Eventual Durability

Tejasvi Kashi
Kenneth Salem
University of Waterloo
Waterloo, Ontario, Canada
tejasvi.kashi@uwaterloo.ca,ken.salem@uwaterloo.ca

Jaemyung Kim
Khuzaima Daudjee
University of Waterloo
Waterloo, Ontario, Canada
jaekim.uw@gmail.com,kdaudjee@uwaterloo.ca

ABSTRACT

For latency-critical transactional applications, durability is often what limits performance. That is, executing transactions is fast, but guaranteeing that they are durable is slow. As a result, most of each transaction's latency is attributable to durability. To address this problem, some database systems allow applications to sacrifice durability guarantees in exchange for lower transaction latencies. These ad hoc techniques are effective, but they can make it difficult for applications to understand and manage the risks associated with failures.

In this paper, our goal is to offer a more principled foundation for these kinds of performance/durability tradeoffs. The major obstacle to doing this is the transaction model itself, because it couples transaction durability with transaction commit. That is, the model defines a single point at which a transaction becomes visible and durable. This forces all transaction guarantees to wait for the slowest one, which is often durability.

The primary contribution of this work is a new *eventually durable* transaction model, which decouples commit from durability. Transactions commit first, and become durable later. We argue for making this model the basis of the contract between transactional data systems and applications. We describe what it means to correctly implement eventually durable transactions, and consider how they can be exposed to applications. We also describe a prototype implementation of eventual durability in PostgreSQL, and show that it enables applications to reduce transaction latencies while managing the durability risks.

PVLDB Reference Format:
Tejasvi Kashi, Kenneth Salem, Jaemyung Kim, and Khuzaima Daudjee. Eventual Durability. PVLDB, 17(13): 4733 - 4745, 2024.
doi:10.14778/3704965.3704979

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/pct960/postgres/tree/eventual-durability.

1 INTRODUCTION

Transactions are a fundamental and widely used abstraction for building reliable concurrent applications over databases. Transactions offer so-called ACID guarantees. ACID is a promise that transactions are atomic in the face of concurrency and failures, that they are isolated until committed, and that they are durable once committed.

For latency-critical applications, durability is often what limits performance. That is, executing the transaction is fast, but guaranteeing durability when the transaction commits takes a long time. As a result, most of each transaction's latency is attributable to achieving durability. We are not the first to make this observation [5, 11], but let's look at some more examples to make this more concrete.

First, consider high-performance in-memory transaction systems. Even ten years ago, SILO [28] was reportedly able to execute more than 20,000 TPC-C transactions per second on a single core. Since SILO's workers execute transactions sequentially, this implies that executing each TPC-C transaction requires only a few tens of *micro*seconds. However, ensuring that a transaction is durable (and highly available) may involve storing multiple copies of the transaction's effects off-box, ideally in failure-isolated locations. For example, Amazon's Aurora [29] replicates transaction logs six times, across three different availability zones. Network round trip times between availability zones are typically on the order of a few *milli*seconds. Thus, making such a fast transaction durable may be orders of magnitude slower than actually executing the transaction. Almost all of the transaction latency observed by the application is due to durability.

As another example, consider replicated geographically distributed database systems like CockroachDB [25] or Spanner [4]. If the system maintains a copy of the data locally, close to a client (which is typically the goal), executing transactions on behalf of that client can be very fast (sub-millisecond). But durably committing the transaction requires replicating its effects across geographically distributed regions, which introduces latencies in the tens or even hundreds of milliseconds. Again, transaction execution is much faster than durability.

As an aside, this was not always the case. Fifty years ago, when the transaction model was very young, database systems were typically centralized and databases were stored on slow disks. Executing a transaction involved multiple disk I/Os, and making a transaction durable just added one more disk I/O (for writing a log record). Thus, transactions were slow, and the marginal cost of making a transaction durable was relatively low. Over time, though, this situation has reversed. We have gotten much better at executing transactions quickly, while our standards for durability and availability have become more demanding.

How can we reduce transaction latencies for latency-sensitive transactional applications, in a world where latency is dominated by durability? This question is the focus of our paper.

Proceedings of the VLDB Endowment, Vol. 17, No. 13 ISSN 2150-8097.
doi:10.14778/3704965.3704979

One possible answer to this question is to *reduce the cost of durability* by taking advantage of fast networks and low-latency persistent storage systems. This can certainly help. However, to offer a strong durability (and availability) guarantee, we need to replicate transactions to *failure-isolated* locations. As a rule of thumb, the more failure-isolated the destination is, the slower we should expect replication to be. For geo-replication, the speed of light imposes a fundamental limit; we are never going to be able to quickly disaster-proof a transaction. Thus, we argue that making durability faster may help, but by itself it is not going to make this problem go away.

The other approach is to *sacrifice durability* in exchange for lower latency. This approach is common enough that widely used database systems can be configured to support it. For example, PostgreSQL can be configured to use *asynchronous commits*, which means that the database system acknowledges a transaction as committed *before* it is durable. That is, before the transaction's write-ahead log records are known to be written to persistent storage. Another example of this is lazy replication [12], in which transactions are allowed to commit quickly and then they are lazily replicated, after commit, for durability.

Database systems provide these options because latency-sensitive applications demand them. They are very effective at hiding durability latency but, of course, they introduce the risk that the effects of committed transactions will be lost. Because they move durability outside of the scope of the transaction model, they make it difficult for applications to understand (and manage) the resulting risk. For example, how does an application know which committed transactions are durable, or when they become durable? Can durable transactions depend on non-durable transactions? What happens when committed but not-yet-durable transactions fail? Although it is effective at reducing latency, moving durability outside of the scope of the transaction model amounts to throwing the baby out with the bathwater.

In this paper, our goal is to offer a more principled foundation for these kinds of performance/durability tradeoffs. Instead of moving transaction durability outside of the scope of the transaction model, we want to extend the model so that it can capture the tradeoff. By doing so, we hope to enable the same latency benefits that existing ad hoc techniques already provide, while at the same time making it easier for applications to understand and manage the durability risks that they introduce. We want different database systems to be able to offer latency/durability tradeoffs based on a common foundation, so that applications do not have to rely on ad hoc solutions.

We claim that *the major obstacle to supporting durability-aware applications is the transaction model itself.* The root problem with the transaction model is that it couples transaction durability with transaction commit. That is, there is a single commit point at which the transaction becomes visible and is guaranteed to be durable. In effect, this model forces *all* transaction guarantees to wait for the slowest guarantee, which is often durability.

The primary contribution of our work is a new transaction model that *decouples* transaction commit from transaction durability. Under this model, transactions commit and then gradually "harden" against failures *after* they have committed. All transactional guarantees except durability are made at the commit point. We refer to this as an *eventual durability (ED)* model, and we argue for making this model the basis of the "contract" between transactional data systems and applications.

Eventually durable data systems that implement the eventual durability model can manage and track transaction hardening. As we will illustrate, they can implement application-facing interfaces that enable *durability-aware applications* to selectively speculate on transactions' durability, when and if it makes sense to do so. Durability-aware applications can control when and where they are exposed to committed but not durable data, so that applications can reduce latencies while managing the "blast radius" of failures. We emphasize that real systems and applications already make these trade-offs, but in an ad hoc way, outside of the scope of the transaction model.

The remainder of this paper is organized as follows. First, in Section 2, we present several hypothetical examples of durability-aware applications, to illustrate how they could use an eventually durable database system to manage latency/durability tradeoffs. In Section 3 we present the eventual durability model in more detail. In Section 5 we define what it means for a database system to correctly execute eventually durable transactions. This essentially defines our proposed contract between an eventually durable database system and its applications. The ED model does not prescribe a particular application interface, and in Section 4 we describe a few examples of application interfaces that could be supported. Finally, to illustrate what is involved in building an ED database system, we built an ED version of PostgreSQL, called Pg-ED. Section 6 describes how we modified PostgreSQL to do this, and Section 7 presents the results of experiments comparing the performance of Pg-ED to the original, unmodified PostgreSQL.

