as well.

We overload the `synchronous_commit` option to implement fast and safe transactions. To execute a fast transaction, a client uses the `synchronous_commit=off` option, and to execute a safe transaction, the client uses `synchronous_commit=on`. All transactions in ED Postgres become visible as soon as they commit. Fast transactions in ED Postgres are acknowledged as soon as they commit, and safe transactions are acknowledged when they become durable. Since the ED implementation in Postgres enforces recoverability, transactions cannot become durable until all their dependencies have become durable. As safe transactions in ED are acknowledged only after they are fully durable, it means that when they are acknowledged, the safe transaction itself and all of its dependencies must be durable.

Let us now look at how implementing ED in Postgres would change the commit path of transactions. Fig. 3.4 shows the difference between the commit processing of `sync=on`, read-write transactions in baseline Postgres, and the commit processing of safe, read-write transactions in ED Postgres. In baseline Postgres with `synchronous_commit=on`, the transaction becomes visible only when the commit record has been flushed to disk and replicated to synchronous standbys, if any. However, ED Postgres allows transactions

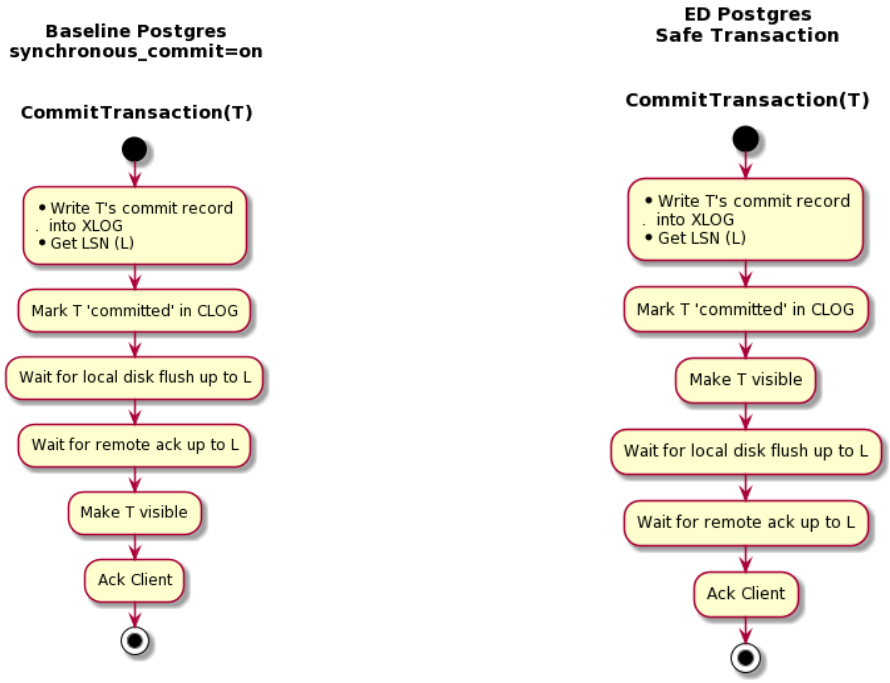


Figure 3.4: Baseline vs ED Postgres back end states during transaction commit

to become visible once their state is updated in the `CLOG`. Therefore, we observe that transactions in unmodified Postgres become visible much later than they are allowed by the ED model. Hence, the first missing piece for implementing ED in Postgres is to make safe transactions visible as soon as they commit.

Fig. 3.5 shows the commit path of an ED fast transaction. The commit record is written to the `XLOG`, and the transaction is marked *committed* in the `CLOG`. Following this, the client's commit request is acknowledged. This commit path is similar to the commit path for `synchronous_commit=off` transactions in baseline Postgres.

For the next missing piece, say that a `sync=on` transaction in unmodified Postgres has both reads and writes to the database. In this case, the property of the ED model that a safe transaction's dependencies should be durable at commit time is honoured because such transactions wait for the `XLOG` flush of their commit record (on local and/or standby servers), thereby ensuring that all transactions serialised before them are durable. However, if the said transaction were to contain only reads, then this property is no longer upheld in unmodified Postgres because read-only transactions do not create `XLOG` entries and hence, do not wait for any flushes to complete. Therefore, a safe read-only transaction's commit may be acknowledged before its dependencies are fully durable. This path is shown in Fig. 3.6a. To rectify this and to provide a full realisation of the safe transaction concept in Postgres, we need to make safe, read-only transactions wait for their dependencies to become fully durable before they return results to the client. The path for safe, read-only transactions in ED Postgres is shown in Fig. 3.6b.

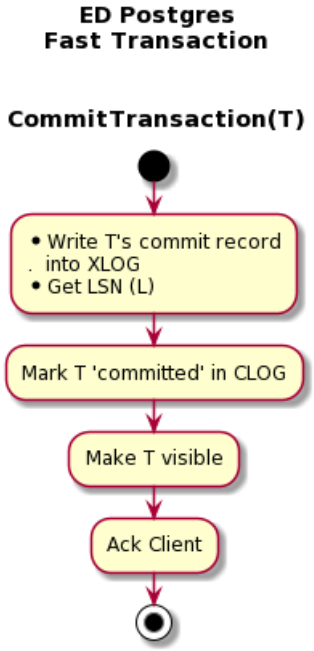


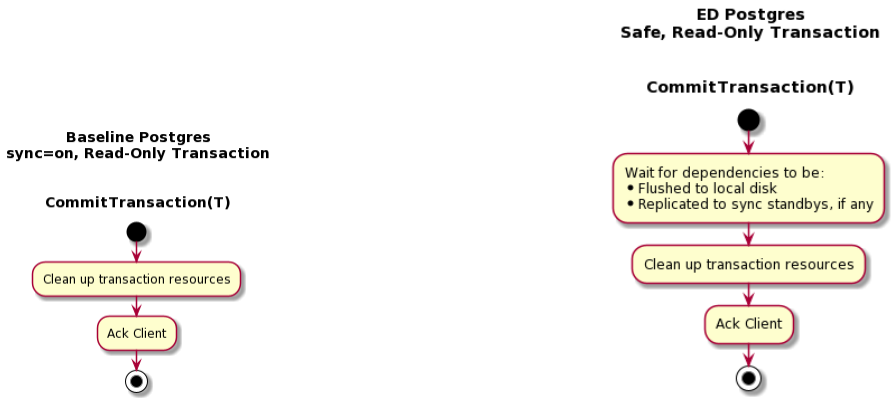
Figure 3.5: ED Postgres back end state for fast transaction commits

3.7 Implementing Early Visibility of Transactions

In baseline Postgres, `sync=on` transactions are not visible until they are durable. However, in the ED model, all transactions become visible as soon as they commit. This section will look at how we made safe transactions in ED Postgres visible early – right after they commit – without waiting for durability. Safe transactions are still acknowledged after durability but become visible as soon as they commit.

First, we will examine the snapshotting and transaction visibility mechanism in unmodified Postgres, and then we will explore the modifications we have introduced and their impact on transaction visibility.

Every Postgres transaction gets a *snapshot* of the database when the transaction begins. This snapshot contains a range of `XIDs` that the transaction can use while executing to decide whether a tuple is visible to it or not. Consider Fig. 3.7, which illustrates the snapshotting mechanism in unmodified Postgres. It depicts four transactions of varying durations that start and end and different times. Say, a snapshot was taken at time unit 5. At this point, `XIDs 700` and `705` are still running, while `XID 702` has committed. `XID 707` has not started yet. In unmodified Postgres, commit implies durability, and therefore `XID 702` is also durable. Postgres includes only committed transactions in a snapshot that finished before the snapshotting process started. Therefore, only `XID 702` is included in the snapshot. So, if a transaction were to use a snapshot taken at time unit 5, it would



(a) Baseline (b) ED

Figure 3.6: Baseline `sync=on` vs ED Postgres Safe for read-only transaction commits

only see the effects of `XID 702`.

In the ED model, transactions commit and become visible first but become durable later. Fig. 3.8 depicts how the above snapshotting mechanism would work in ED Postgres. The setting is almost the same as depicted in Fig. 3.7, but the transactions are ED transactions this time. They commit and become durable at different times, allowing for interesting scenarios. At time unit 5 when the snapshot is taken, `XID`s `700` and `705` are still "running", while `XID 702` has "finished", and `XID 707` has not started yet. However, the difference, in this case, is that `XID 700` has *committed* and is *visible*, which means the snapshotter can see it and include it in the snapshot. This time, the snapshot would not only contain `XID 702`, but also contain `XID 700` because it committed before the snapshotting started. In summary, the snapshot would contain just `XID 702` in unmodified Postgres, while it would contain *both* `XID 702` and `XID 700` in ED Postgres.

We now present how this visibility change was achieved in Postgres. Algorithm 1 outlines, at a high level, the steps that are involved when a transaction commits in unmodified Postgres and gives a concrete shape to the states presented in Fig. 3.4. When the transaction attempts to commit, the `CommitTransaction` procedure first generates a commit record and writes it in the in-memory `XLOG`. If *sync* is greater than *off* (or simply *on* in our case), the back end waits for disk flush and remote replication before making the transaction visible. If the *sync* is *off*, however, it immediately acknowledges the client after writing the commit record to the in-memory `XLOG`.

The ED model allows us to make the transaction visible as early as right after writing the commit record to `XLOG` (line 3 in Alg. 1). We, therefore, allow all transactions to be visible as soon as their commit record is in the in-memory `XLOG`. This is shown in line 6 in Algorithm. 2. The actual process of changing the visibility involves marking the transaction as 'complete' in `ProcArray` but still holding on to heavyweight locks, transaction metadata and resources. These locks and resources are released, and the final clean-up steps are completed once the transaction replicates on synchronous standbys.

