LEVERAGING DISTRIBUTED STORAGE REDUNDANCY IN DATACENTERS

Amy Tai

A DISSERTATION

PRESENTED TO THE FACULTY

OF PRINCETON UNIVERSITY

IN CANDIDACY FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE BY

THE DEPARTMENT OF COMPUTER SCIENCE

Advisor: Michael J. Freedman

January 2019

Abstract

All distributed storage systems replicate data objects, providing built-in redundancy that is designed to help the system withstand failures. Such redundancy is *unavoidable* because withstanding failures is a critical goal of distributed systems, and redundancy is the only way to tolerate failures that cause loss of data access.

However, with the proliferation of data, it is becoming ever more paramount to reduce the costs of distributed storage systems. To balance the need to reduce storage costs and the need to withstand failures, this thesis explores two ways we can leverage the unavoidable redundancy in distributed storage systems to eliminate additional storage overheads in other parts of the storage stack.

The first system we present is Replex. The key end-to-end observation in this work is that distributed secondary indices duplicate the work done by replication. Secondary indices often store full copies of data objects, *in addition to* the replicas of data objects that are created by default to handle failures. In Replex, we eliminate the additional storage overhead of secondary indices by treating them as data replicas during replication time.

The second system we present is DIRECT. The key end-to-end observation here is that the redundancy created by replication can and *should* be used to correct bit errors at the hardware level. Traditionally, disks are expected to abstract bit errors from software, and in fact flash devices are shipped with aggressive internal error correction mechanisms to prevent errors from percolating to the user for the calculated lifetime of the device. In DIRECT, we argue that the underlying premise that disks should not expose bit errors is incorrect. In doing so, DIRECT enables the use of flash devices well beyond their advertised lifetime, which is a huge cost savings for datacenter operators.

Therefore, by applying existing storage redundancy to enable two key properties in datacenter storage systems– secondary indexing and flash reliability– this thesis shows that distributed storage systems can be designed without burdensome storage overheads.

Acknowledgments

To my advisor, Mike Freedman, without whom this thesis would not be possible. Thank you for understanding that a PhD is both about the intellectual pursuits and the personal journey. Thank you for giving me the freedom to explore research interests and for cultivating my skills as a researcher. Most importantly, thank you for seeing my potential when even I did not.

Also Dahlia Malkhi, one of the most invaluable mentors I have met in my life. Thank you for giving me the chance to work on interesting problems with a truly impressive team. Also Ittai Abraham and Asaf Cidon, fantastic mentor-collaborators I've had along the way.

Also to my very excellent collaborators: Michael Wei, who deserves a distinct call-out for his friendship, mentorship, invaluable and endlessly entertaining whiteboard sessions, and the many car-ride conversations (sorry about your car oops). And Andrew Kryczka, with wonderfully insightful whiteboard sessions, friendship, and advice.

Thank you to my thesis readers, Wyatt Lloyd and Kyle Jamieson, for their advice and encouragement.

To the women of Princeton Lady Clock. For the first time, I was part of a team, became part of something bigger. In the years I spent with you, I found mental grit, self confidence, and friendship. May we always live life in pursuit of the Spirit of the Game.

To the friends I've made along the way due to a shared misery: Naga, Marcela, Gfed, the members of SNS, and the many office mates I've had. May we continue to feel bonded even as we are scattered to the four winds.

Finally to my parents, my sister, and my dog, for providing not only the emotional support to make it through this PhD but also for nurturing the life that has enabled me to undertake and complete this journey.

Contents

Chapter 1

Introduction

1.1 Distributed Storage Systems

With data being created everywhere at every second, distributed storage systems are of growing importance. For example, as of 2014, Facebook stored 300 PB of data in just one of their storage systems, with data added at a daily rate of 600 TB [16]. This explosion in data volume makes it ever more necessary to decrease the cost of storing data. But to understand how to optimize storage overheads in distributed storage systems, it is first important to understand how they have evolved.