2 DURABILITY-AWARE APPLICATIONS

Before diving into a presentation of the eventual durability model, we start with some discussion of how eventual durability could be exposed to *durability-aware applications*. Our goal is to illustrate how durability-aware applications could benefit from eventually durable transactions by making explicit trade-offs between performance and failure risks.

The eventual durability model does not prescribe a particular application interface, and in Section 4 we describe various ways that eventual durability can be exposed to applications. However, for the sake of the examples described here, we will assume a very simple interface that relies on transaction tagging. Specifically, whenever the application commits a new transaction, it may optionally tag that transaction as *fast*. We refer to transactions whose commits are not tagged as *safe*:

- *Safe transactions* behave like normal, classical transactions. When the database system acknowledges the commit of a safe transaction, the transaction is guaranteed to be durable and any transactions that it may have depended on are guaranteed to be durable as well.
- *Fast transactions* are able to commit quickly because they are not guaranteed to be durable when they commit. Fast transactions are visible when their commit is acknowledged, but they eventually become durable at some point *after* committing. It is possible that a fast transaction will *fail* after committing, instead of becoming durable. If this occurs, the

effects of the fast transaction are (atomically) lost from the database, even though those effects might have been seen by other transactions prior to the failure.

It is worth emphasizing that all transactions, whether fast or safe, have all of the other guarantees that are normally associated with transactions. They are atomic, and their effects remain isolated until they commit. Thus, there is no risk that the application will be exposed to data inconsistencies resulting from partial or incomplete transactions. All transactions are executed serializably as well. In Section 5, we will more precisely define what it means to execute transactions serializably under the eventual durability model.

By tagging a transaction as fast, a durability-aware application is accepting some failure risk in exchange for better performance. Durability-aware applications can do this at the granularity of individual transactions, so that they can control when they are willing to accept this risk. By tagging a transaction as fast, an application enables itself to *speculate* on the eventual durability of that transaction. That is, it can perform additional work, including executing additional transactions, without waiting for the fast transaction to become durable. In the likely case in which the fast transaction eventually becomes durable, the application's speculation is successful, and it has saved time. If the fast transaction fails to become durable and is lost, the application may lose its speculative work. If the application chose to expose any of that work, e.g., to end users, then it must be willing to accept the risk that that work might be lost.

2.1 Example: Programmatic Advertising

Next, we present a more concrete example to illustrate how a durability-aware application might use fast and safe transactions. Our example is a greatly simplified latency-sensitive ad exchange application, used for programmatic advertising. A *publisher* is generating a web page for an end user, and wants to quickly fill an advertising slot (e.g, a banner) on that web page with a targeted advertisement. To obtain an advertisement for its ad slot, the publisher sents a request to an *ad exchange*, which acts as an intermediary between the publisher and advertisers. When it receives the publisher's request, the ad exchange runs an auction among the advertisers, who bid to have their ad displayed to the end user in the publisher's ad slot.

The ad exchange maintains a database of auctions and bids. When it receives the ad request from the publisher, it uses the following workflow:

(1) It runs a CreateAuction transaction to create a new auction record in its database, identifying the publisher whose request triggered the auction.
(2) It broadcasts a request for bids to potential advertisers. Advertisers evaluate the bid opportunity and respond to the ad exchange with a bid if they are interested in placing an ad.
(3) Each time it receives an advertiser's bid, the ad exchange runs a NewBid transaction to record the bid in its database, associating the bid with the publisher's auction.
(4) At some point, the ad exchange will run a CloseAuction transaction to mark the auction closed, and choose and record a winning bid from among those that it received.

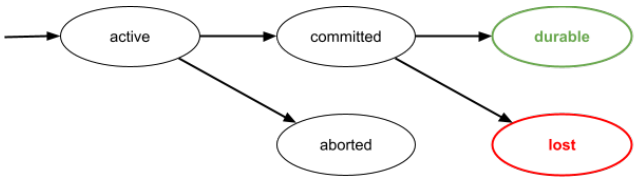


Figure 1: Lifecycle of Eventually Durable Transactions

(5) Finally, the ad exchange notifies the publisher of the winning advertisement, and the publisher includes the ad on the web page it is generating.

Auctions and bids recorded in the ad exchange database are later used to bill advertisers, pay publishers, and provide analytics to both.

To ensure timely display of the publisher's web page, this workflow, from receipt of the publisher's request to notifying it of the winning advertisement, operates under a tight latency budget. The workflow's critical path includes the CreateAuction, NewBid, and CloseAuction transactions, so the ad exchange is motivated to execute them as quickly as possible.

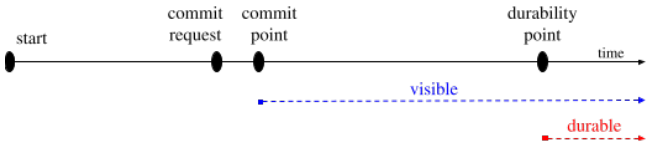
Let us suppose that the ad exchange chooses to run CreateAuction and NewBid as fast transactions, and CloseAuction as safe. This will eliminate much of the durability latency from the workflow's critical path. In particular, the ad exchange can proceed speculatively to request advertisers' bids before the newly created auction is durable, and bids will become visible quickly in the ad exchange database. Only the durabilty wait for CloseAuction remains. Until the auction closes, it is possible that failures will cause individual bids to be lost, potentially resulting in a sub-optimal bid winning the auction. It is also possible that the auction itself will be lost before it closes (if CreateAuction fails), which would prevent the ad exchange from providing an ad to the publisher. However, the ad exchange can be sure that if CloseAuction's commit is acknowledged, then both the auction and surviving bids will be durable. We emphasize that this kind of reasoning about latency and failure risks is based on the ED model itself, and not on specific implementation details or ad hoc mechanisms offered by a particular underlying database system.

The ED model helps the ad exchange reason about the risks it will be exposed to by using fast transactions. However, it is up to the ad exchange - not the model - to determine whether those risks are acceptable in exchange for the reduced latency offered by fast transactions. The application can find an acceptable balance by choosing which transactions will be fast and which will be safe. For example, the ad exchange could choose to run CreateAuction as safe rather than fast. This adds extra durability latency to the workflow, but it ensures that the auction will not fail once it has been created - though individual bids may be lost.

3 TRANSACTION MODEL

The important feature of the eventual durability transaction model is that it decouples transaction durability from transaction commit. Under the classical model, a transaction becomes visible atomically when it commits, and it is guaranteed to be durable. Aborted transactions have no effects.

(a) Classical Transaction Timeline



(b) Eventually Durable Transaction Timeline

Figure 2: Transaction Timeline Comparision

The eventual durability model adds two new transaction states to the classical model, as illustrated in Figure 1. Transactions become visible when they commit, as in the classical model, but they are *not durable* at commit time. In the happy case, a committed transaction eventually hardens (becomes durable) after it commits, and transitions to the new `durable` state. In the unhappy case, a failure might prevent a committed transaction from becoming durable. Such transactions transition to the new `lost` state, in which they are no longer visible to other transactions. Just as transactions become visible atomically when they commit, they become *invisible* atomically when they are lost.

The advantage of this model is that it allows transactions to commit and become visible faster, since commit does not require durability. Figure 2a illustrates a transaction execution timeline under the classical transaction model. The transaction's commit point does not occur until the transaction is durable. Since the commit request is not acknowledged until after the commit point, the acknowledgement must wait for the transaction to become durable. Figure 2b illustrates a timeline of the same transaction under the eventual durability model. The transaction reaches its commit point and becomes visible quickly, without waiting for durability. Later, the transaction finishes hardening and reaches its durability point.