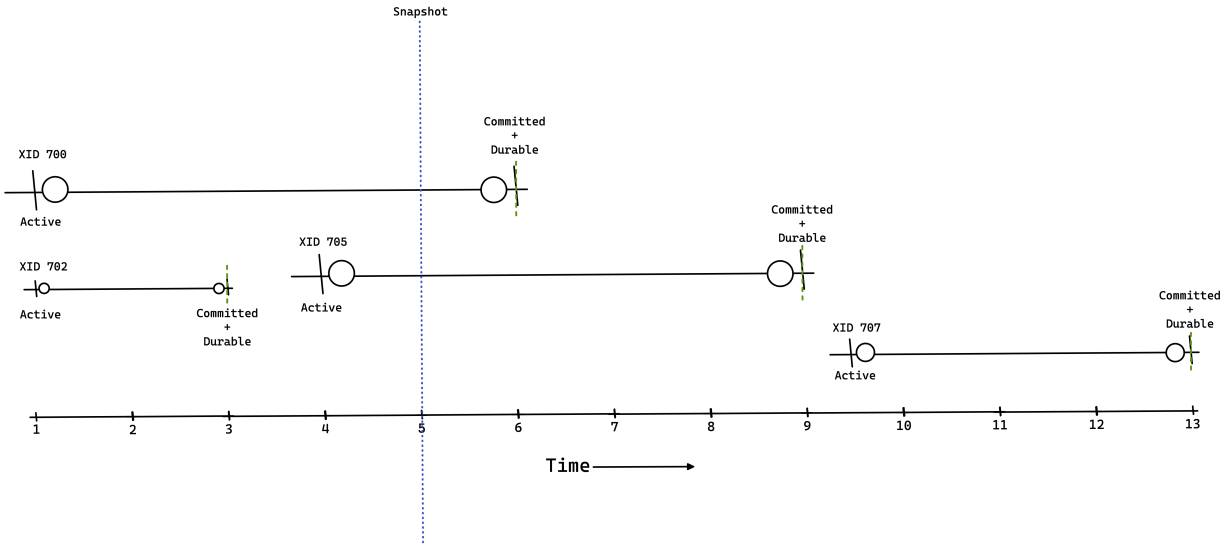


Figure 3.7: Snapshotting in unmodified Postgres

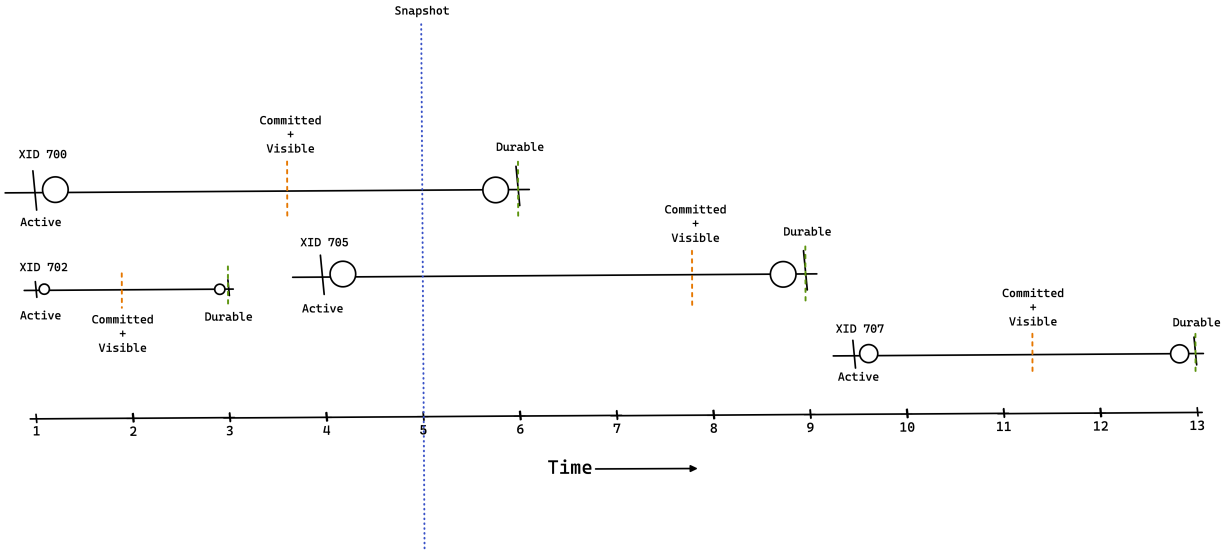


Figure 3.8: Snapshotting in ED Postgres

Algorithm 1 Commit Path of a Read-Write Transaction in Unmodified Postgres

```
1:  procedure CommitTransaction(T)
2:      c ← Commit Record of T              ▷ Generate a commit record for the transaction
3:      lsn ← XLogWrite(c)                  ▷ Write commit record to in-memory XLOG
4:      CLogWrite(T, xact_committed)        ▷ Mark T as committed in CLOG
5:
6:      if sync=on then
7:          XLogFlush(lsn)                  ▷ Flush up to the commit record's LSN
8:          WaitForReplication(lsn)         ▷ Wait for replication to finish
9:      end if
10:
11:     MakeVisible(T)                      ▷ Make T visible to other transactions
12:     ReleaseResources(T)                 ▷ Release resources held by T
13:     sendClientAck()                     ▷ Acknowledge client's commit request
14: end procedure
```

This seemingly minor re-ordering of actions in the commit path of transactions actually results in a massive effect on reducing abort rates and increasing throughput – as we will see in the evaluations chapter. With this change, transactions 'stay alive' only for tens of microseconds as opposed to some milliseconds, irrespective of how slow disks are or how long it takes for replication to complete. Indeed, safe transactions wait for both disk flush and remote replication before getting acknowledged, but, as far as Postgres is concerned, all transactions cease to remain 'in progress' as soon as their commit record is in the in-memory `XLOG`. This directly translates to fewer conflicts, reduced aborts, and, thus, a higher overall throughput rate under a contentious workload.

Algorithm 2 Commit Path of a Read-Write Transaction in ED Postgres

```
1:  procedure CommitTransaction(T)
2:      c ← Commit Record of T              ▷ Generate a commit record for the transaction
3:      lsn ← XLogWrite(c)                  ▷ Write commit record to in-memory XLOG
4:      CLogWrite(T, xact_committed)        ▷ Mark T as committed in CLOG
5:
6:      MakeVisible(T)                      ▷ Make T visible to other transactions
7:
8:      if sync=on then
9:          XLogFlush(lsn)                  ▷ Flush up to the commit record's LSN
10:         WaitForReplication(lsn)         ▷ Wait for replication to finish
11:     end if
12:
13:     ReleaseResources(T)                 ▷ Release resources held by T
14:     sendClientAck()                     ▷ Ack client's commit request
15: end procedure
```

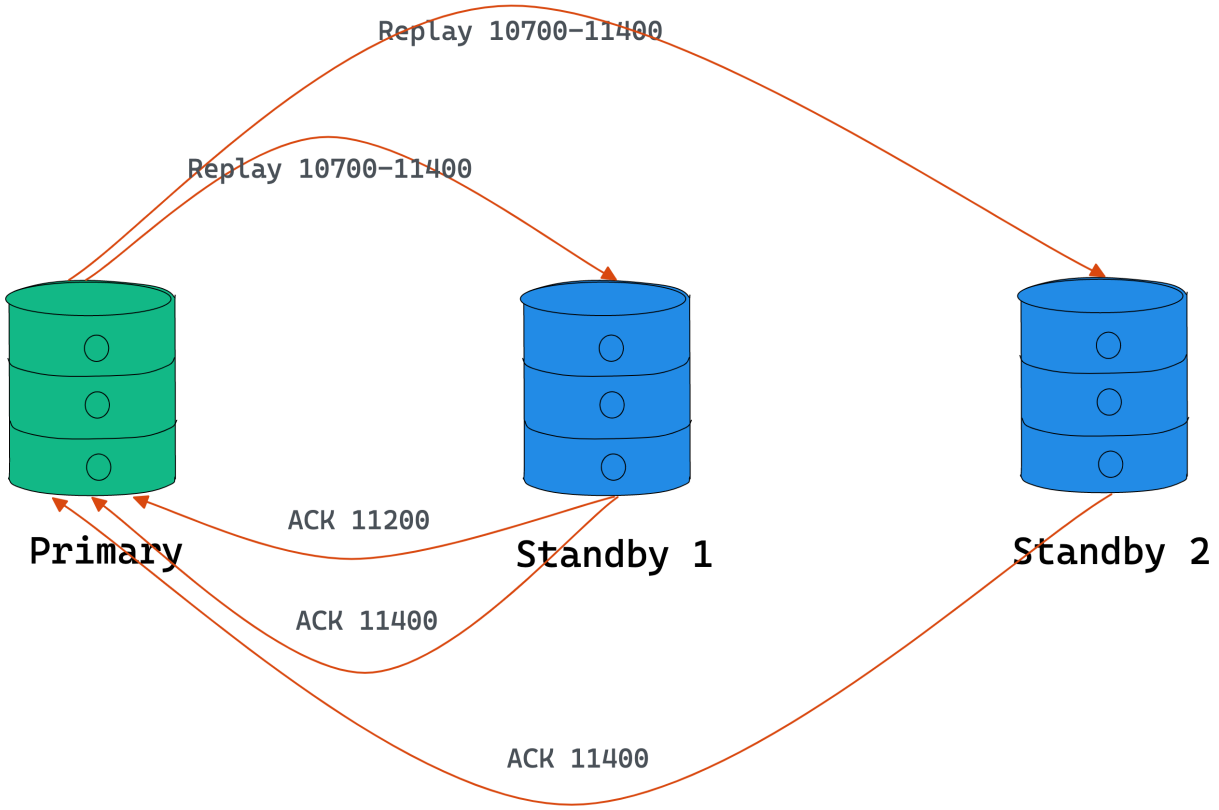


Figure 3.9: WAL Send/Receive in Streaming Replication

3.8 Implementing ED Recoverability

The next big piece in implementing ED in Postgres was making safe, read-only transactions wait for their dependencies to become fully durable – i.e., flushed to local disks and replicated to synchronous standbys. In unmodified Postgres, read-write transactions with `sync=off`, use a procedure called `SyncRepWaitForLSN()` to implement the wait logic. This procedure accepts the `LSN` of the commit record of the transaction and initiates the waiting process. It first resets the current Postgres back end's latch and adds it to a shared memory queue. When the `WALReceiver` gets an acknowledgement from synchronous standbys for previously sent `XLOG` records, it also receives `LSN` values. These `LSN` values serve as high-watermarks that indicate the location up to which the `XLOG` records were replayed on the standbys. Using these `LSN` values, the `WALReceiver` wakes up all waiting back ends in the wait queue up to the received `LSN`s by setting their latches.

In the example illustrated in Fig. 3.9, there is one primary server and two synchronous standbys. The primary initially sends `XLOG` records from `LSN` 10700 – `LSN` 11400 to both the standbys. Standby 2 acknowledges up to `LSN` 11400 in its first reply (or heartbeat). But standby 1 initially acknowledges up to 11200 and then eventually acknowledges up to 11400. If the commit record of the current transaction was at or below `LSN` 11400, the

primary can wake up back ends only up to LSN 11200 when it first receives replies from both standbys. It cannot wake up the back end waiting for LSN 11400. It is only when it receives the second reply from standby can it proceed to wake up the transaction waiting on LSN 11400.

We wanted to reuse the SyncRepWaitForLSN() procedure since it is already a well-designed wait mechanism built into Postgres. However, the issue was that read-only transactions do not create any XLOG entries and hence, do not have a commit record LSN. One option to circumvent this problem is to track reads and calculate a moving maximum of the highest LSN of the commit record of the read dependencies. However, this approach would have been relatively complex, and we wanted to stick to our principle of minimal intrusion for changing Postgres. Therefore, the more straightforward alternative to this was to compute the current insertion point in the XLOG at snapshot creation time for the read-only transaction and use that as the LSN to wait for in SyncRepWaitForLSN(). We call this the maxLSN. This maxLSN of the read-only transaction is guaranteed to be higher than the commit-record LSN of all committed transactions. Therefore, it is a safe but conservative choice.