Distributed storage systems were created when it became clear that databases needed to store data beyond the capacity of a single disk on a single machine. For this thesis, we focus in particular on distributed storage systems *in the datacenter*. By this we mean that the following assumptions apply:

1. The storage systems are deployed at scale, which means failure events with small probabilities become much more likely. This assumption forces us to focus efforts on optimizing recovery performance throughout this thesis.
2. All machines on which the system runs are contained within a single datacenter. In particular, this rules out geo-distribution, which is a separate consideration in and of itself. This assumption also implies that communication between machines is on the

Figure 1.1: We focus on distributed storage systems in the datacenter. This means that the system stores data (colored blocks) across machines in a datacenter (the outlined box).

Property	**Definition**
Durability	Data stored in the system should not be lost despite hardware (such as disk) failures.
Availability	Data stored in the system should be accessible despite hardware (such as network) failures.

Table 1.1: Some properties that distributed storage systems guarantee.

order of a few milliseconds or less instead of several hundreds of milliseconds. This assumption is particularly helpful when reducing recovery costs during failure.

Figure 1.1 depicts how these systems generally store data in a datacenter.

Distributed storage systems make an implicit contract with users wherein they guarantee certain properties, typically durability and availability, as summarized in Table 1.1.

Consider the scenario in Figure 1.2. If the specified disk fails, then naively, the green data block would be lost forever. If a storage system guarantees durability, losing the green data block is unacceptable. Therefore the system has to do something smarter in order to prevent losing data when such failures occur.

1.2 Replication

The easiest way to guarantee both durability and availability is with a technique called data replication. Replication relies on a simple but powerful concept: by creating redundant

Figure 1.2: Durability is a property that guarantees that hardware failures do not result in data loss. For example, if the indicated machine crashes, the distributed system should still be able to recover the green data block.

Figure 1.3: Replication ensures that data blocks are still preserved despite some number of failures.

copies (replicas) of a data block throughout the datacenter, systems can withstand some number of machine failures because copies of the data reside on healthy machines. Figure 1.3 clearly shows how, because every data block is replicated three times within the datacenter, a machine failure will not result in data loss.

It is clear how replication achieves durability and availability, but in doing so, it also imposes several new overheads, as summarized in Table 1.2. Unfortunately, because redundant information is the only way to withstand failures, the overheads inherent to replication are unavoidable; every distributed storage system that intends to be used seriously in production has some level of data replication. Because the storage and performance costs

Overhead	Description
Storage	If a data object is replicated r times, then the storage overhead of replication is $r\times$. This can be a huge overhead in datacenter deployments, where the data itself can be on the order of petabytes without replication.
Performance	With replication, whenever an update is made to a data object, all replicas of the data object must also be updated.

Table 1.2: While replication enables storage systems to withstand failures, it does so by introducing a number of overheads.

of naive replication can be exorbitant, there is ample related work, which we will cover in Chapter 2, that attempts to alleviate these overheads by introducing tradeoffs. However, the limitation of these existing attempts is that they respect the abstraction of the replication layer; existing research efforts only work to explore tradeoffs within the replication operation itself – for example, how different replication strategies (consistency models, Section 2.3) reduce the performance overheads of replication by shifting the burden to programmers. Instead of optimizing within the replication layer, this thesis looks to other layers of the storage stack and asks how we can leverage the redundancy from replication to provide beneficial, end-to-end properties across the entire stack.

1.3 Contributions

In this thesis we identify novel approaches to addressing replication overheads by opening abstraction layers and co-designing replication with other layers of the storage stack. Layers of abstraction result in unnecessary storage redundancy across the entire stack. Armed with the end-to-end argument, we realize that these abstractions, having served their purpose by making rapid iterations of software engineering possible, should be removed in order to redesign a stack with less redundancy; by opening the storage stack, many cases of storage redundancy become unnecessary. In the two works of this thesis, we identify two new applications of the redundancy in replication that can be used to reduce storage overheads in other parts of the stack.

First, we present Replex, which shows that replication can be co-designed with the indexing layer in storage systems. Replex treats distributed secondary indices as replicas during replication time. In doing so, Replex explores the tradeoff of storage redundancy vs. recovery overhead at the indexing layer, a flexibility that did not previously exist. Traditionally, systems fall in favor of high storage redundancy and low recovery overhead, but the datacenter presents new assumptions: storage is costly, and applications tolerate a range of acceptable recovery times, which means datacenter storage systems benefit from exploring the full tradeoff space.