Neither transaction model specifies precisely what "durability" means. In the classical model, where durability is tied to transaction commit, durability is a promise that a transaction will remain visible beyond its commit point, even if certain types of tolerated failures occur. Different types of systems are designed to tolerate different types of failures. In the eventual durability model, durability is a promise that an already-committed transaction will remain visible, despite failures. That is, it is a promise that a committed transaction will never transition into the "lost" state.

As noted in Section 1, committing transactions before they are durable is not a new idea. Lazy replication, asynchronous commits, and other performance optimizations all sacrifice durability for improved performance. What we are trying to do with eventual durability is to put such techniques on a firmer foundation by building the idea of at-risk transactions into the transaction model itself. This is important because it is the transaction model that forms the basis of the behavioral "contract" between the database system and its applications. For example, it allows the database system to define precisely what happens when a committed transaction fails, and how other transactions are affected by that failure, and whether and how to expose these events to the application. This, in turn, gives applications a basis for understanding and managing the risks that come with speculation.

4 APPLICATION INTERFACE

The eventual durability model defines separate commit and durability points for each transaction. However, it does not specify whether, when, or how to expose those two points to applications.

In Section 2, we described one possible application interface, which distinguishes *fast* and *safe* transactions. In terms of the ED model illustrated in Figure 2b, these two types of transaction behave as follows. A commit request for a fast transaction is acknowledged as soon as the transaction reaches its commit point. Thus, fast transactions are visible when they are acknowledged, but they are not yet durable. In contrast, the commit request for a safe transactions is acknowledged after the transaction reaches its *durability* point. Thus, like classical transactions, safe transactions will never be lost once they have been acknowledged. This simple fast/safe interface allows an application to trade off durability for performance at the granularity of individual transactions.

Alternatively, an eventually durable data system could acknowledge *all* transaction commit requests immediately after the commit point, without waiting for durability. In addition, it could provide a separate *sync* operation to allow the application to wait for durability when necessary. For example, a connection-level *sync* operation would block until all preceding transactions committed on that database connection have reached their durability points. This is similar to the behavior of file systems which defer durability guarantees on file writes until either the file is closed or an explicit sync operation is performed. We discuss more examples of this kind of "bulk" durability acknowledgement in Section 8.

Another interface design decision is whether to provide the application with an explicit signal when committed transactions fail and are lost. Under the eventual durability model, the loss of a committed transaction is an asynchronous event, triggered by a failure, which may occur after the transaction commit has been acknowledged to the application. One interface option is to provide an explicit callback to the application for each transaction, at the point where it becomes durable or is lost. Alternatively, assuming that failures are uncommon, the system could call back to the application only in response to lost transactions. The simple fast/safe interface we described in Section 2 provides no such callbacks. Transaction loss is manifest only by the atomic removal of the lost transaction's effects from the database, which can be discovered on subsequent transactions' database reads.

5 CORRECTNESS

In this section, we will consider what it means for a database system to correctly execute a set of eventually durable transactions. We consider two well-known properties of transaction executions, *serializability* and *recoverability*, and consider how to adapt them for eventually durable transactions.

5.1 Serializability Under Eventual Durability

Serializability and other consistency guarantees constrain what each transaction should see when it reads the database. Serializability requires that all committed transactions appear to execute sequentially, in some order. Thus, a transaction T that reads an item x should read the value of x that was written by the latest committed transaction preceding it in the serialization order.

Eventual durability complicates this picture, because it introduces the possibility that a committed transaction might be lost at some point after it commits. Such a transaction is visible for a time, but then ceases to be visible. Thus, in our example, we need to define what T should read if the latest committed transaction preceding it in the serialization order has been lost.

To define serializability, we need to model executions of eventually durable transactions. We do this by extending the model defined by Bernstein, Hadzilacos, and Goodman [2], which we refer to as the *classical* model. In the classical model, a transaction consists of a sequence[1] of read and write operations, followed by either a commit event (c) or an abort event (a). To model eventually durable (ED) transactions, we add two new events: durability (d), which represents the point at which a committed transaction becomes durable, and failure (f), which represents the point at which a committed transaction is lost. A transaction may include either a d or f, but not both, and these events occur after the transaction commits. (Aborted transactions do not have durability or failure events.) Here are examples of ED transaction executions:

$$T_1 = r_1[x]\ w_1[x]\ c_1\ f_1$$
$$T_2 = r_2[x]\ w_2[x]\ c_2\ d_2$$

Transaction T_1 reads and writes x and then commits, but ultimately fails and does not become durable. T_2 reads and writes x and commits, and eventually becomes durable.

An ED execution history of a set of ED transactions represents the order in which the transactions' operations occur during execution. It is an interleaving of the transactions' operations, preserving the operation order of each individual transaction. Here is an example of a possible execution history of the two transactions above:

$$H_a = r_1[x_0]\ w_1[x]\ c_1\ r_2[x_1]\ w_2[x]\ c_2\ f_1\ d_2$$

Note that we annotate the read operations to indicate which version of the object is being read, e.g., $r_2[x_1]$ indicates that T_2 reads the version of x that was written by T_1

We want to define whether a given ED history is or is not serializable. In the classical model, a history H is serializable if its *committed projection*, $C(H)$, is equivalent to a serial execution. The committed projection includes all of the operations of transactions for which there is a commit event in the history. For ED histories, we need to tweak this definition to account for the fact that committed transactions may fail. To construct the committed projection $C(H)$ of an ED history H, we eliminate the operations of all aborted transactions, as we would under the classical model. *In addition, we eliminate the operations of all failed transactions, as well as all d operations.* For example, the committed projection of H_a is

$$C(H_a) = r_2[x_1]\ w_2[x]\ c_2$$

It does not include any of the operations of T_1, which failed, and it does not include T_2's durability event.

With this re-definition of committed projection in hand, we define serializability for ED histories by appealing to the classical serializability definition:

Definition 5.1. *An eventually durable history H is serializable if, for all prefixes H' of H, $C(H')$ is serializable in the classical sense.*

This is well-defined because the committed projection of an ED history is a classical history, i.e., it contains neither durability events nor failure events. Essentially, this definition requires that at all times, the history is equivalent to executing the transactions that have not (yet) failed in some serial order. As an example, consider the history H_b:

$$H_b = r_1[x_0]\ r_2[x_0]\ w_1[x]\ c_1\ w_2[x]\ c_2\ f_1\ f_2$$

H_b is not ED serializable because the prefix of H_b without the two failure operations is not serializable in the classical sense.

Because we define ED serializability by appealing to classical serializability, we can define other consistency guarantees for ED histories using a similar approach. For example, we can define an ED history to be *ED snapshot isolated* if the committed projections of its prefixes are snapshot isolated in the classical sense. This is in keeping with our intention that eventual durability is largely orthogonal to consistency guarantees. Consistency guarantees define *what* an application should see when it reads the database. Eventual durability, on the other hand, only affects *when* the application can see it.

5.2 Recoverability Under Eventual Durability

In classical histories, recoverability is about ensuring that the database system is in a position to erase the effects of aborted transactions. Classical recoverability demands that any transaction that has read from an in-flight transaction not be allowed to commit. If it were to commit and the in-flight transaction were then to abort, the database system would be stuck. Thus, any classical system that aims to ensure atomic transaction execution should ensure recoverability.

As was the case for serializability, recoverability is a bit more complicated for ED histories, because transactions may fail *after* committing. In addition to being in a position to erase the effects of aborted transactions (like a classical system), *an ED system must also be in a position to erase the effects of committed transactions that subsequently fail.* To account for this, we need to extend the definition of recoverability for ED histories.