Algorithm 3 Wait algorithm for safe, read-only transactions in ED Postgres

```
1:  procedure SafeWait(T, maxLSN)
2:      readOnly ← isReadOnly(T)
3:      if readOnly && sync=on then
4:          if !standbysConfigured then
5:              XLogFlush(maxLSN)
6:          end if
7:          remoteFlushLSN ← GetRemoteLSNs()
8:          if maxLSN > remoteFlushLSN and remoteFlushLSN > 0 then
9:              SyncRepWaitForLSN(maxLSN)
10:         end if
11:     end if
12: end procedure
```

The implemented wait logic for safe, read-only transactions is shown in Algorithm. 3. The current insertion point in the XLOG – maxLSN is recorded when the snapshot is created for the read-only transaction. If it were a read-write transaction, a new XLOG, record would have been inserted at this location. However, for our case, we need this maxLSN to wait for replication and not for inserting new records. The algorithm begins by checking if the current transaction is read-only and if the *sync* level is *on*. If no synchronous standbys have been configured, then we need to ensure that the transaction waits for the durability of prior fast transactions. We do this by forcing a disk flush up to the obtained insertion pointer – the maxLSN. This is sufficient to ensure the correct behaviour of safe read-only transactions under the ED model in a single-node case.

When synchronous standbys are configured, we first obtain the LSN up to which the XLOG has been flushed on remote servers. We then check if the maxLSN value is greater

than `remoteFlushLSN`. This would indicate that there are newer records that were not yet replayed on the standbys. In this case, we invoke `SyncRepWaitForLSN()` and pass in the obtained `maxLSN`. This performs the actual waiting process, and the back end gets woken up when the standbys have acknowledged up to the supplied `LSN`.

With the above two changes implemented in Postgres – visibility change and safe, read-only waits, a prototype of the ED model with PostgreSQL as a test bed is complete.

3.9 An Argument for Correctness

3.9.1 Recoverability

We will quickly revisit recoverability and serialisability definitions under ED before providing an argument for the correctness of the ED implementation in Postgres. From Chapter 2, an ED history is recoverable if *both* of the following conditions hold:

1. If T_2 reads from T_1 in H, then T_2 does not commit before T_1 commits.
2. If T_2 reads from T_1 in H, then T_2 does not become durable before T_1 becomes durable. That is, if $d_2 \in H$, then $d_1 \in H$ and $d_1 < d_2$.

Although we change the point at which transactions become visible, we do not do it *before* a transaction is committed. In other words, we continue to ensure that only committed transactions are visible. Say that in ED Postgres, a transaction T_2 reads from another transaction T_1. This means that T_1 was included in T_2's snapshot – formed on T_2's first operation. Therefore, T_1 *must* have committed before T_2's snapshot is created for it to be included in T_2's snapshot. Now, if T_2 commits, it will be after the point at which its snapshot was taken, and therefore, its commit record will be ordered *after* the commit record of T_1 in the `XLOG`. Hence, enforcing the first condition of recoverability.

To prove that condition 2 of ED recoverability is upheld in ED Postgres, consider, once again, that a transaction T_2 has read from another transaction T_1. The recoverability property has to hold if T_2 is a read-write transaction as well as if it is just a read-only transaction.

In the first case, when T_2 is a read-write transaction, we have seen from the previous argument that T_2 can commit only after T_1 has. Therefore, it follows that T_2's commit record succeeds T_1's commit record in the `XLOG`, and hence, must have a higher `LSN` value than T_1's commit record. Postgres always flushes the in-memory `XLOG` in `LSN`-order. Furthermore, `XLOG` records also are replicated to standbys in `LSN` order. So, if T_2's commit record has a higher `LSN` than T_1's commit record, T_1 *must* have become durable (flushed to disk and replicated to standbys) before T_2's commit record becomes durable.

In the second case, when T_2 is a read-only transaction, it does not generate any `XLOG` records, and therefore, recoverability is guaranteed differently here. From the first condition

of ED recoverability, we know that if T_2 has seen T_1's effects, then T_1 *must* have committed. In ED Postgres, the latest insertion point in the `XLOG` is captured as an `LSN` value in a transaction's snapshot – we call this the `maxLSN`. We have seen earlier that if a read-only transaction is safe, it waits until all records up to `maxLSN` in the `XLOG` are fully durable (flushed to disk and replicated to standbys) before being acknowledged. Therefore, if T_2 is a safe, read-only transaction, its `maxLSN` will be at least as high as T_1's commit record's `LSN`. So, if, at commit time, T_2 waits for the `maxLSN` to become durable, it would have effectively waited for all of T_1's effects to have become durable. This concludes the correctness argument for ED recoverability in ED Postgres.

3.9.2 Serialisability

In general, a history H with a partially ordered set of interleaved transaction operations is serialisable if its execution produces the same output as some serial execution of transactions in H. For ED serialisability, we say that the committed projection of H, denoted as $C(H)$, must be serialisable for H to be serialisable. The committed projection $C(H)$ will only contain operations from committed transactions and their respective *commit* events, but none of the operations from *failed* transactions or any *durability* events of committed transactions. For a full discussion, please refer back to Chapter 2.

We will consider two cases. For the first case, we assume that Postgres does not crash and restart. Since committed (but not durable) transactions only fail because of crashes, this means that the execution history H does not include any transaction failures. This, in turn, means that $C(H)$ includes all and only the committed transactions from H, whether they are durable or not. If we assume that Postgres' existing concurrency control mechanism correctly serialises all committed transactions, this mechanism will also guarantee ED serialisability in this case.

For the second case, suppose that there has been a crash. When Postgres recovers from the crash, one or more committed (but not durable) transactions from before the crash may be lost, i.e., they may fail. Thus, although such transactions were in H, they will no longer be present in $C(H)$ after recovery. If any surviving transaction depends on such a failed transaction, our schedule will no longer be ED serialisable: a surviving transaction (in $C(H)$) cannot depend on a transaction that is not in $C(H)$ in an ED serialisable history. Fortunately, this cannot occur if the system enforces ED recoverability, which ensures that committed transactions survive (are durable) only if their dependencies are also durable. Thus, the history must be ED serialisable in this case as well.

3.10 Shortcomings and Scope for Future Work

Safe transactions in the current ED implementation in Postgres wait for an `LSN` value much higher than is actually necessary. When the safe, read-only transaction starts, we

get the highest LSN value from the XLOG module and wait for the remote disk flush up to that LSN. However, the transaction might have only read from tuples that have already replicated and therefore does not need to wait. In such cases, in-flight write transactions might unnecessarily slow down safe, read-only transactions. To remedy this, we need to keep track of the highest commit LSN among all XIDs the read-only transaction has read from and wait only for that LSN. If we find that the target LSN is already replicated, we return early from the wait loop. We plan to implement this in the next ED release of Postgres.

Chapter 4

Evaluation

In this chapter, we evaluate ED Postgres to obtain answers to the following questions:

- How much faster are "fast" transactions compared to "safe" ones in ED Postgres?
- How does the performance of fast and safe transactions vary when the system is subjected to a mixed workload consisting of both transaction types?
- Does ED Postgres reduce data contention, and thus abort rates, compared to baseline Postgres since resources are released earlier?
- How does ED implementation in Postgres affect performance under realistic workloads like TPC-C, and how can ED Postgres be used in similar realistic situations?

Section 4.1 describes the general setup used for the experiments. We then present the results of basic latency experiments in Section 4.2, mixed workload results in Section 4.3 and contention results in Section 4.4. Finally, we present the results of running TPC-C on ED Postgres and discuss the findings.

4.1 Setup

Most experiments, unless otherwise specified, use a 2-node setup of AWS `m5.large` instances with two Intel(R) Xeon(R) Platinum 8175M CPUs @ 2.50GHz, 8 GiB of memory and 100 GiB of Elastic Block Storage (EBS) volume attached to each instance. One node is configured as the primary, and the second is configured as a synchronous standby. A third node, an AWS `c5.4xlarge` instance with 16 Intel(R) Xeon(R) Platinum 8124M CPUs @ 3.00GHz, 32 GiB of memory and 30 GiB of EBS attached, is used as a client node. The client node is only used for running test scripts and workloads against the Postgres servers.

In single-region experiments, all three Postgres nodes are in the same region but may be in different availability zones (AZs). In multi-region experiments, the primary is in one

of the regions, and the synchronous standby is in a different region. The client is always co-located in the same region as the primary node for all experiment runs. Specific placement details are discussed just before presenting the results of the respective experiment.

We compile Postgres from source for both the unmodified and ED versions. For the unmodified version, we use the 15.1 release of Postgres. For the eventual durability version, we branch off from the 15.1 release, make modifications to implement eventual durability and use the 0.5 release of the new ED Postgres.

4.2 Basic Latency Experiments

We start by conducting basic latency measurements on ED Postgres to observe how transaction latencies vary when the synchronous standby is placed increasingly further away from the primary. There is a single table on the primary node with two columns, `key` and `value`, that are both integers. The test node uses pgbench[2] to generate transactions that each update one row. The `pgbench` process on the test server runs for 120 seconds and uses one thread and one client to issue one transaction at a time and records latency. This experiment is repeated multiple with the standby placed in different locations. We run this experiment on both ED Postgres and baseline Postgres. The results are shown in Fig. 4.1

The first set of values in the figure (`us-east-1a; sync=off`) compares the latencies of ED fast transactions and baseline `sync=off` transactions. Following this, we compare the latencies of ED safe transactions against baseline `sync=on` transactions when the standby is placed in different regions. The results of this experiment, when it ran on baseline Postgres alone, were presented in the introduction (Chapter 1) to make a point about durability costs. As with the previous run, the results obtained from ED Postgres also indicate that network and durability costs dominate transaction latencies as the standby server is placed progressively further away from the primary. The 95% confidence intervals of these measurements ranged from $\approx 6\mu$s to $\approx 52\mu$s. The main conclusion here is that, as expected, ED fast transactions have similar latencies as `sync=off` transactions in baseline Postgres, and ED safe transactions have similar latencies as `sync=on` transactions in baseline Postgres for different placements of the standby server.