Next, we present DIRECT, which co-designs replication with the error handling layer in hardware. DIRECT shows that replication at the distributed storage layer can correct bit errors at the hardware level. In this case, storage redundancy exists at the device *and* application level because devices are traditionally assumed to mask all data corruptions from software. DIRECT challenges the notion that disks must abstract all errors from software layers by showing that bit errors can be fixed at the application layer with application-level redundancy with little performance overhead. By co-designing hardware redundancy with replication, we enable flash devices to be used for much longer than their currently advertised lifetimes, which reduces the storage overhead of a system.

There is no previous work that attempts to leverage datacenter redundancy in these ways, and we believe that the work in this thesis is a first step towards eliminating unnecessary storage overheads throughout the distributed storage stack.

Chapter 2

Background

As discussed in the previous chapter, replication has several overheads. Much of the existing work in this area has focussed on mitigating these overheads by introducing tradeoffs. For example, to mitigate the storage overhead replication imposes, the entire field of erasure coding introduces a storage versus recovery time computation tradeoff. Similarly, to mitigate the update overhead introduced by having to update multiple copies of the same data object, a slew of NoSQL databases explores the tradeoff between scalable performance and consistency model.

In this chapter, we introduce these tradeoffs by discussing these two bodies of related work: erasure coding and NoSQL databases.

2.1 Erasure Coding

Traditional replication stores a data object r times, so if the data object is of size k, then the storage overhead associated with the object becomes $k \cdot r$. Instead, erasure codes provide a way to protect a data object against failures with only a fraction of the storage overhead.

Erasure codes offset the storage overheads of distributed replication by introducing a storage versus recovery tradeoff. Furthermore, by reducing storage overheads, erasure codes can also reduce power and other operational overheads associated with datacen-

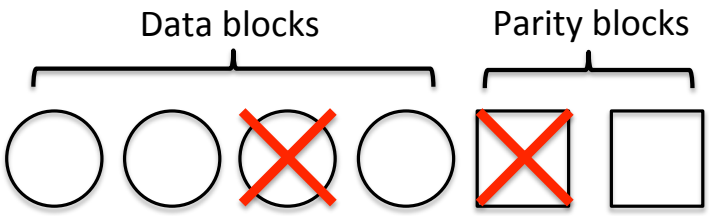


Figure 2.1: Generally, erasure coding computes $n - k$ parity blocks from a set of k data blocks. Then if there are fewer than $n - k$ failures in the set of n blocks, any failed block can be recomputed from the remaining blocks. In this case, $k = 4$ and $n = 6$. Then this setup can tolerate 2 failures. In particular, the lost data block can be recomputed from the remaining four available blocks.

ters [58]. In particular, with erasure codes, a data object of size k can be represented with $k < n$ bits, where $k < n < 2k$. Therefore, the storage overhead imposed by the erasure code is $n - k$, some fraction of k.

2.1.1 Basics of erasure coding

Generally, a data object of size k is extended into a data object of size n bits; the additional $n - k$ are termed *parity* bits, which are computed from the k data bits. Then the idea is that any k of the n bits are enough to reconstruct the original data object.

Extending this idea to distributed storage systems, we replace bits with data objects, and a data object with some aggregate set of data objects. Then if there are k data objects in the set, we can calculate $n - k$ parity objects, and the resulting set of n data objects are successfully erasure coded. Then the guarantee we get is that if any one of the original k data objects is unavailable, we can read from any k other available data objects and recompute the missing object, as shown in Figure 2.1.

In general, a (n, k) erasure code can tolerate $n - k$ "erasures", in this case missing data fragments. Recall that replicating r-ways is to tolerate $r - 1$ "erasures" of a data object. Then, to ensure that all k data objects in our aggregate set can tolerate $r - 1$ erasures, we simply set $n - k = r - 1$, in which case $n = k + r - 1$. Hence the storage overhead of an erasure code that can tolerate $r - 1$ failures is $n/k = (k + r - 1)/k << r \cdot k$, where the right-hand value is the storage overhead of replication.

The tradeoff for this low storage overhead is an increase in update and reconstruction cost. For the regular replication case, the update overhead is having to update all r copies of the data block. For erasure coding, an update must update all data fragments that are associated with the data block. As we calculated above, there are $k + r - 1$ such objects. Generally, $k > 1$ (if $k = 1$, then we have simply degenerated to the replication case), which means that the update overhead of erasure coding is strictly greater than the update overhead of replication.