Definition 5.2 (ED Recoverability). *An ED history H is recoverable if* both *of the following conditions hold:*

(1) *If transaction T_2 reads from transaction T_1 in H, then T_2 does not commit before T_1 commits.*
(2) *If transaction T_2 reads from transaction 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 precedes d_2.*

[1]The classical model allows operations to be only partially ordered, but we'll use total orders here for ease of presentation

Just as any classical system should ensure that its schedules are recoverable, any ED system should ensure that its schedules are ED Recoverable.

The first ED Recoverability 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, and which therefore might abort.

The second ED Recoverability condition ensures that the system is prepared to erase the effects of committed transactions that subsequently fail. Consider the following history, in which T_2 has read from T_1:

$$H_b = w_1[x]\ r_2[x_1]\ w_2[x_2]\ c_1\ c_2\ d_2$$

H_b is a partial history, since T_1 is not yet finished: it is committed, but it is neither durable nor failed. It is ED Serializable, since it is equivalent to executing T_1 followed by T_2. However, it is *not* ED Recoverable, because it violates the second ED Recoverability condition: T_2 reads from T_1, but T_2 has become durable though T_1 has not. If T_1 were to fail at this point, like this:

$$H_c = w_1[x]\ r_2[x_1]\ w_2[x_2]\ c_1\ c_2\ d_2\ f_1$$

then the ED transaction system would be "stuck". T_1 became visible when it committed, and T_2 saw its effects. When T_1 fails, T_2 should not be allowed to become durable, since it depends on T_1. However, in H_c, T_2 is *already* durable. Essentially, ED Recoverability avoids this kind of situation by demanding that transactions become durable only after the transactions they depend on are durable.

5.3 Read-Only Transactions

We conclude this section with a brief discussion of read-only transactions. Like all ED transactions, read-only ED transactions eventually either fail or become durable after commit. 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 serializability, applications are guaranteed that a committed read-only transaction has seen a serializable 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 5.2 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 enforcing ED Recoverability for all transactions, including read-only transactions, an ED transaction system offers applications the ability to control the risks associated with reading non-durable data. For example, under the simple fast/safe transaction interface we introduced in Section 2, fast transactions are acknowledged as soon as they commit, while safe transactions are not acknowledged until they have reached their durability point. Thus, an application can use fast read-only transactions to quickly obtain a serializable view of the database, which may include not-yet-durable data. Alternatively, by using a safe read-only transaction and waiting for its commit to be acknowledged, an application can ensure that the data it has read are durable.

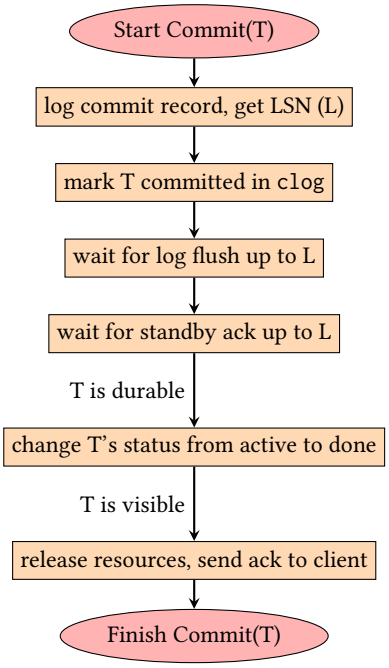


Figure 3: Committing a Transaction (T) in PostgreSQL

6 EVENTUAL DURABILITY PROTOTYPE

To illustrate the impact of eventual durability on database systems, we developed an eventually durable variant of PostgreSQL, which we will refer to as PG-ED. We chose PostgreSQL as our starting point because it is widely used, and because it offers some flexibility (via configuration options) in the way that transactions are committed. We begin by providing a brief overview of transactions in PostgreSQL, with a focus on how PostgreSQL makes transactions durable and visible. Suzuki [23] offers a more detailed and thorough description. We then describe the changes that we made to convert PostgreSQL to PG-ED.

6.1 PostgreSQL Transaction Overview

When PostgreSQL client initiates a connection, PostgreSQL's supervisor process spawns and assigns a backend process for that client. PostgreSQL assigns each update transaction a unique identifier, called an `xid`. PostgreSQL maintains a list of backend processes, and each process records the `xid` of the transaction it is currently executing. By scanning the server process list, PostgreSQL can identify the current set of *active* `xid`s, i.e., those update transactions that are currently in progress.

When clients make update requests, the backend process updates the database and records the update on a write-ahead log, called the *xlog*, which is shared by all backend processes. When the client

asks to commit its current transaction, the backend process inserts a commit record for the transaction in the xlog. Each record in the xlog is assigned a *log sequence number (LSN)*. As the name suggests, LSNs increase monotonically as new log records are inserted. This means that each transaction's commit record will have a larger LSN than that transaction's update records, since the commit record is logged last. After logging a transaction's commit record, the server marks the transaction as committed in another structure, called the *clog*. Figure 3 summarizes the commit flow in PostgreSQL.

6.1.1 Transaction Durability. PostgreSQL makes transactions durable using the write-ahead log. It ensures that a transaction is durable by waiting until all log records up through the transaction's commit record's LSN have been flushed to persistent storage before making the transaction's updates visible and acknowledging the transaction's commit. Since LSNs increase monotonically, a transaction's commit record will have a higher LSN than all other log records written for that transaction.

PostgreSQL can also be deployed in a high-availability (HA) configuration, with a primary server and a standby. The standby holds a copy of the database and the log, and clients can fail over to the standby if the primary is lost. In this HA configuration, the primary server streams log records, in LSN order, to the standby, which stores them persistently and then acknowledges receipt to the primary. PostgreSQL waits until a transaction's log records are persistently stored at both the primary and standby sites before considering the transaction to be committed.

6.1.2 Transaction Visibility. PostgreSQL's database is a multi-version row-store, and each new record (row) version records two `xids`, called `xmin` and `xmax`. `xmin` records the `xid` of the transaction that created it. `xmax` records the `xid` of the transaction that deletes that version, or replaces it with an updated version. Each PostgreSQL transaction reads from a logical snapshot of the database that is defined to include all and only the updates that were committed before the transaction starts.[2] In order for T's updates to be included in subsequent transactions' snapshots, i.e., in order for T to be *visible*, two conditions have to be satisfied. First, T must be marked as committed in the `clog`. Second, T must no longer be active. As illustrated in Figure 3, a transaction remains active until its log records are durably stored. Thus, this second condition ensures that only durably committed transactions are visible.

6.2 Implementing Eventual Durability

To expose fast and safe transactions to PG-ED clients, we overload PostgreSQL's existing `synchronous_commit` parameter, which controls when PostgreSQL acknowledges transaction commits. A transaction that is committed with `synchronous_commit = OFF` is handled by PG-ED as a *fast transaction*. Fast transactions become visible and are acknowledged as soon as they are committed, without waiting for durability. A transaction that is committed with `synchronous_commit = ON` is handled by PG-ED as a *safe transaction*. It becomes visible as soon as it commits, but it is not acknowledged to the client until it is durable.

[2]This assumes that PostgreSQL is running with a transaction isolation level of Repeatable Read or higher.