4.3 Mixed Workload

To address the second question we sought to answer, we subject ED Postgres to a workload consisting of a mix of fast and safe transactions issued at a constant rate per second. We vary the percentage of fast transactions in the mix from 0% – 100% and measure the overall average latency, average fast transaction latency and the average safe transaction latency. This is done by running two `pgbench` clients in parallel and varying their individual rates

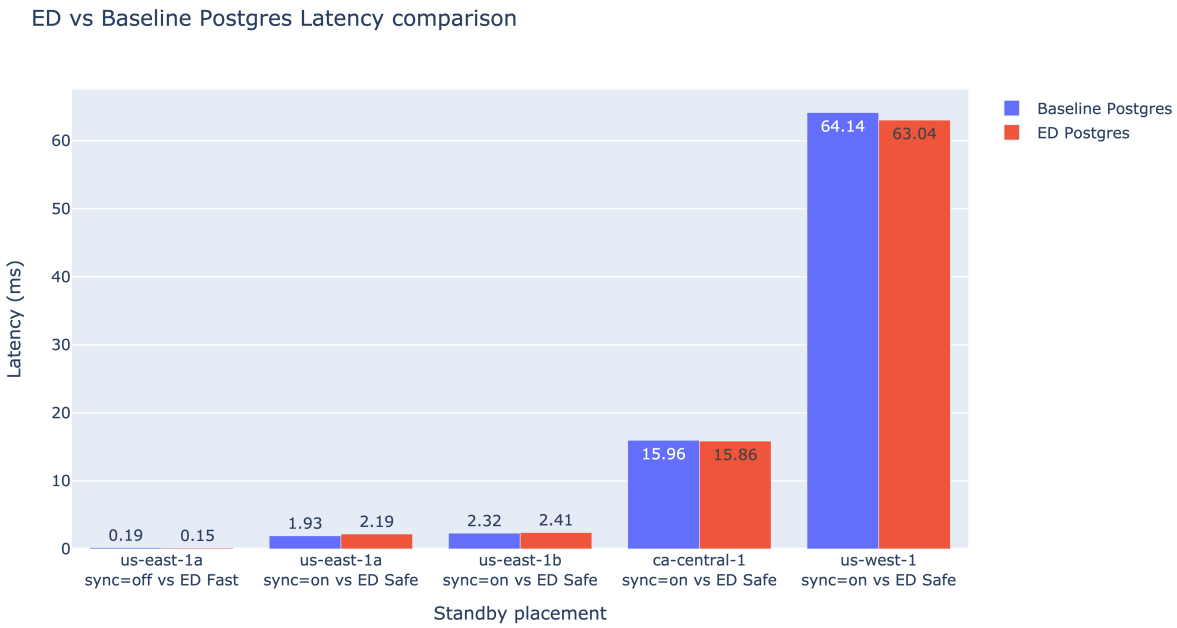


Figure 4.1: Basic Latency tests: Baseline vs ED Postgres

to control the percentage of fast transactions in the workload mix.[1] The setup for this experiment is slightly different from the one described at the beginning of this chapter. We use an AWS `c5d.4xlarge` instance for the primary and an `m5.large` instance for the standby. Furthermore, we ensure that Postgres' data directory on the primary is on the instance store and not on an EBS volume to reduce disk contention for safe transactions. More details on this are forthcoming. Each `pgbench` client uses 16 threads and 16 clients to issue transactions at a total rate of 4000 TPS for 120 seconds. Each transaction performs a simple insert on a table with two integer columns. Lastly, we run a purely fast and purely safe workload with just one thread and one client for the same duration and plot those lines for reference. The results are shown in Fig. 4.2

With this experiment, we wanted to know if fast transaction latencies are, in any way, affected when safe transactions are introduced. We see from the results that the latency of fast transactions remains practically unchanged no matter how many safe transactions are added to the mix. As the percentage of fast transactions in the total mix increases, it drives down the total average latency, as expected. The fast transactions' latencies in the mixed workload stay fairly flat but, on average, are about 50% higher when compared to single-client fast transaction latencies. This increase can be attributed to the fact that the system-under-test is handling around 10X more load and several more clients than the reference fast transaction test. On the other hand, safe transaction latencies display an interesting behaviour, as the percentage of fast transactions is increased in the total

[1] We did not use `pgbench`'s weight parameter to limit possible confounding costs from `set synchronous_commit` commands.

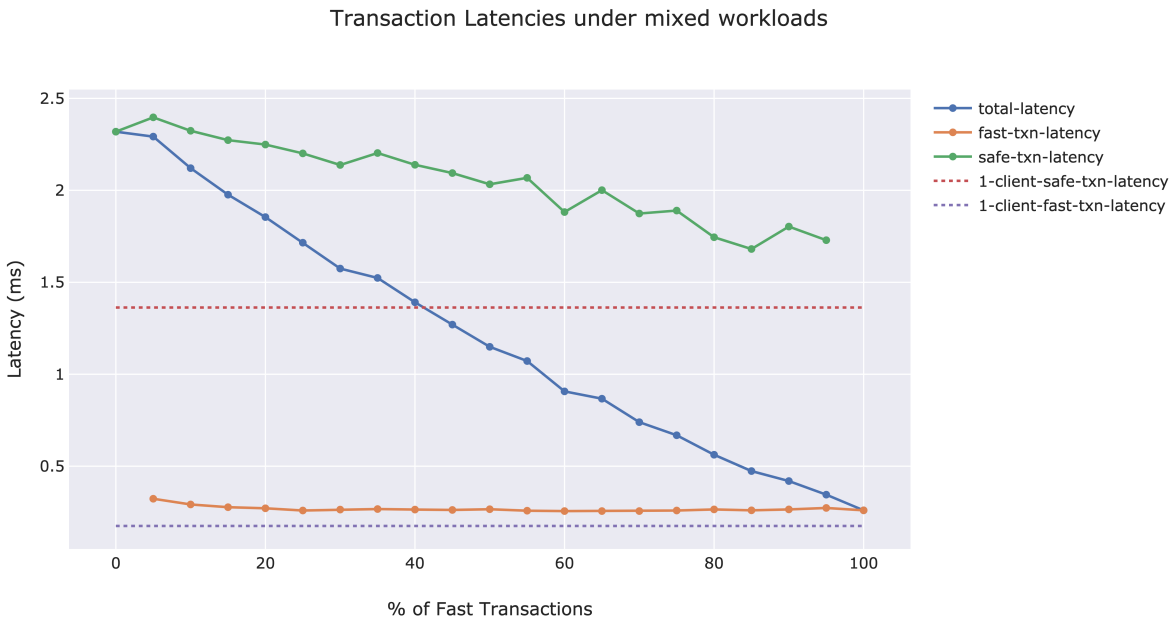


Figure 4.2: ED Postgres performance under a mixed workload

workload mix. They average at 2.3 ms when there are no fast transactions in the system, jump slightly by 3% when fast transactions are introduced at 5% of the total rate, and then gradually decrease to get closer to the reference, single-client safe latency as their share in the total mix comes down.

We found that this difference in safe transaction latencies from its single-client reference latency line is primarily because of the `WAL` writes. When there is a higher percentage of safe transactions in the mixed workload, they all compete to acquire the `WAL` write lock and subsequently flush `WAL` entries to disk. As the percentage of safe transactions, and thus their rate, decreases, there is no longer as much contention for the lock or the disk. Hence, we see a reduction in safe transaction latency as their percentage in the total workload reduces.

However, at any given point, the Postgres backend handling safe transactions still competes with the `WALWriter` for the `WAL` lock and the disk flush. This is because the `WALWriter` is responsible for flushing `WAL` entries to disk in the case of fast transactions. Hence, we see a slight increase in safe transaction latency when fast transactions are introduced into the workload. It is also why safe transaction latencies do not fully converge with the single-client reference latency at the 95% fast, 5% safe configuration.

Both deviant-from-ideal behaviours – increase in fast transaction latencies compared to their single-client reference line and the gradual decrease of safe transaction latency when their rate decreases – are also seen in baseline Postgres. The displayed behaviour of safe transactions in this experiment was further verified using an instance with an EBS store.

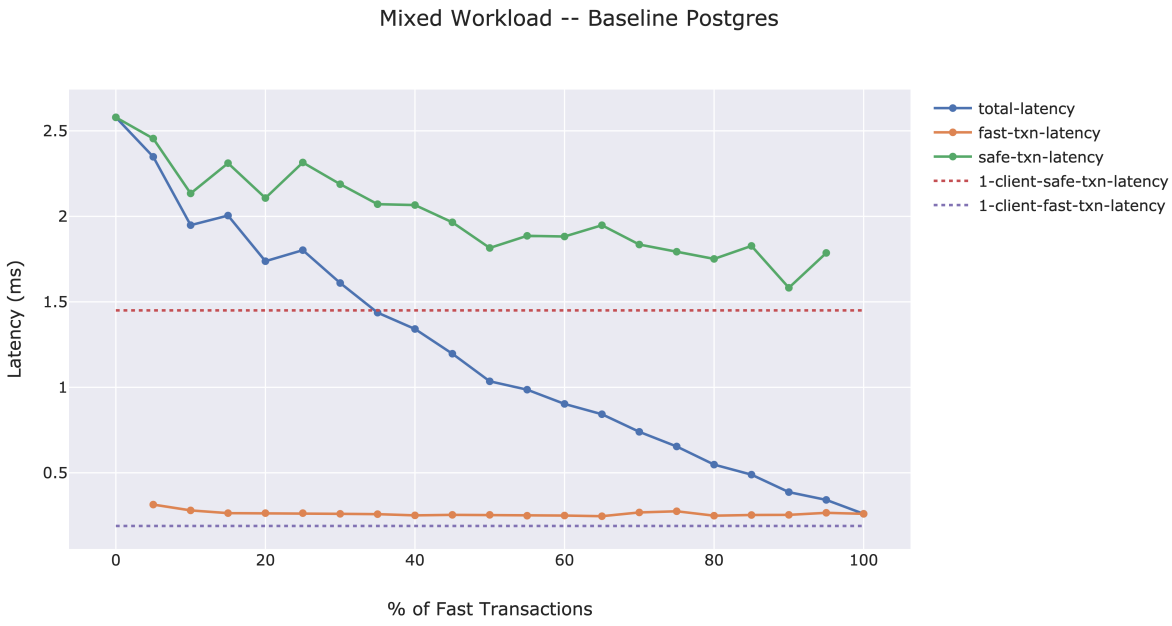


Figure 4.3: Baseline Postgres performance under a mixed workload

The decrease was found to be steeper since disk I/O is slower compared to instance storage, and hence, reduced safe transaction rates translated to steeper decreases in latency.

The results of running the mixed workload experiment on baseline Postgres are shown in Fig. 4.3. In conclusion, running fast and safe transactions together in ED Postgres at a reasonably demanding transaction rate does not significantly alter their individual average latencies by more than what the experiment itself effects – as was verified by running the mixed workload on baseline Postgres.