Furthermore, if a fragment in an erasure coding set experiences a failure, then k data fragments must be accessed *and* the lost block must be recomputed from these k fragments. Contrast this with replication, where there is no computational overhead: simply fetching a remote replica is sufficient. This cost becomes particularly magnified when there is a network cost involved in fetching the missing fragments.

There are many families of erasure codes, including Reed-Solomon codes [96] and locally-recoverable codes [106], which improve on the recoverability of data at the expense of additional storage overhead.

2.2 RAID

Redundant Array of Inexpensive Disks (RAID) is another construct that trades off availability for storage and performance overheads [94]. RAID refers to a collection of disks that are attached to a single server and gives the abstraction of a single disk.

In particular, RAID usually assumes inexpensive and hence failure-prone disks, which means RAID must be able to preserve data in the face of disk failures, as shown in Figure 2.2. As with erasure coding, RAID does so by introducing storage redundancy and additional write overheads.

There are a variety of RAID techniques, called *levels*, each a different point in the tradeoff space between failure tolerance and storage and computational overhead. For example, RAID 0 is the degenerate case where there is no redundancy, which means that the RAID

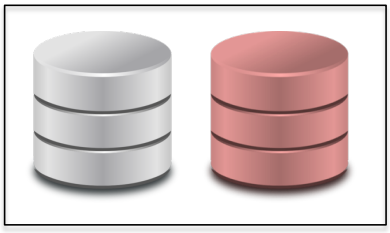
Figure 2.2: Redundant array of Inexpensive Disks (RAID) is a technique for combining multiple disks in a single disk abstraction. Depending on the RAID level, the setup must be able to tolerate individual disk failures.

0 level cannot tolerate any disk failures.

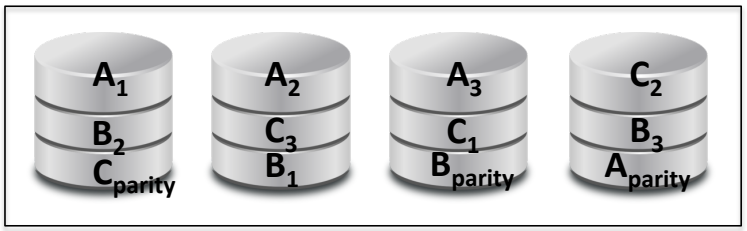
RAID Level	Description
RAID 0	No protection against failures
RAID 1	Every data block is replicated once, for a total of two full copies of a data block in the RAID setup. This level can tolerate one disk failure, but the storage overhead is 2x the size of the data. See Figure 2.3a.
RAID 5	This setup can tolerate one disk failure, but with parity blocks instead of the full replication of RAID 1. This means that if there are n disks in the setup, every $n-1$ blocks has a parity block associated with it, that resides on a separate physical disk. Every write, however, must then recalculate the parity block, which results in a higher computational overhead than RAID 1. See Figure 2.3b.
RAID 6	This setup can tolerate two disk failures, which means if there are n disks in the setup, every $n-2$ blocks has two parity blocks associated with it. This level can tolerate more failures than the others, at the expense of a larger storage overhead (two instead of one parity block), as well as higher computational costs for each write.

Table 2.1: Summary of some standard RAID levels and their tradeoffs.

On the other hand, RAID 6 can tolerate two disk failures, because each data block has two parity (and hence redundant) blocks associated with it. The overhead introduced by these parity blocks is both a write overhead and a storage overhead. Table 2.1 summarizes the standard RAID levels.



(a) RAID 1, where the red disk is a full replica of the original, gray disk.



(b) RAID 5, where $n = 4$, which means each specified set of $n - 1 = 3$ blocks has an associated parity block. A, B, and C indicate the disjoint sets of data blocks.

Figure 2.3: Visualizations of two different RAID levels.

2.3 NoSQL Datastores

In addition to tackling the storage overhead tradeoff by implementing erasure coding, NoSQL datastores can also tackle the *update overhead* of supporting multiple replicas of a single data object. Simply, the update overhead of replication is having to update all replicas of a data object when that object is updated. The requirement of updating all replicas *atomically* exists so that distributed storage systems can give the illusion of running on a single machine.