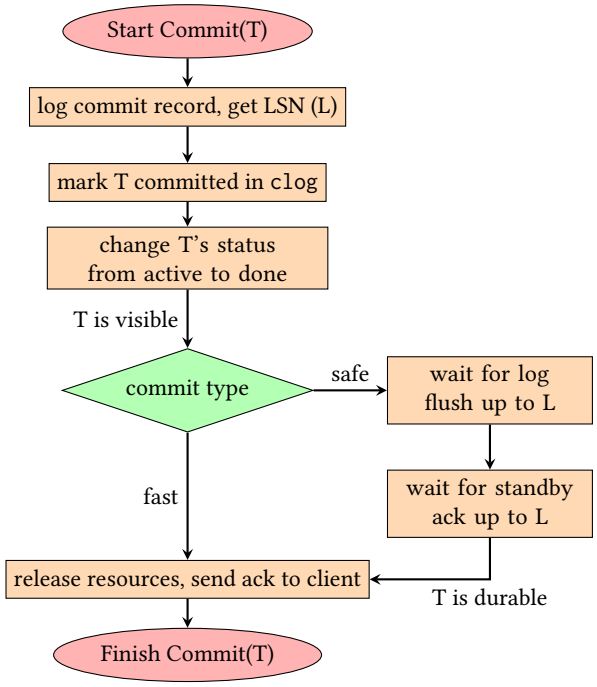


Figure 4: Transaction Commit Flowchart for PG-ED

To support eventual durability in PG-ED, we need to address two problems. The first is providing *early visibility*. Under the ED model, all transactions, whether fast or safe, should become visible as soon as they commit, without waiting for durability. Second, we need to enforce ED Recoverability (§ 5.2). In the remainder of this section, we describe how PG-ED addresses these two problems.

6.2.1 Early Visiblity. Figure 4 illustrates the commit procedure in PG-ED, for both safe and fast transactions. To implement early visibility for transaction T, we changed the commit flow so that each transaction T is marked as "done" immediately after the commit decision is recorded in the `clog`, without waiting for durability. As a result, T's database updates will be included in the read snapshots of any subsequent transactions, though T might not be durable.

One limitation of our prototype is that certain database resources, such as table-level locks, that may be associated with a safe transaction T are held until T becomes durable, as shown in Figure 4. This conservative design decision was made for simplicity. It does not directly impact transaction latencies, but it does miss some opportunity for throughput optimization, as we show later, in § 7.4.

6.2.2 Enforcing Recoverabilty. ED Recoverabilty (§ 5.2) imposes two conditions on PG-ED. First, no transaction should commit until the transactions it has read from have committed. Second, no transaction's durability point should precede the durability points of the transactions it has read from. The second condition is important because PG-ED may not acknowledge a safe transaction's commit prior to its durability point.

The first condition is enforced by PostgreSQL's existing snapshot mechanism, which is maintained in PG-ED. Every transaction T reads from a database snapshot that (a) is defined before T's first

read, and (b) only includes the effects of transactions that were committed before the snapshot was defined. Thus, PG-ED transactions read only committed (though not necessarily durable) updates.

Enforcement of the second ED Recoverabilty condition is simplified by the fact that transactions become durable in order of their commit LSNs. This means that if T is a read-write transaction that reads from T' and T's own updates are durable, then T' must also be durable, since T must commit after T' and thus must have a larger commit LSN than T'. Thus, to ensure that T is durable, it is sufficient for PG-ED to wait for T's own commit record to become durable. This is exactly what PG-ED does (Figure 4).

However, this argument does not hold if T is a read-only transaction. Read-only transactions do not produce log entries, so T will not have a commit LSN to wait on before it can be declared durable. If T reads from T', ED Recoverability demands that PG-ED wait for T' to be durable before T can be considered durable.

To handle read-only transactions, we introduced two new mechanisms to PG-ED. The first is an in-memory structure that records the `xid` and commit LSN of each non-durable update transaction. Transactions are added to the structure when they commit, and removed from the structure once the system's durable LSN high-water-mark has advanced past the transaction's commit LSN. The second mechanism tracks transactions' direct read dependencies. When transaction T reads a tuple, PG-ED records the `xid` of the transaction that created the tuple as a direct read dependency of T. When T commits, and if it is read-only, PG-ED uses the non-durable transaction structure to check whether any of T's direct read dependencies are non-durable. If any are, T's commit acknowledgement is delayed until all have become durable. Since PG-ED transactions become durable in commit LSN order, this delay is sufficient to ensure that T's *indirect* read dependencies will be durable as well.

This mechanism ensures that commit acknowledgements are delayed only for safe read-only transactions that have read recently updated (and hence not yet durable) data. However, read dependency tracking does add some overhead to transaction processing. PG-ED tracks read dependencies for *all* transactions, not just read-only transactions. This is because PG-ED may not know whether a transaction will be read-only when it starts reading. Read dependencies tracked for update transactions are simply ignored at commit time.

7 EVALUATION

We designed a series of microbenchmarks with which we characterize the impact of eventual durability on transaction latency and throughput by comparing PG-ED against baseline PostgreSQL. In addition, we explored the impact of eventual durability in a more realistic setting, using the TPC-C benchmark.

7.1 Experimental Setup

Our experiments were run using Amazon Web Services (AWS). Unless otherwise specified, experiments were run in a high-availability configuration, with primary and standby servers running on `m5.large` instances, each having two Intel(R) Xeon(R) Platinum 8175M CPUs @ 2.50GHz, 8 GiB of memory, and an attached 100 GiB Elastic Block Storage (EBS) volume. The servers are configured to log to their EBS volumes. A third server, on an AWS `c5.4xlarge` instance, is used as a client node. The client node is only used for running test scripts and workloads against the PostgreSQL servers. We use `pgbench` [21] on the client node to generate workloads. All workloads were run at PostgreSQL's SERIALIZABLE isolation level.

The baseline PostgreSQL server used in our experiments is compiled from source, using the 15.1 release of PostgreSQL. PG-ED is also based on the 15.1 PostgreSQL release.

7.2 Impact of Durability on Latency

First, we present a simple experiment intended to show the impact of durability on transaction latency in our setting. The experiment uses a database with a single table containing two columns (integer keys and values) and one million rows. There is a single client which generates one transaction at a time, with no think time between successive transactions. Each transaction updates a value in a single row, selected uniformly at random. We run the client for two minutes, and measure and report the mean transaction latency observed by the client. We ran this experiment using several server configurations, which offer different levels of durability and availability:

non-durable: In this configuration, there is no standby server, and the primary is configured to use asynchronous commit. This means that the server acknowledges transaction commits to the client without waiting for the transaction's commit record to be written to the transaction log on EBS. In this configuration, committed transactions are not guaranteed to survive a restart of the PostgreSQL server or loss of the server instance. Thus, this experiment measures transaction latency without durability.

1AZ: The 1AZ configuration is identical to the non-durable configuration, except that the PostgreSQL server is configured with `synchronous_commit=local`. This means that the server waits for each transaction's commit record to be flushed to the transaction log on the server's attached EBS drive before acknowledging the transaction's commit to the client. Since the EBS volume is independent of the database server, durable transactions will survive a loss of the database server. Internally, EBS replicates writes within an availability zone (AZ) but not across AZs. Thus, transactions run in this configuration are *not* guaranteed to survive a failure of the server's AZ.

1Region: This configuration is identical to the 1AZ configuration, except that the standby server is located in a different AZ within the same AWS region as the primary. (The primary is located in `us-east-1a`, and the secondary in `us-east-1b`.) In this configuration, durable transactions are guaranteed to survive a failover to the standby server, even if the primary's AZ is lost. However, durable transactions are not guaranteed to survive a disaster that causes loss of the entire us-east region in AWS.

2Regions: Finally, we tested a disaster-tolerant 2Region configuration that is identical to the 1Region configuration except that the standby server is placed in a remote AWS region (`ca-central`). This ensures that durable transactions will even survive a failover to `ca-central` in the event of a complete loss of the `us-east` region.

In all configurations, the client is located in the same availability zone as the primary server (`us-east-1a`).