4.4 Contention Tests

Since, in ED Postgres, we make transactions visible earlier, we expect there to be fewer aborts under contentious workloads. We test this hypothesis by conducting experiments under varying levels of contention to observe abort rates in both baseline and ED Postgres. The test setup is very similar to the setup used for the mixed workload experiments above. We use an AWS `c5.4xlarge` instance for the primary node in `us-east-1a`, and an `m5.large` instance for the standby node in `us-east-1b`. The test client configuration and placement remain the same. We populate the database initially with a table containing 250 rows of keys and values – both integers. The `pgbench` client uses 16 threads and 16 clients to issue transactions at a rate of 4000 TPS. Each transaction updates one row in the table, and the row is chosen based on the set contention level. For example, if the contention level is 0.95, then there is a 95% chance that the transaction chooses a row from the first

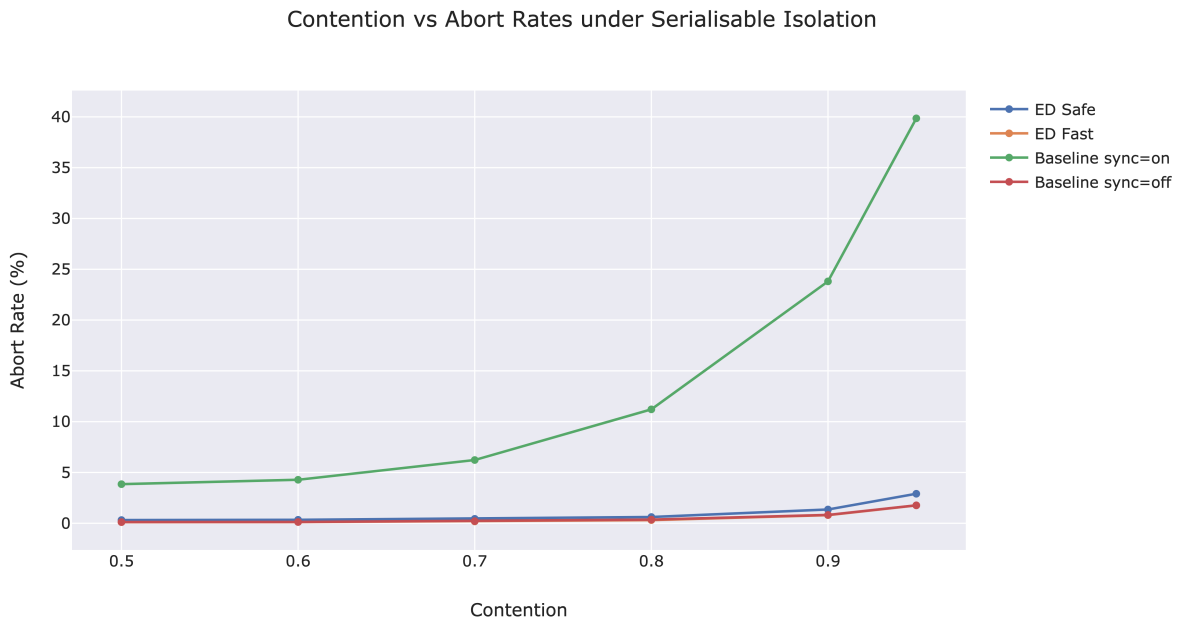
5% of the rows in the table and a 5% chance that a row is chosen from the bottom 95% of the table space. We vary contention levels from 0.5 – which is about the same as sampling from a uniform distribution, all the way up to 0.95, which we use as the highest contention level in our experiments. For each level of contention, we issue transactions at a rate of 4000 TPS for 120 seconds and measure abort rates.

We begin by performing this experiment at Postgres' *serializable* isolation level. The results are shown in Fig. 4.4a. There is a pronounced difference in abort rates between baseline `sync=on` transactions and the other transaction types in the graph. We will be using `sync=on` to refer to the `synchronous_commit=on` option and `sync=off` to refer to the `synchronous_commit=off` option in baseline Postgres. Baseline Postgres has an average of about 15X more aborts compared to ED Safe transactions and ranges between having 12X more aborts in the best case when the contention level is 0.5; up to having 18X more aborts than ED safe transactions in the worst case when the contention level is 0.8. ED safe transactions, on average, offer a 93% reduction in aborts compared to baseline `sync=on` transactions under the *serializable* isolation level.

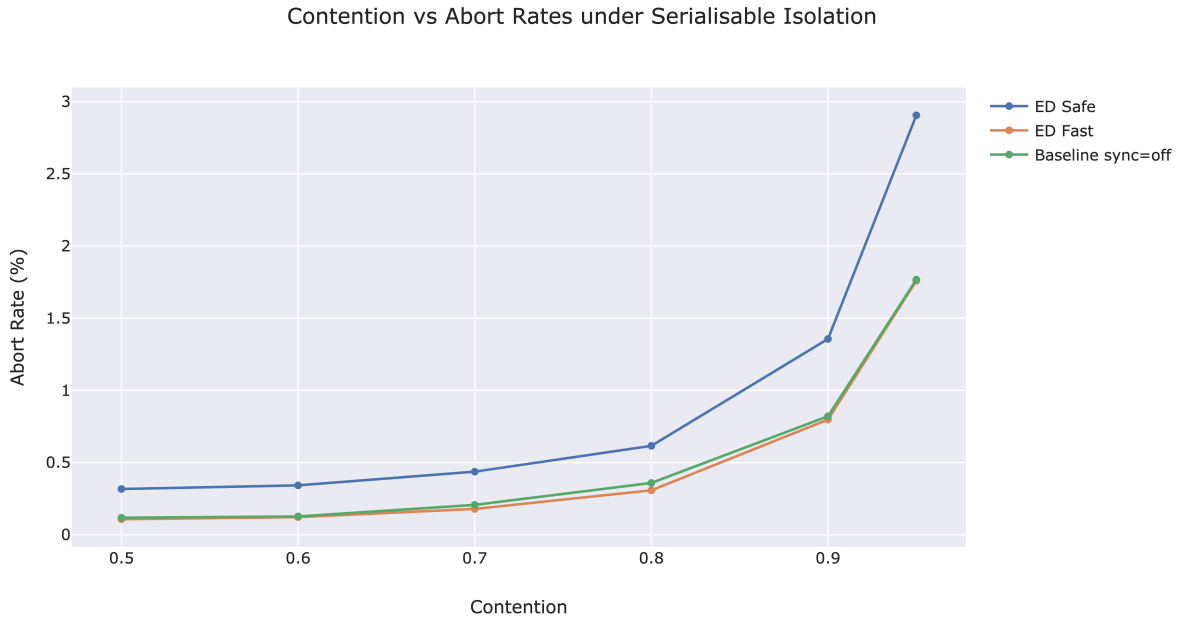
This significant difference in aborts between baseline `sync=on` transactions and the rest of transactions is because of two reasons – concurrent updates and serialisation anomalies. Postgres' 'repeatable read' isolation corresponds to a slightly stricter version of Snapshot Isolation (SI) [3] defined by the ANSI SQL standard [10]. This isolation level uses the *first-committer-wins* policy and aborts transactions when they have overlapping write sets with transactions that were concurrent but just finished committing. Apart from these kinds of aborts caused by SI rules, Postgres further aborts more transactions under its 'serializable' isolation level. Postgres' 'serializable' isolation level uses serialisable snapshot isolation that aborts transactions with r–w dependency cycles.

In the case of baseline `sync=on` transactions, they do not become visible or release their locks until after `XLogFlush` of their commit record and waiting for synchronous replication (`SyncRepWait`). This results in the transaction staying active until it is fully durable. When the transaction stays active for a longer period, there is a higher chance of concurrent transactions queuing up for the row lock, waiting to update the same row, and waiting for the completion or abortion of our long-running transaction. If the long-running transaction commits successfully, all transactions waiting are aborted. These aborts are, in addition to aborts caused by SSI check failures on the predicate locks.

On the other hand, the behaviour of ED safe transactions is different. They become visible as soon they commit, and their status is updated in the in-memory `CLOG`. However, they still hold on to their locks until `XLogFlush` and `SyncRepWait` are complete. That means that concurrent transactions would see their status as *committed* while the transaction is still waiting for the disk flush of its commit record and replication to standbys. Therefore, when concurrent transactions try to update the same row, they see that the row is locked, but the updating transaction has committed. Postgres allows the row to be updated in such cases. There are still some aborts in ED safe transactions because of concurrent updates – when competing transactions see that the row lock is held *and* the updating transaction is in progress. The combined abort rates because of concurrent



(a) With baseline `sync=on` transactions



(b) Without baseline `sync=on` transactions

Figure 4.4: Contention vs Abort Rates under Serialisable Isolation

updates and SSI check failures are still significantly lower than baseline `sync=on` abort rates.

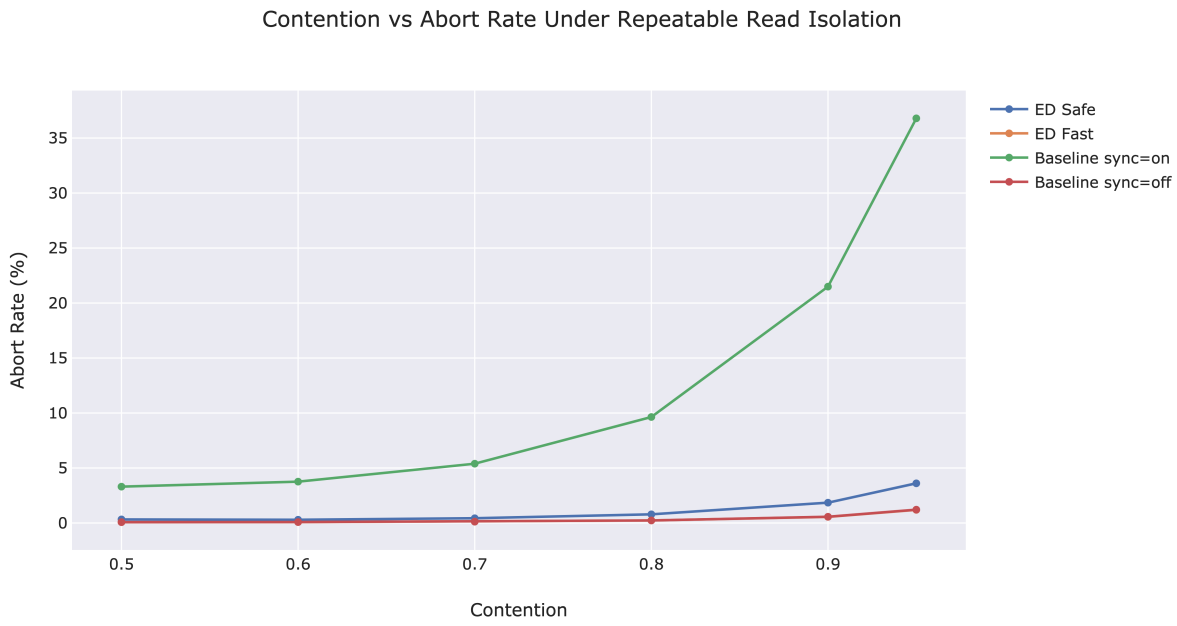
Fig. 4.4b shows the difference in abort rates between ED Safe, ED Fast and baseline `sync=off` transactions. It is the same graph as Fig. 4.4a, but with baseline `sync=on` transactions removed, and the graph zoomed in to show the abort rate differences clearly. ED Fast and baseline `sync=off` transactions have very similar abort rates since we do not significantly alter the execution behaviour of fast transactions in ED Postgres from their read-write, `sync=off` counterparts in baseline Postgres. However, we observe that ED safe transactions have 1.65X to 2.9X more aborts compared to ED fast transactions, even though their execution paths are the same up until `XLogFlush` and `SyncRepWait` happen. In other words, both ED fast and ED safe transactions perform the same steps in the transaction manager and become visible right after they commit. At this point, ED fast transactions short-circuit out of waiting for `XLogFlush` and `SyncRepWait`, while ED safe transactions wait for them. Both transaction kinds release locks and resources after this step.