If we relax the condition– for example, if an update operation does not need to propagate to all replicas– then some reads to a data object will see different values. The *consistency model* chosen by a datastore specifies how far the datastore relaxes the condition that all updates much propagate to all replicas. As expected, generally the stronger the consistency model, the less performant the datastore is. Conversely, weaker consistency models enable a datastore to advertise both lower latency and higher throughput operations. In this section, we summarize a few salient consistency models that are commonly used in NoSQL datastores.

2.3.1 Strong Consistency

Strong consistency is a model where all operations will see the result of all other finished operations in the system. Generally, this means that the state of the system reflects some global order of operations, which makes it very easy for users to reason about data. However, the tradeoff is poor performance, because operations cannot return until their slot in a global ordering of operations can been determined. Usually, the easiest way of doing this is to ensure the operation is executed on all nodes. Chain replication [116], CRAQ [109], an optimization of chain replication, HyperDex [49], and Windows Azure Storage [39] are all strongly consistent systems that achieve strong consistency by enforcing that writes visit all nodes before returning to the user.

Another way to achieve strong consistency is to use a consensus engine to determine the global ordering operations. Even though a write only has to visit some large subset of all nodes (typically a quorum) instead of all the nodes, the consensus engine pays for this tradeoff by introducing an additional layer of complexity because its nodes must come to an agreement on how to order the operations. Systems like Spanner [43], CockroachDB [1], and some configurations of Cassandra [111] and ScyllaDB [100] use consensus (quorum)-based replication.

2.3.2 Causal Consistency

Causal consistency [23], as the name suggests, is a consistency model that guarantees that operations that depend on each other will have the effect of being executed in a serial order. In particular, writes that affect the same data object will be serialized. Therefore, the difference between causal and strong consistency is that causal consistency does not impose an ordering on operations that are independent of each other. This flexibility gives causal consistency both a performance edge over strong consistency and is in fact the strongest consistency possible in the presence of network partitions [83].

Eiger [79] is an example of a geo-distributed system that provides causal consistency,

while COPS is an example of a system that provides causal+ consistency, which is even stronger than causal consistency. Also, Redis [12] can be configured to provide causal consistency.

2.3.3 Eventual Consistency

Eventual consistency is the weakest of these three consistency models. Originally introduced in the Bayou system [110], eventual consistency guarantees that eventually, every write or update operation made against a node in a distributed system propagates to and is hence replicated on all other nodes. Because there are no guarantees on the timing or ordering of when these operations make it to all nodes, systems that wish to provide some sense of coherency across nodes must use a higher protocol layer to resolve out-of-order operations. In particular, the original Bayou system implemented a protocol layer above the replication layer that ended up enforcing a model similar to causal+ consistency.

Obviously, the benefits of eventual consistency are great: writes can return as soon as they contact a single node in the system, rather than returning after visiting all replicas. This means that such systems can advertise very high write throughputs. The 2000s saw a rise in systems advertising eventual consistency precisely as a reaction to the relatively poor performance of strongly consistent databases. This rebirth in eventual consistency led to systems such as Dynamo [45], MongoDB [41], CouchDB [28], Redis [12], and Cassandra/ScyllaDB. All of these systems trade off consistency for high performance at the replication layer.

2.4 Summary

To summarize, the research fronts to reduce the storage and performance overheads introduced by replication are limited to the replication layer itself. Erasure coding explores ways to replicate data to achieve optimal points in the storage overhead vs recovery performance tradeoff. NoSQL datastores employ different replication strategies to explore the performance overhead vs. ease of programming tradeoff.

In this thesis we address the same tradeoffs, but in contexts in which we look beyond the replication layer to identify new ways to alleviate the burdens of replication. In the first work, Replex, we show that replication can be co-designed with the indexing layer in storage systems to reduce storage overheads. In the second work, DIRECT, we show that replication can be co-designed with availability mechanisms at the hardware level, to again reduce storage overheads. In both Replex and DIRECT, we explore the same storage overhead vs. recovery performance tradeoff, but in new contexts.