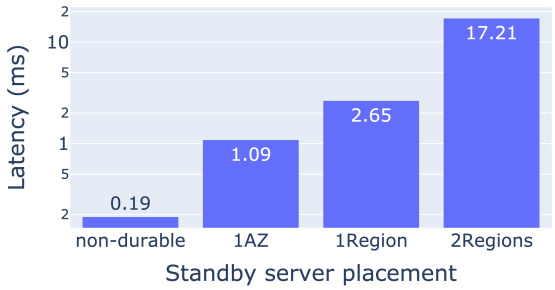


Figure 5: Impact of Durability on Transaction Latency

Figure 5 compares transaction latencies in each of these configurations. Transaction latency in the *1AZ* configuration is roughly 5x that of the non-durable baseline. Thus, roughly 80% of latency in the *1AZ* configuration is due to durability. If the system is configured with a standby in a different AZ (*1Region* configuration), which is common practice, more than 90% of transaction latency is attributable to durability. Replicating the transaction to a remote region can increase latency by another order of magnitude or more, depending on the distance to the remote.

7.3 ED Transaction Latency

Our next experiment shows the latency of fast and safe ED transactions in PG-ED. For this experiment, we use the same database as in Section 7.2. Four concurrent pgbench clients each send one transaction at a time, with no think time between requests. Two of the clients issue update transactions, each of which updates a single randomly selected row from the table. One of the two update clients issues fast transactions, and the other issues safe transactions. The two remaining clients issue read-only transactions, each of which reads a single randomly selected row from the table. Again, one of the read-only clients issues fast read-only transactions, while the other issues safe ones. Each experimental run lasts for 2 minutes, and we measure the mean client-side latency of each of the four types of transactions. We ran this experiment three times, using the 1AZ, 1Region, and 2Region server configurations, which offer successively stronger durability guarantees.

Figure 6a shows the latencies of fast and safe update transactions in PG-ED, as well as update transaction latency in baseline PostgreSQL, under all three configurations. As expected, the latency for safe update transactions in PG-ED is similar to that of baseline updates, since both provide the same durability guarantee. Fast update transactions offer much lower latency than safe transactions in PG-ED, because they avoid durability-related delays. The advantage increases for server configurations that offer stronger durability guarantees. PG-ED's fast update transactions have slightly higher latency than non-durable updates in baseline PostgreSQL. This difference is due to read dependency tracking in PG-ED. The advantage of using fast update transactions in PG-ED is that PG-ED is aware of when such transactions eventually become durable. This allows it, for example, to determine a durability point for read-only transactions, as discussed next.

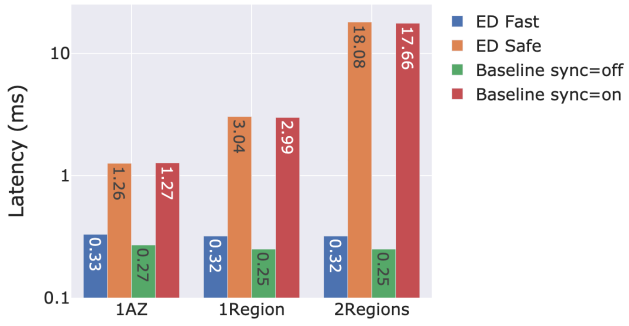
Figure 6b shows the latency of read-only transactions. Fast read-only transactions in PG-ED have latencies similar to those of read-only transactions in the baselines. Latencies for safe read-only transactions are slightly higher than those of fast transaction, because PG-ED's read-dependency tracking will ensure that their commits will not be acknowledged until all of the data they have read are durable. The latency gap between safe and fast transactions is very small in this experiment because there is little data contention. Thus, it is not likely that a safe read-only transaction will read very recently updated data. We would expect the latency gap between PG-ED's safe and fast read-only transactions to increase with increasing contention.

Reads in baseline PostgreSQL are both safe and fast because PostgreSQL transactions may read from an earlier and slightly staler snapshot of the database than a similar transaction in PG-ED. Figure 7 explains why this is. It shows the timelines of three read-write transactions ($T0$, $T1$, and $T2$) and a read-only transaction (Tr) in both PostgreSQL and PG-ED. Recall that transactions read from a snapshot of the database that includes the updates of all transactions that commit before the reader starts. Thus, in Figure 7a, in PostgreSQL Tr reads from a snapshot that includes updates made by $T0$, but that does not include $T1$ or $T2$. Since transactions in PostgreSQL are durable when they commit, Tr's read snapshot includes only durable updates. Furthermore, Tr's latency is not affected by the latencies of $T1$ and $T2$, since their updates are not included in Tr's read snapshot.

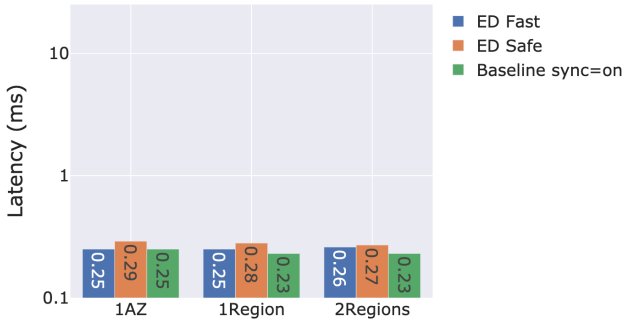
Figure 7b illustrates the same set of transactions in PG-ED, where each transaction commits first and becomes durable later. In this case, Tr's read snapshot includes the updates of *both* $T0$ and $T1$, since both commit before Tr starts. Thus, in PG-ED, Tr actually sees a *later* read snapshot than in it would have seen in the baseline system. However, that snapshot may include both durable and non-durable updates, such as those of transaction $T1$, which is committed but not yet durable at the time Tr starts. If Tr is a fast transaction, PG-ED can acknowledge its commit immediately, even though Tr's read set includes $T1$'s non-durable updates. Thus, Tr's latency in PG-ED will be similar to its latency in the baseline system, though it sees more recent updates in PG-ED. However, if Tr is a safe transaction, then PG-ED cannot acknowledge Tr's commit until any non-durable updates it has read from its snapshot have become durable. In the example in Figure 7b, assuming Tr reads from $T1$, Tr cannot be acknowledged until $T1$ becomes durable. This is why a read-only transaction may have higher latency if it is safe than if it is fast, and higher latency than a similar transaction would have in the baseline system.

7.4 Contention

Although the primary motivation for eventual durability is to provide latency-sensitive applications with a tool for reducing transaction latencies, eventual durability can also improve database system throughput by reducing data contention. Eventual durability can reduce contention because transactions commit quickly, without waiting for durability. Thus, commit-released resources, such as locks, will be held for less time in an ED database system.

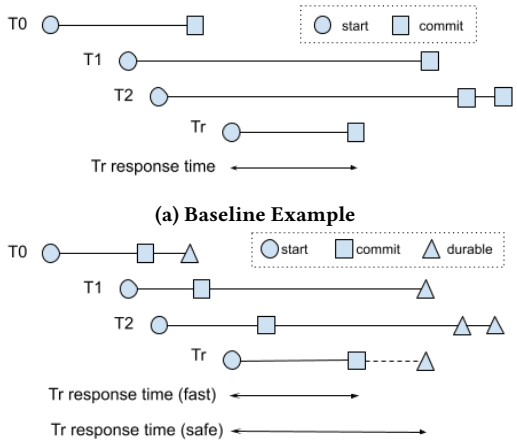


(a) Update Transaction Latency



(b) Read-Only ED Transaction Latency

Figure 6: Transaction Latencies



(a) Baseline Example

(b) Eventual Durability Example

Figure 7: Transaction Read Examples

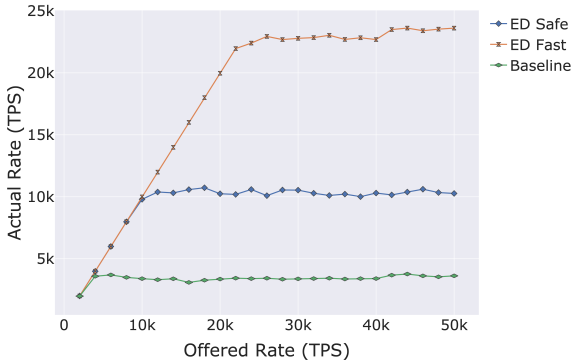


Figure 8: Transaction Throughput Under High Contention

To demonstrate this, we ran an experiment in which both PG-ED and baseline PostgreSQL are offered a simple high-contention workload. In each case, the database system configuration was the same as that used in the *1Region* latency experiments. The test database consists of a single small table with two integer columns (key and value) and 256 rows. We configured pgbench with 128 clients, each of which submits simple transactions that each update the value of a single row, selected uniformly at random. We controlled the offered load, i.e., the aggregate rate with which the clients try to submit transactions to the server.