Since ED safe transactions hold on to heavyweight locks until they are fully durable, it is possible that concurrent ED safe updaters queue up on these heavyweight locks, after which they all get aborted once the locking transaction commits. This is because Postgres aborts transactions that finish waiting for a lock, acquire it, but try to update the same tuple as the earlier locking transaction. Since Postgres does not have a well-defined queueing mechanism for row locks, the presence of heavyweight locks in the lock table acquired by ED safe transactions may interfere with concurrent updaters after the locking transaction has committed but before it is durable. Hence, we see a higher number of aborts compared to ED fast transactions, even though their execution paths are similar. However, ED safe transaction aborts are still lower than baseline `sync=on` transactions because, in the latter case, a greater number of updaters queue up since the locking transaction is *in-progress* for a longer duration.

We now re-run this experiment by setting the isolation level to *repeatable read* in Postgres. Since the repeatable read isolation level does not have SSI checks (predicate lock checks), we expect fewer aborts in both baseline and ED Postgres. Fig. 4.5a shows the results with baseline `sync=on` transactions included.

There is still a significant difference between ED safe and baseline `sync=on` abort rates, but it is slightly lesser compared to the difference when the isolation was set to 'serializable'. Baseline `sync=on` transactions have between 10X – 12.45X times more aborts than ED safe transactions across different contention levels, with an average of 11.5X more aborts. ED safe transactions, on average, offer a 91.25% reduction in aborts compared to baseline `sync=on` transactions under the *repeatable read* isolation level.

Fig. 4.5b compares the abort rates between ED safe, ED fast and baseline `sync=off` transactions. The abort rates of ED fast and baseline `sync=off` transactions are nearly identical, while the abort rates of ED safe transactions are about 3X higher. In fact, it ranges from 3X to 4.3X more aborts and averages around 3.5X more aborts compared to ED fast transactions. This difference in abort rates can be attributed to the same reasons



(a) With baseline `sync=on` transactions

(b) Without baseline `sync=on` transactions

Figure 4.5: Contention vs Abort Rates under Repeatable Read Isolation

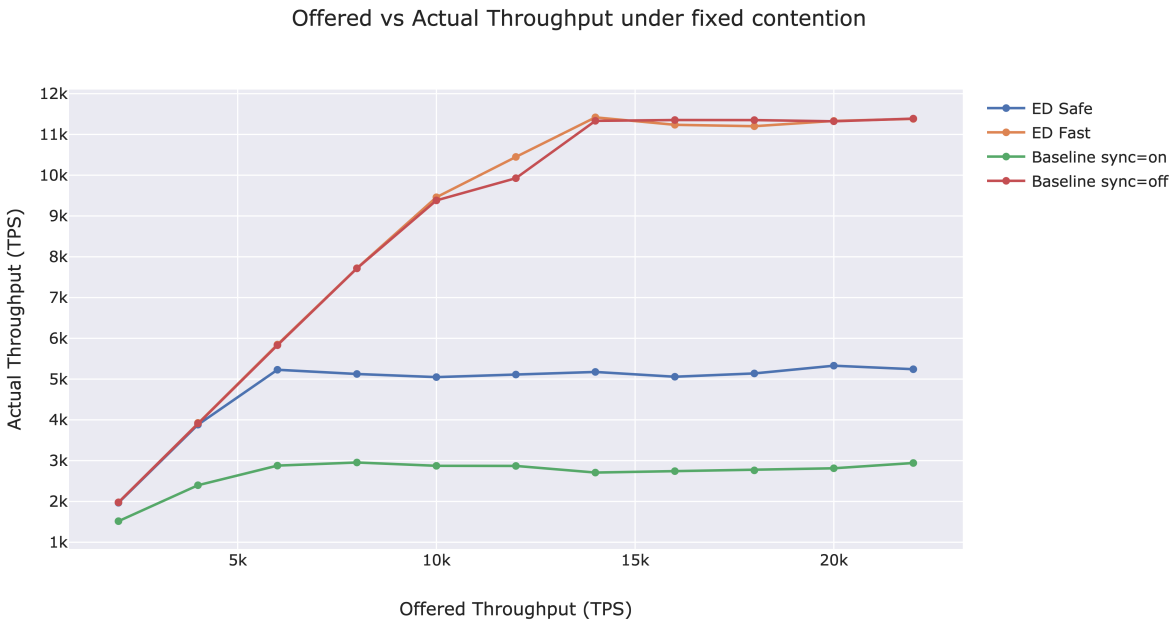


Figure 4.6: Actual Throughput vs Offered Rate Under 0.95 Contention

specified earlier for the *serializable* isolation level. This result also shows that the aborts are primarily because of concurrent update failures and not because of SSI check failures. If they had been mostly because of the latter, we would have seen a steeper reduction in abort rates in ED safe transactions under the *repeatable read* isolation level.

4.4.1 Throughput vs Offered Rate under Fixed Contention

We have just seen that ED Postgres offers between 91.25% to 93% reduction in abort rates compared to baseline `sync=on` transactions under a contentious workload. Since fewer aborts mean that a higher number of transactions succeed, we expect ED Postgres to offer higher throughput compared to baseline Postgres.

To test this, we use the same setup as we did for contention tests above, but instead of varying contention, we vary the offered rate from 2000 TPS to 22,000 TPS and fix contention at 0.95. Each run lasts for 120 seconds. The results of this experiment are shown in Fig. 4.6. ED Postgres offers, on average, about a 75% increase in throughput compared to baseline Postgres. We observe that as the offered load increases, the actual throughput from all four Postgres flavours initially rises and then saturates at a rate above which they cannot serve requests.

Naturally, both ED fast and baseline `sync=off` transactions plateau much later compared to ED safe and baseline `sync=on` transactions because they are asynchronous, short-lived, and hold on to resources and locks for tens of microseconds as opposed to some

milliseconds in the case of safe or `sync=on` transactions. This allows backends to become free earlier and serve more transactions, which ultimately results in an overall increase in throughput.

ED safe transactions plateau slightly later than baseline `sync=on` transactions while maintaining a difference of about 75% more throughput. Once again, this increase in throughput can be attributed to backends marking transactions *committed* much earlier compared to baseline Postgres, thus freeing up backends to serve new transactions and resulting in fewer aborts.

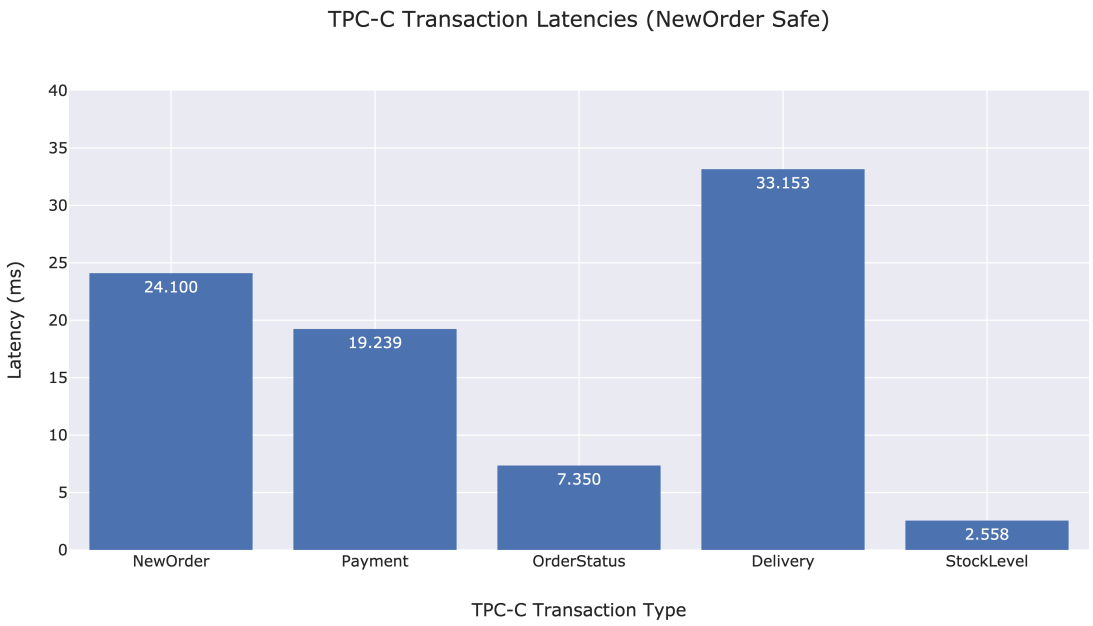
4.5 TPC-C

To test ED Postgres under a realistic workload, we run TPC-C and measure the latencies of different transaction types. Our intention behind running a workload like TPC-C is to show that in a real-world situation when multiple types of transactions are involved, ED Postgres can be used to set which ones should execute quickly and which ones safely – all while enjoying the benefits ED provides. The setup of this experiment is similar to the one used for the contention tests, but instead of using a standby instance in `us-east-1b`, we use a standby placed in `ca-central-1`. This change allowed us to clearly observe latency differences between safe and fast transactions – specifically, the `NewOrder` transaction.