Chapter 3

Replex

3.1 Introduction

Applications have traditionally stored data in SQL databases, which provide programmers with an efficient and convenient query language to retrieve data. However, as storage needs of applications grew, programmers began shifting towards NoSQL databases, which achieve scalability by supporting a much simpler query model, typically by a single primary key. This simplification made scaling NoSQL datastores easy: by using the key to divide the data into partitions or "shards", the datastore could be efficiently mapped onto multiple nodes. Unfortunately, this model is inconvenient for programmers, who often still need to query data by a value other than the primary key.

Several NoSQL datastores [28, 40, 41, 63, 65, 111] have emerged that can support queries on multiple keys through the use of secondary indices. Many of these datastores simply query all partitions to search for an entry which matches a secondary key. In this approach, performance quickly degrades as the number of partitions increases, defeating the reason for partitioning for scalability. HyperDex [49], a NoSQL datastore which takes another approach, generates and partitions an additional copy of the datastore for each key. This design allows for quick, efficient queries on secondary keys, but at the expense of storage and performance overhead: supporting just one secondary key doubles storage

requirements and write latencies.

In this paper, we describe Replex, a scalable, highly available multi-key datastore. In Replex, each full copy of the data may be partitioned by a different key, thereby retaining the ability to support queries against multiple keys without incurring a performance penalty or storage overhead beyond what is required to protect the database against failure. In fact, since Replex does not make unnecessary copies of data, it outperforms other NoSQL systems during both steady-state and recovery.

To address the challenge of determining when and where to replicate data, we explore, develop, and parameterize a new replication scheme, which makes use of a novel replication unit we call a *replex*. The key insight of a replex is to combine the need to replicate for fault-tolerance and the need to replicate for index availability. By merging these concerns, our protocol avoids using extraneous copies as the means to enable queries by additional keys. However, this introduces a tradeoff between recovery time and storage cost, which we fully explore (§ 3.3). Replex actually recovers from failure faster than other NoSQL systems because of storage savings during replication.

We implement (§ 3.4) and evaluate (§ 3.5) the performance of Replex using several different parameters and consider both steady-state performance and performance under multiple failure scenarios. We compare Replex to Hyperdex and Cassandra and show that Replex's steady-state performance is 76% better than Hyperdex and on-par with Cassandra for writes. For reads, Replex outperforms Cassandra by as much as 2-9$\times$ while maintaining performance equivalent with HyperDex. In addition, we show that Replex can recover from one or two failures 2-3$\times$ faster than Hyperdex, all while using a fraction of the resources.

Our results contradict the popular belief that supporting multiple keys in a NoSQL datastore is expensive. With replexes, NoSQL datastores can easily support multiple keys with little overhead.

3.2 System Design

We present Replex's data model and replication design, which enables fast index reads and updates while being parsimonious with storage usage.

3.2.1 Data Model and API

Replex stores data in the form of RDBMS-style tables: every table has a schema that specifies a fixed set of columns, and data is inserted and replicated at the row-granularity. Every table also specifies a single column to be the primary key, which becomes the default index for the table.

As with traditional RDBMSs, the user can also specify any number of additional indices. An index is defined by the set of columns that comprise the index's **sorting key**. For example, the sorting key for the primary index is the column of the primary key.

The client queries we focus on in this paper are $insert(r)$, where r is a row of values, and $lookup(R)$, where R is a row of predicates. Predicates can be null, which matches on anything. Then $lookup(R)$ returns all rows r that match on all predicates in R. The non-null predicates should correspond to the sorting key of an index in the table. Then that index is used to find all matching rows.

Henceforth, we will refer to the data stored in Replex as the *table*. Then Replex is concerned with maintaining the indices of and replicating the table.

3.2.2 Data Partitioning with Replexes

In order to enable fast queries by a particular index, a table must be partitioned by that index. To solve this problem, Replex builds what we call a *replex* for every index. A replex stores a table and shards the rows across multiple partitions. All replexes store the same *data* (every row in the table), the only difference across replexes is the way data is partitioned and sorted, which is by the sorting key of the index associated with the replex.