We ran a series of experiments in which the offered load ranged from 2000 transactions per second (TPS) to 50000 TPS. In PostgreSQL (both the baseline and our ED prototype), write/write data contention can manifest as latency or as transaction aborts. In these experiments, we configured the clients to automatically retry aborted transactions up to five times, so that almost all transactions eventually commit. In each experiment, we measured the actual transaction rate sustained by the server, i.e., the actual rate at which transactions successfully committed.

Figure 8 shows the result of this experiment. The "baseline" result shows our measurements for the baseline PostgreSQL system. The "ED Fast" and "ED safe" results show our measurements for PG-ED when the clients submitted fast or safe transactions.

In these experiments, the baseline system's throughput reached a maximum of a little over 4000 TPS. With safe transactions, the ED prototype reached a maximum throughput of roughly 12000 TPS. In both cases, the clients are not exposed to any failure risks; transactions are not acknowledged until they are durable. However, since the ED prototype is able to release locks at commit time (well before the transaction is acknowledged), there is less data contention and less chance that a transaction will have to be aborted and retried due to a write-write conflict. At peak load, roughly 30% of the work done by the baseline PostgreSQL system was due to transaction retries. For PG-ED, retries made up only about 5% of total work.

In our experiment, PG-ED was able to run fast transactions at about 23000 TPS. Ideally, we would expect to see similar performance from PG-ED in this experiment for both fast and safe transactions, since both types of transactions are committed quickly (before durability). The gap between fast and safe transaction performance is due to a limitation of PG-ED. It releases some, but not all, transaction locks when a transaction commits. Specifically, it effectively releases row locks at commit time, but "object" locks (those that are visible in `pg_locks`) are not released until the transaction is durable. This overly conservative lock release strategy simplified PG-ED, but it leaves some performance on the table, as shown by the experiment. An implementation with more aggressive lock release should be able to achieve safe transaction throughput comparable to that of fast transactions, which release all locks at commit time.

Finally, we note that by releasing ED transaction locks at commit time, the ED prototype is essentially realizing the same benefits that are obtained by early lock release [6, 10, 15] for normal (non-ED) transactions. We discuss this further in Section 8.

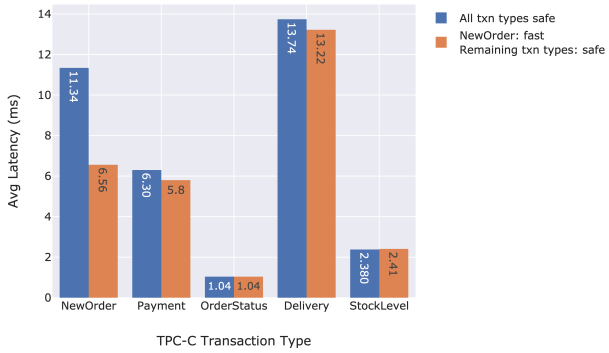


Figure 9: TPC-C Transaction Latencies

7.5 TPC-C

Our intention in running a TPC-C workload is to explore how an application might trade durability for performance. As a baseline scenario, we first run TPC-C against PG-ED with all transactions safe. Then, we consider an alternative scenario in which we run NewOrder transactions as fast, and all other transactions as safe. Under this alternative, the application should see (significantly) lower latency for NewOrder transactions than it did in the baseline. However, there is a risk that acknowledged NewOrder transactions might be lost. By running the remaining transactions as safe, the application can manage and limit this risk. In particular, since OrderStatus is safe, any order reported by an OrderStatus transaction is guaranteed to be durable. Similarly, any order that is paid for (Payment) or delivered (Delivery) is guaranteed to be durable.

We ran both TPC-C scenarios against PG-ED. The setup of this experiment is similar to the one used for the contention tests, but we used a *1Region* configuration rather than a *1AZ* configuration. (The results are qualitatively similar in the *1AZ* configuration, but the latencies are lower in *1AZ*.) We use CMU's benchbase [7] – 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 10 and 100 terminals. The test ran for 10 minutes at a rate of 300 TPS, and we recorded the latencies of the different transaction types.

Figure 9 shows the latency measurements for both TPC-C scenarios. In our test configuration, the application can cut the latency of NewOrder transactions almost in half by running them as fast transactions. As expected, the latencies of the remaining transaction types are about the same under both scenarios.

8 RELATED WORK

Durability's impact on performance has been a known problem for decades. One strategy for reducing the performance impact of durability is *group commit* [5]. When an application requests to commit a transaction, the system first decides whether it is willing to allow the transaction to commit. If it is, then the transaction is said to be *pre-committed*, without durability. Pre-committed transactions are made durable in batches (groups). The system waits for durability before acknowledging the commit to the application, which is unaware of pre-commitment.

Group commit can improve performance in several ways. First, making transactions durable in batches can be more efficient than making them durable individually. For example, a single disk write can be used to carry the commit records of multiple transactions to the disk. Second, the system can release database locks and other resources at pre-commit time instead of holding them until durability, thus potentially reducing contention for those resources. Of course, this means that updates made by pre-committed transactions might be read by subsequent transactions. Some mechanism, such as explicit dependency tracking, is needed to ensure that such transactions do not commit before their dependencies do.

This approach was described by DeWitt et al [5], and was implemented by IBM's IMS/VS as part of its FastPath feature for in-memory data [9]. It was also implemented in Shore-MT [15] and by Graefe et al for foster B-Trees [10]. Graefe et al also proposed a similar technique, called controlled lock violation [11], which retains locks past the pre-commit point and uses them as the means of tracking dependencies.

Similar ideas have been applied in distributed settings. COCO [18] amortizes the cost of durably committing transactions by dividing transaction execution into epochs and committing all of the transactions in an epoch as a batch. More recently, *Concurrent Prefix Recovery (CPR)* [22], which is implemented in a transactional key value store called FASTER, pushed this idea further. With CPR, pre-commited transactions are periodically made durable in bulk using an efficient transaction-consistent snapshotting technique, and commits are also acknowledged to the database application in bulk. Distributed Prefix Recovery [17] extends the CPR approach to a distributed setting.

There are two main differences between pre-commit and eventual durability. First, pre-commit is purely a system performance optimization, and is transparent to database applications. Pre-committed transactions are never acknowledged before they are

durable. Thus, pre-commit does not demand any changes to the transaction model, which forms the contract between system and application. Second, although pre-commit/group commit techniques can improve transaction throughput by reducing resource contention, they do not directly reduce transaction latencies, since commits are not acknowledged before durability. Indeed, they may *increase* individual transaction latencies in order to accumulate batches of transactions so that they can be made durable in bulk.

Eventual durability can be seen as pushing the pre-commit approach further, in that committed but not-yet-durable transactions may be exposed to applications. This provides opportunities for speculation at the application level, provided that the application is able to manage and accept the risk of failed transactions. The eventual durability model also naturally exposes the same performance optimization opportunities that are exploited by pre-commit. In particular, resources held by eventually durable transactions can be released at the commit point, without waiting for durability. In this sense, the commit point of an ED transaction is analogous to the pre-commit point of a classical transaction.