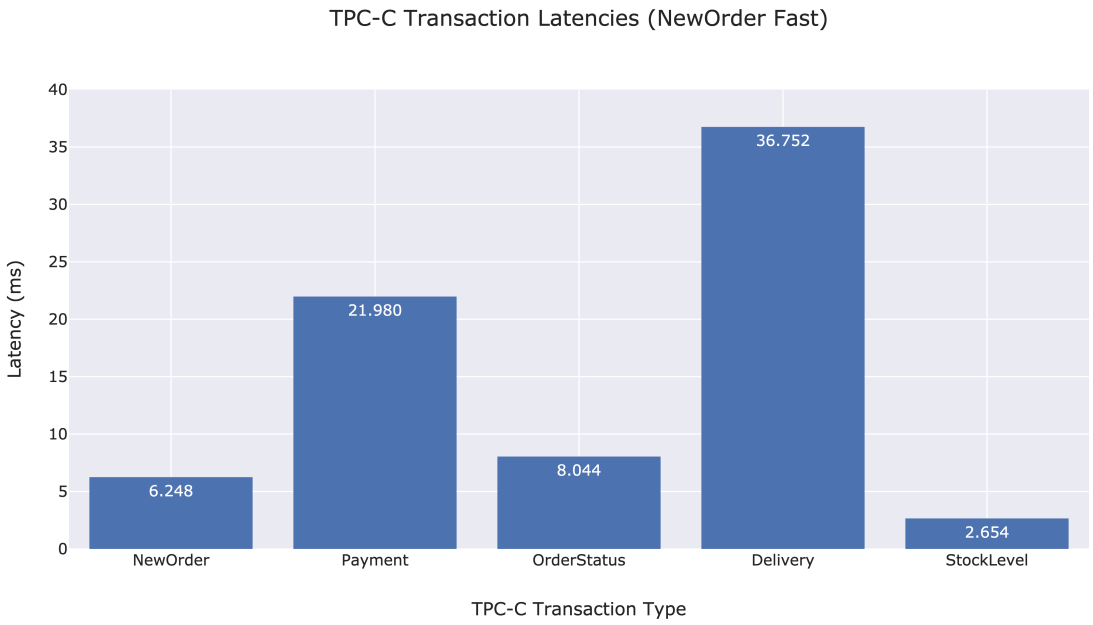
We use CMU's benchbase [19] – with minor modifications to control transaction durability – to carry out the TPC-C test. We wanted to keep the experiment setting simple, so we used a scale factor (number of warehouses) of 100 and 16 terminals. The test ran for 10 minutes at an unlimited rate (terminals issued transactions as fast as possible), and we recorded latencies and throughputs of the different transaction types. Fig. 4.7a shows the results when all transaction types of TPC-C are issued as safe transactions. All of the transaction types that perform updates (`NewOrder`, Payment and Delivery) have latencies that are consistent with previous experiment runs when the standby was placed in `ca-central-1`.

Fig. 4.7b shows the latencies of transactions when NewOrder is issued as a fast transaction. The latency of the `NewOrder` transaction drops by about 74% because it is now issued as a fast transaction. Secondly, although the latencies of the other transaction types do not deviate much from when `NewOrder` was safe, they do increase slightly. This increase is because the system is now processing more transactions per second compared to when `NewOrder` was safe.

Fig. 4.8 compares the average throughputs of transaction types. The purple bars correspond to transaction throughputs when all transaction types were issued as safe transactions. The red bars show the transaction throughputs when NewOrder was fast. Since we make `NewOrder` transactions fast, we expectedly see an increase in their throughput. However, we also notice that the throughput of Payment transactions increases. As database and system resources are released much earlier in the case of fast transactions (NewOrder),



(a) TPC-C Transaction Latencies when `NewOrder` is Safe



(b) TPC-C Transaction Latencies when `NewOrder` is Fast

Figure 4.7: TPC-C Transaction Latencies

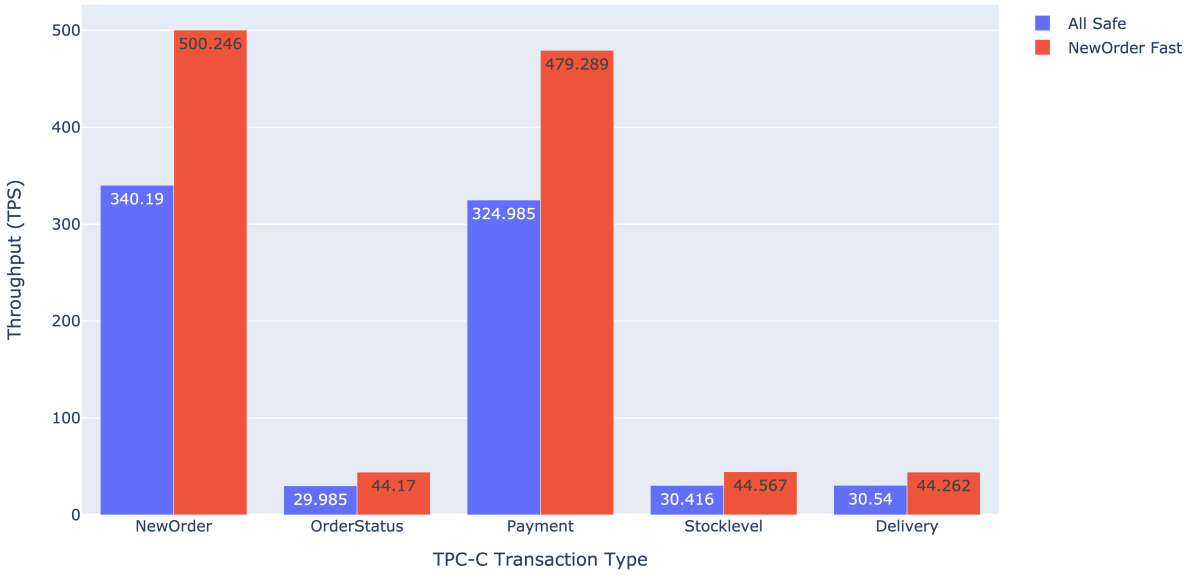


Figure 4.8: TPC-C Average Throughput Comparison by Transaction Type

the shorter turn-around time and an "unlimited" rate setting translate to terminals issuing a greater number of transactions resulting in higher throughput across all transaction types. We observe that the throughput of the NewOrder transaction is about 47% higher compared to when it was a safe transaction. Secondly, even though we retain the Payment transaction as safe, we still notice a $\approx 47.5\%$ increase in throughput for the aforementioned reasons. There is a consistent increase of 45%-47% throughput across all transaction types when NewOrder is issued as a fast transaction. We attribute this increase to the greater availability of resources because of the fast NewOrder transaction and the unlimited rate setting on the test client. Had we used a constant rate that the database was fully comfortable handling, we would not have seen the increase in throughput in the latter case.

Although the scale factor and the number of terminals used in this experiment are small, they are still high enough to be moderately demanding on the database system. Our intention with running TPC-C was not to benchmark ED Postgres against baseline Postgres but rather to show that ED Postgres can be used in realistic settings with potentially multiple transaction types to have fine-grained control over which transactions should execute quickly and which ones safely – all while enjoying ED Postgres benefits like reduced contention, increased throughput and recoverability.

The latency and throughput benefits obtained by tagging NewOrder as fast could also have been obtained in baseline Postgres by setting `sync=off`. However, the difference with ED Postgres is that an application can comfortably issue fast NewOrder transactions knowing that OrderStatus, Delivery and Payment are safe transactions. Because of the ED recoverability property, the latter set of transactions can only succeed if NewOrder

committed and became durable. This seemingly simple difference which can arguably be solved by application logic becomes important in applications like auctions and gaming.

Chapter 5

Related Work

5.0.1 Tackling Durability Costs

The idea of tackling durability costs has been around for almost as long as databases and storage systems have. One the earliest database systems – IBM's IMS/VS [23] provided a fast path for memory-resident data alongside the standard IMS transactions. The IMS database predates Codd's relational model [16], Gray's transaction model [26] and the development of many other fundamental ideas around modern database systems. Therefore, it is not surprising that the issue of managing durability costs has seen a considerable number of approaches since then.

Tackling durability costs initially began with efforts to leverage main memory for faster transaction processing. Some of the earliest works in this space are DeWitt et al. [18], Ammann et al. [6], and Garcia-Molina and Salem [21]. Garcia-Molina and Salem [22] provide a good commentary on the implications of memory resident database systems on concurrency control, commit processing, data indexing and so on. The ED model, in contrast, focuses on providing a framework for flexible durability within the transactional model.

A recent consequence of attempting to manage durability trade-offs has been the development of NoSQL databases. With the dot-com boom [43], massive growth in internet adoption and technology-based companies in the 2000s, the amount of data that needed managing shot up significantly. Traditional OLTP[34] databases like PostgreSQL [31], MySQL[46] and Microsoft's SQL Server[1] were not built to scale horizontally and keep up with the high throughput requirements of these newly developed applications. Almost every major technology company – Google, Facebook, Microsoft and so on, started building their own systems to address their needs. In doing so, they deviated heavily from the principles and guarantees of the relational database world. Not only did these systems break away from the relational concept by dropping the use of tables, but they also let go of ACID guarantees[32] as they saw fit. Some systems completely moved away from transactions and transaction-related guarantees, while others watered down these guarantees or handled them poorly.

As these systems relaxed strong guarantees and semantics of relational database systems, it gave rise to a new set of problems like lost writes [51] and data consistency issues[39]. C. Mohan provides a trenchant commentary on these issues[44] and brings up important questions about design decisions that need answering before one is sold into the idea of using a NoSQL database for their next application. Among the many instances where people realised that tearing down decades-old, well-thought-out mechanisms for transactions, recovery, concurrency, and durability is not a good idea, a notable instance was when Facebook engineers came to terms with the fact that the semantics of eventual consistency in Cassandra were too painful to deal with and hence switched over to HBase[4].

Many of these NoSQL databases, which let go of strongly defined semantics and guarantees of relational database systems, started adding back features on an ad-hoc basis to compensate for the lack of a formal framework that provided these guarantees. For example, MongoDB introduced document joins, much like relational table joins, despite initially distancing itself from the relational model. Furthermore, as these ad-hoc features became unruly, NoSQL database developers decided to reintroduce formal guarantees that were relaxed earlier. An example was when MongoDB added support for transactions and ACID guarantees[37]. Of course, many of these new developments have emerged not just to overcome durability costs but also to address other issues like scalability and reduced flexibility with structured data. Nevertheless, addressing durability costs continues to remain a paramount concern for several systems.

5.0.2 Relaxed Durability

As far as we have seen, the work closest to Eventual Durability has been "Weakly Durable Transactions" introduced by Chang et al. [15]. This paper discusses decoupling a transaction's commit point from its durability. It also supports strongly and weakly durable transactions – similar to the ED model's fast and safe transactions. The authors argue that ad-hoc durability mechanisms can cause external and internal data inconsistencies in databases and attempt to provide a theoretical grounding to safely relaxing durability. Following this, they argue that the serialisability and consistency guarantees upheld in ACID systems must also be upheld in ACID$^-$ (weakly durable) systems. However, their reasoning around recoverability may not fully hold in certain cases. Eventual Durability is very similar to weakly durable transactions, but we take the idea further by formally extending Bernstein et al.'s [11] classical definition of serialisability and recoverability under ED. Furthermore, we also lay down a concrete set of properties offered by fast and safe transactions to enable applications to understand the benefits and risks of making durability decisions. Finally, we provide early visibility of transactions to reduce contention and improve throughput.