Each replex is associated with a sharding function, h, such that $h(r)$ defines the partition

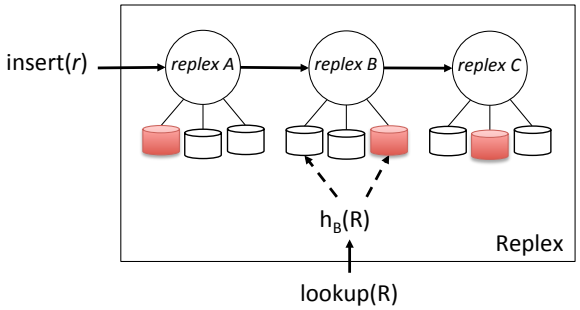


Figure 3.1: Every replex stores the table across of a number of partitions. This diagram shows the system model for a table with 3 indices. When a row r is inserted, h_A, h_B, and h_C determine which partition (shaded) in the replex stores r. Similarly, a lookup on a replex is broadcast to a number of partitions based on h.

number in the replex that stores row r. For predicate R, $h(R)$ returns a set because the rows of values that satisfy R may lie in multiple partitions. The only columns that affect h are the columns in the sorting key of the index associated with the replex.

A novel contribution of Replex is to treat each partition of a replex as first-class replicas in the system. Systems typically replicate a row for durability and availability by writing it to a number of replicas. Similarly, Replex uses chain replication [116] to replicate a row to a number of replex partitions, each of which sorts the row by the replex's corresponding index, as shown in Figure 3.1; in § 3.2.3 we explain why we choose chain replication. The key observation is that after replication, Replex has both replicated *and* indexed a row. There is no need for explicit indexing.

By treating replexes as true replicas, we eliminate the overheads associated with maintaining and replicating individual index structures, which translates to reductions in network traffic, operation latency, and storage inflation.

3.2.3 Replication Protocol

Replacing replicas with replexes requires a modified replication protocol. The difficulty arises because individual replexes can have requirements, such as uniqueness constraints, that cause the same operation to be both valid and invalid depending on the replex. Hence before an operation can be replicated, a consensus decision must be made among the re-

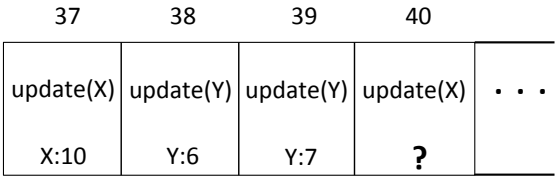


Figure 3.2: Consider storing every log entry in a Replex table. For linearizability, a local timestamp cannot appear to go backwards with respect to the global timestamp. For example, tagging in last entry with local timestamp X:9 violates the semantics of the global timestamp.

plexes to agree on the validity of an operation.

As an example of an ordering constraint, consider a distributed log that totally orders updates to a number of shared data structures, *a la* state machine replication. In addition to the global ordering, each data structure requires a local ordering that must reflect the global total ordering. For example, suppose there are two data structures X and Y, and a subset of the log is shown in Figure 3.2. To store the updates in Replex, we can create a table with two columns: a global timestamp and a local timestamp. Because consumers of the log will want to look up entries both against the global timestamp and within the sublog of a specific data structure, we also specify an index per column; examples of logs with such requirements appear in systems such as Corfu [30], Hyder [33], and CalvinFS [112].

Then the validity requirement in this case is a dense prefix of timestamps: a timestamp t cannot be written until all timestamps $t' < t$ have been inserted into the table; this is true for both the local and global timestamps. For example, an attempt to insert the row (40, X:9) would be valid by the index of the global timestamp, but invalid by the index of the local timestamp, because the existence of X:10 in the index means X:9 must have already been inserted. Then the row should not be inserted into the table; this is problematic if the first replex has already processed the insert, which means lookups on the first index will see row (40, X:9).