Exposing Non-Durable Transactions. Many database systems offer the ability to sacrifice durability for performance. Examples include *asynchronous commit* options in PostgreSQL [27, Ch. 30.4] and Oracle TimesTen [20], which acknowledge a transaction's commit before the transaction is guaranteed to be durable. We refer to these as *ad hoc* techniques, since their behavior is system-dependent. They make it difficult for applications to manage failure risks, as it is difficult for the application to tell when transactions have become durable, or whether data read from the database are durable.

Chang et al [3] propose a notion of weak durability called ACID$^-$, under which transactions are not durable when they commit. The ACID$^-$ proposal prescribes a specific API, under which committed transactions become durable *if and only if* durability is requested by the application via a special `persist` operation. In contrast, eventual durability proposes a more general extension of the underlying transaction model. The ED model can used to define a variety of APIs, and it treats durability as a *system-managed* property: all transactions (hopefully) become durable sometime after commit. More importantly, our work shows how to adapt classical notions of recoverability and consistency (e.g., serializabilty) to the the ED setting (Section 5). Unlike ACID$^-$, which essentially describes a form of crash consistency, the ED model can capture a variety of failures that cause transactions to be lost, ranging from crash-and-restart to more localized partial failures in large-scale systems.

Persistent Memory. Durability is also an issue for the design of in-memory data structures for persistent memory, or for hybrid memories, which include both volatile and persistent state. Israelevitz et al [14] observe, as we have, that demanding that all operations be durable when they are acknowledged is likely to be expensive. They introduce *buffered durable linearizability* as a correctness condition for such systems. Like ED serializability, it requires that operations be ordered when they return, and it allows for the possibility that some transactions at the tail of the sequence might be lost as a result of a crash.

File Systems. File systems often persist updates lazily, deferring any durability guarantees until the file is closed or until explicit synchronization, e.g., POSIX `fsync` [26]. `xsyncfs` replaces explicit program-issued synchronization operations with an *external consistency* guarantee: file writes are guaranteed to be durable before any process outputs (e.g., terminal I/O) that might depend on those writes can be exposed. This provides some opportunity for lazy persistence without requiring explicit durability fences.

Replicated Distributed Systems. In replicated distributed systems, durability is achieved by storing multiple copies of updates. Updates are propagated to all copies, either synchronously or asynchronously. These two strategies were termed *eager* and *lazy* by Gray et al [12], who summarized the advantages of each approach. Replicated distributed database systems, like Spanner [4], CockroachDB [24], and TiDB [13] typically use eager replication to ensure consistency and durability, but some systems offer lazy replication as an option. For example, both PostgreSQL [27, Ch. 27] and Microsoft SQL Server [19] can be run in high-availability configurations in which transactions are replicated to multiple locations, and both systems offer an asynchronous replication option.

Li et al. [16] introduce the idea of RedBlue consistency, which allows users to issue fast (blue) operations that execute locally and are lazily replicated, alongside slow (red) operations which provide strong consistency and serializability guarantees. This work focuses on consistency, rather than durability, but we note it here because it exposes different consistency guarantees to the application by tagging transactions (red vs. blue) in much the same way that our example API distinguishes fast and safe transactions.

Saucr [1] reduces the cost of durability by dynamically adjusting whether updates are written to disk on each node, or only to memory. If enough nodes are up, Saucr writes only to memory, trusting that a failed system can recover lost state from another system's memory. Otherwise, Saucr uses a more expensive path in which updates are pushed to local persistent storage (disk). However, in either case, Saucr must communicate with a majority of the replicas before it can acknowledge any updates. Orca [8] reduces update latency by deferring durability guarantees for updates until those updates are later read, a technique which is referred to as *consistency-aware durabilty*. ED recoverability (§ 5.2) imposes a similar requirement at the transaction level: a transaction that reads an update cannot itself become durable until the updating transaction is durable.

9 CONCLUSION

We have proposed a model of eventually durable transactions. It offers a foundation for fine-grained application-managed durability/performance tradeoffs. We showed how traditional notions of correctness, such as serializability and recoverability, translate to the eventual durability setting. We also presented Pg-ED, a modified version of PostgreSQL which supports eventual durability. Our goal for this work is to enable *eventually durable data systems* supporting *durability-aware applications*, with the eventually durable transaction model as the basis of the contract between them.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada, and by a Google DAPA Research Award.

REFERENCES

[1] Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. Fault-tolerance, fast and slow: exploiting failure asynchrony in distributed systems. In *Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, USA, 391–408.

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

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

[4] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems* 31, 3 (Aug. 2013), 1–22. https://doi.org/10.1145/2491245

[5] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. 1984. Implementation techniques for main memory database systems. *ACM SIGMOD Record* 14, 2 (June 1984), 1–8. https://doi.org/10.1145/971697.602261

[6] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. 1984. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data*. 1–8.

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

[8] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. Strong and Efficient Consistency with Consistency-aware Durability. *ACM Transactions on Storage* 17, 1 (Feb. 2021), 1–27. https://doi.org/10.1145/3423138

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

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

[11] Goetz Graefe, Mark Lillibridge, Harumi Kuno, Joseph Tucek, and Alistair Veitch. 2013. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2463676.2465325

[12] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. 1996. The dangers of replication and a solution. *ACM SIGMOD Record* 25, 2 (June 1996), 173–182. https://doi.org/10.1145/235968.233330

[13] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[14] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*. Springer, 313–327.

[15] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. 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*. 24–35.

[16] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making {Geo-Replicated} systems fast as possible, consistent when necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 265–278.

[17] Tianyu Li, Badrish Chandramouli, Jose M Faleiro, Samuel Madden, and Donald Kossmann. 2021. Asynchronous Prefix Recoverability for Fast Distributed Stores. In *Proceedings of the 2021 International Conference on Management of Data*. 1090–1102.

[18] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-based commit and replication in distributed OLTP databases. *Proceedings of the VLDB Endowment* 14, 5 (Jan. 2021), 743–756. https://doi.org/10.14778/3446095.3446098

[19] Microsoft 2023. *Differences between availability modes for an Always On availability group*. Microsoft. Retrieved 2024-01-22 from https://learn.microsoft.com/en-us/sql/database-engine/availability-groups/windows/availability-modes-always-on-availability-groups?view=sql-server-ver16

[20] Oracle 2023. *Oracle TimesTen In-Memory Database* (release 22.1 ed.). Oracle.

[21] The PostgreSQL Global Development Group 2024. *PostgreSQL 16.1 Documentation, pgbench - A Benchmarking Tool for PostgreSQL*. The PostgreSQL Global Development Group. Retrieved 2024-01-22 from https://www.postgresql.org/docs/current/pgbench.html

[22] Guna Prasaad, Badrish Chandramouli, and Donald Kossmann. 2019. Concurrent Prefix Recovery: Performing CPR on a Database. In *Proceedings of the 2019 International Conference on Management of Data*. 687–704.

[23] Hironobu Suzuki. 2023. *The Internals of Postgres*. Retrieved 2024-01-22 from http://www.interdb.jp/pg/index.html

[24] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[25] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. International Foundation for Autonomous Agents and Multiagent Systems, Portland OR USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[26] The Open Group 2018. *The Open Group Base Specifications Issue 7, 2018 Edition*. The Open Group. Retrieved 2024-10-01 from https://pubs.opengroup.org/onlinepubs/9699919799/

[27] The PostgreSQL Global Development Group 2023. *PostgreSQL 16.1 Documentation*. The PostgreSQL Global Development Group. Retrieved 2024-02-05 from https://www.postgresql.org/docs/16/index.html

[28] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[29] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, Chicago Illinois USA, 1041–1052. https://doi.org/10.1145/3035918.3056101