Some other recent works that have recognised high costs of durability in transactional systems and offered workarounds have been Prasaad et al.'s "Concurrent Prefix Recovery (CPR)" [50], and Li et al.'s "Distributed Prefix Recovery" [42] that extends CPR to a

distributed setting. CPR introduces a high-performance recovery mechanism by combining the semantics of asynchronous checkpoints and WAL group commit. Commit boundaries of the transaction log across all running threads are decided by the system rather than the user. The user is then notified of this point which they can use to prune their in-flight operations log.

Li et al. [41] introduce the idea of RedBlue consistency which allows users to issue fast (or blue) operations that execute locally and are lazily replicated; alongside slow (or red) operations that provide strong consistency and serialisability guarantees. If a group of operations issued by a client contained only blue operations, the system provides eventual consistency semantics, whereas if all operations were red, then the system provides serialisability. While this idea, to some extent, is similar to the ED model's fast and safe transactions, it imposes significant complexity since all blue operations have to be globally commutative with both red and blue operations. Because of this complication, marking operations as red or blue is not a trivial decision. The ED model is considerably simpler and provides greater flexibility to users.

The next class of approaches to dealing with durability primarily advocate for some kind of asynchronous processing of transactions – whether locally or in a replicated setting. Some of the most notable and earlier works in this list are [33][28][49][13] and [40]. Eventual Durability differs from these approaches in that it advocates against dealing with durability outside the scope of the transaction model. Interestingly, in this family of lazy evaluations, one approach stands out and warrants further discussion. Faleiro et al. [20] introduce the idea of "Lazy Transactions", which borrows the idea of lazy evaluations from programming languages research. Traditionally, database systems accept transactional operations, evaluate them, and decide whether to commit or abort them. This approach, on the other hand, first decides whether to commit or abort a transaction and makes a 'promise' to the user. This promise is only fulfilled once a reader actually tries to read the values changed by the transaction. So, a transaction executes in two phases, namely, the `now-phase` and the `later-phase`. In the `now-phase`, the transaction's read and write set is determined, serialisation checks are carried out and 'stickies' are inserted over records indicating that they have pending updates. Clients can demarcate `now-phase`s and `later-phase`s. If a transaction does not have any `now-phase`, it is executed as an eager transaction similar to a regular transaction. But, if it contains a `now-phase`, it is treated as a lazy transaction. This is an interesting approach that provides fine-grained flexibility to users to decide which operations should be lazy and which ones should be executed eagerly. However, it significantly increases the burden on the user to choose an appropriate point for `now-phase` so that it does not lead to an abort decision and still yields throughput benefits. In contrast, the ED model is much simpler for clients – they simply need to tag transactions as 'fast' or 'safe'. Lazy replication also reduces a transaction's contention footprint resulting in a lower contention rate among transactions.

Another class of studies have looked at releasing locks early to reduce contention and improve transaction throughput. The key insight is that transactions hold on to locks for much longer than they actually need them – especially since they spend the majority of

their time becoming durable. Interestingly, DeWitt et al. [18] first propose the idea of releasing transaction locks early, which is implemented in IMS/VS [23]. Recently, Johnson et al. [38] and Graefe et al. [24] implemented early lock release in Shore-MT and Foster B-Trees, respectively. However, in some cases, early lock release techniques may produce wrong results and fail to honour commit dependencies. Graefe et al. [25] propose the idea of controlled lock violation. They observe that a transaction spends most of its time becoming durable and that they need not hold on to their locks for their full lifetime. However, instead of releasing the locks early, concurrent transactions are allowed to selectively 'violate' these locks. The latter transaction will then incur a commit dependency on the former if the lock was an exclusive one. If the second transaction were read-only, it would wait for the earlier transaction to be flushed to disk.

This approach is similar to the ED model on several counts. Firstly, the ED model allows for early lock release without causing serialisation anomalies. In our ED implementation in Postgres, a safe transaction releases all row locks at commit time but holds on to heavyweight locks until it becomes durable. Changing the visibility of transactions to follow the ED model had the fortuitous benefit of early lock release, thus yielding significantly reduced contention and increased throughput. Secondly, safe, read-only transactions in ED incur a commit dependency on transactions from which it read and, therefore, waits for them to become durable – similar to how read-only, lock-violating transactions incur commit dependencies.

Chapter 6

Conclusions and Future Work

6.0.1 Conclusions

In this work, we present the idea of Eventual Durability to provide a principled framework for clients to make fine-grained decisions about transaction durability trade-offs. Furthermore, we formally extend the traditional transactional model and redefine serialisability and recoverability under ED. We then discuss implementing ED in Postgres, and show that ED reduces abort rates by an average of 91.25% – 93% and increases throughput by an average of 75% compared to baseline Postgres.

6.0.2 Future Work

PostgreSQL provides a fairly amenable test bed to implement Eventual Durability. Postgres has a single `XLOG` which gets replicated to standby systems; `LSN` values are monotonically increasing, and the durability of an `XLOG` record with an `LSN` implies durability of all `XLOG` records with smaller `LSN`s. Therefore, the changes necessitated by the ED implementation in Postgres are not nearly as complex as what is potentially needed for a complex system like CockroachDB that implements distributed transactions.

Our next step, therefore, is to implement ED in systems like CockroachDB[55], TiKV[36], FoundationDB[56], and other similar systems that support distributed, ACID transactions. The transactions in such systems might span multiple transaction logs, and the implementation of *fast* and *safe* transactions might involve significant changes to their respective transaction processing modules.

References

[1] SQL Server 2019. https://www.microsoft.com/en-ca/sql-server/sql-server-2019.

[2] pgbench - A Benchmarking Tool for PostgreSQL, Feb 2023. The PostgreSQL Global Development Group https://www.postgresql.org/docs/current/pgbench.html.

[3] PostgreSQL Isolation Levels, May 2023. https://www.postgresql.org/docs/current/transaction-iso.html.

[4] Amitanand S Aiyer, Mikhail Bautin, Guoqiang Jerry Chen, Pritam Damania, Prakash Khemani, Kannan Muthukkaruppan, Karthik Ranganathan, Nicolas Spiegelberg, Liyin Tang, and Madhuwanti Vaidya. Storage infrastructure behind Facebook messages: Using HBase at scale. *IEEE Data Eng. Bull.*, 35(2):4–13, 2012.

[5] Jonathan Allen. Jepsen disputes MongoDB's data consistency claims, May 2020.

[6] Arthur C Ammann, Maria Hanrahan, and Ravi Krishnamurthy. Design of a Memory Resident DBMS. In *Compcon*, pages 54–58, 1985.

[7] V Angkanawaraphan and A Pavlo. Auctionmark: A Benchmark for High-Performance OLTP Systems.

[8] Jobin Augustine. Postgresql synchronous_commit options and synchronous standby replication, Aug 2020. https://www.percona.com/blog/postgresql-synchronous_commit-options-and-synchronous-standby-replication/.

[9] David Axmark and Michael Widenius. MySQL introduction. *Linux journal*, 1999(67es):5–es, 1999.

[10] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[11] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley Reading, 1987.

[12] Alex Brasetvik. Elasticsearch from the bottom up, Part 1, Sep 2021. https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up.

[13] Yuri Breitbart, Raghavan Komondoor, Rajeev Rastogi, S Seshadri, and Avi Silberschatz. Update propagation protocols for replicated databates. In *Proceedings of the 1999 ACM SIGMOD international Conference on Management of Data*, pages 97–108, 1999.

[14] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

[15] Yun-Sheng Chang, Yu-Fang Chen, and Hsiang-Shang Ko. Weakly Durable High-Performance Transactions. *arXiv preprint arXiv:2110.01465*, 2021.

[16] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[17] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987.

[18] 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.

[19] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[20] Jose M Faleiro, Alexander Thomson, and Daniel J Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 15–26, 2014.

[21] Hector Garcia-Molina and Kenneth Salem. High performance transaction processing with memory resident data. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, 1987.

[22] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on knowledge and data engineering*, 4(6):509–516, 1992.

[23] Dieter Gawlick and David Kinkade. Varieties of concurrency control in IMS/VS fast path. *IEEE Database Eng. Bull.*, 8(2):3–10, 1985.

[24] Goetz Graefe, Hideaki Kimura, and Harumi Kuno. Foster b-trees. *ACM Transactions on Database Systems (TODS)*, 37(3):1–29, 2012.

[25] 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.

[26] Jim Gray. A Transaction Model. In *International Colloquium on Automata, Languages, and Programming*, pages 282–298. Springer, 1980.

[27] Jim Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154, 1981.

[28] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data*, pages 173–182, 1996.

[29] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[30] Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[31] PostgreSQL Global Development Group, Dec 2022. https://www.postgresql.org.

[32] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.

[33] Theo Haerder and Kurt Rothermel. Concepts for transaction recovery in nested transactions. *ACM Sigmod Record*, 16(3):239–248, 1987.

[34] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 409–439. 2018.

[35] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[36] 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.

[37] MongoDB Inc. https://www.mongodb.com/press/mongodb-announces-multi-document-acid-transactions-in-release-40.

[38] 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.

[39] Kyle Kingsbury. MongoDB Jepsen Report. http://jepsen.io/analyses/mongodb-4.2.6.

[40] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

[41] 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.

[42] 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.

[43] Alexander Ljungqvist and William J Wilhelm Jr. IPO pricing in the dot-com bubble. *The Journal of Finance*, 58(2):723–752, 2003.

[44] C Mohan. History Repeats Itself: Sensible and NonsenSQL aspects of the NoSQL hoopla. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 11–16, 2013.

[45] Kannan Muthukkaruppan. Storage infrastructure behind Facebook messages. In *Proceedings of International Workshop on High Performance Transaction Systems (HPTS'11)*, 2011.

[46] MySQL. https://mysql.com.

[47] Peter Norvig. Teach Yourself Programming in Ten Years. http://norvig.com/21-days.html#answers.

[48] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*, volume 2. Springer, 1999.

[49] Esther Pacitti, Pascale Minet, and Eric Simon. *Fast algorithms for maintaining replica consistency in lazy master replicated databases*. PhD thesis, INRIA, 1999.

[50] 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.

[51] John Schulz. The things I hate about Apache Cassandra, Jun 2020. https://blog.pythian.com/the-things-i-hate-about-apache-cassandra/.

[52] Silberschatz, Abraham and Korth, Henry F. and Sudarshan, S. and Alagiannis (Swisscom AG), Ioannis and Borovica-Gajic (University of Melbourne, AU), Renata. *Chapter 32: PostgreSQL*, page 1–57 (Online). McGraw-Hill Education, 7th edition, 2019.

[53] Jean da Silva. Pros and cons: When you should and should not use MongoDB, Aug 2021. https://www.percona.com/blog/pros-and-cons-when-you-should-and-should-not-use-mongodb/.

[54] Michael Stonebraker and Lawrence A Rowe. The design of Postgres. *ACM Sigmod Record*, 15(2):340–355, 1986.

[55] 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.

[56] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2653–2666, 2021.