This example is implemented in the system vCorfu [118]. In vCorfu, Replex is used to enable efficient materialization of data structures. As long as the local timestamps (all timestamps associated with data structure X in Figure 3.2) maintain density, then to mate-

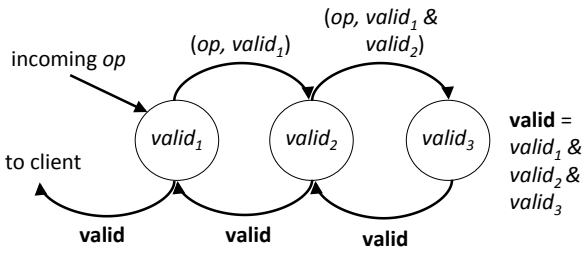


Figure 3.3: Each node represents an index. This modified replication protocol has two phases: 1) Top phase: propagates the operation to all relevant partitions and collects each partition's decision. 2) Bottom phase: the last partition aggregates these decisions into the final **valid** boolean, which is then propagated back up the chain. When a replex receives **valid**, it knows to commit or abort the operation

rialize a data structure, a reader simply sequentially scans local timestamps based on the secondary index provided by Replex.

Datastores without global secondary indices do not have this validity problem, because a key is only sorted by a single index. Datastores with global secondary indices either employ a distributed transaction for update operations, because an operation must be atomically replicated as valid or invalid across all the indices [43], or do not support constraints on secondary indices. Because replexes are similar to global secondary indices, a distributed transaction can do the job. But instead of running a distributed transaction *in addition* to the standard replication protocol, Replex rolls the indexing and replication into a single protocol.

To remove the need for a distributed transaction in our replication protocol, we modify chain replication to include a consensus protocol. We choose chain replication instead of quorum-based replication because all replexes must participate to determine validity. As in chain replication, our protocol visits every replex in some fixed order. Figure 3.3 illustrates the steps in this new replication protocol.

Our new protocol can be split into two phases: (1) **consensus phase**, where we propagate the operation to all replexes, as in chain replication. The actual partition within the replex that handles the operation is the partition that will eventually replicate the operation,

as depicted in Figure 3.1. As the protocol leaves each partition, it collects that partition's validity decision. When this phase reaches the last partition in the chain, the last partition aggregates each partition's decision into a final decision, which is simply the logical AND of all decisions: if there is a single abort decision, the operation is invalid. (2) **replication phase**, where the last partition initiates the propagation of this final decision back up the chain. As each partition receives this final decision, if the decision is to abort, then the partition discards that operation. If the decision is to commit, then that partition commits the operation to disk and continues propagating the decision.

It is guaranteed that when the client sees the result of the operation, all partitions will agree on the outcome of the operation, and if the operation is valid, all partitions will have made the decision durable. An intuitive proof of correctness for this consensus protocol is simple. We can treat the first phase of our protocol as an instance of chain replication, which is an instance of Vertical Paxos, which has existing correctness proofs [70]. The second phase of our protocol is simply a discovery phase in Paxos protocols and is hence irrelevant in the proof of correctness. This discovery phase is necessary for replexes to discover the final decision so they may persist (replicate) necessary data, but has no bearings on the consensus decision itself.

It is possible for a client to see committed operations at one replex before another. For example, suppose client 1 is propagating an operation to replexes A and B. The operation reaches B and commits successfully, writing the commit bit at B. Then this committed operation is visible to client 2 that queries replex B, even though client 2 cannot see it by querying replex A, if the commit bit is still in flight. Note that this does not violate the consensus guarantee, because any operation viewed by one client is necessarily committed.

Our protocol is similar to the CRAQ protocol which adds dirty-read bits to objects replicated with chain replication [109]. The difference between the two protocols is that CRAQ operates on objects, rather than operations: our protocol determines whether or not an operation may be committed to an object's replicated state machine history, while

CRAQ determines whether or not an object is dirty. In particular, operations can be aborted through our protocol.

Finally, we observe that our replication protocol does not allow writes during failure. In chain replication, writes to an object on a failed node cannot resume until its full persisted history has been restored; similarly, writes may not be committed in Replex until the failed node is fully recovered.

3.2.4 Failure Amplification

Indexing during replication enables Replex to achieve fast steady-state requests. But there is a cost, which becomes evident when we consider partition failures.

Failed partitions bring up two concerns: how to reconstruct the failed partition and how to respond to queries that would have been serviced by the failed partition. Both of these problems can be solved as long as the system knows how to find data stored on the failed partition. The problem is even though two replexes contain the same *data*, they have different sharding functions, so replicated data is scattered differently.

We define **failure amplification** as the overhead of finding data when the desired partition is unavailable. We characterize failure amplification along two axes: 1) disk IOPS and CPU: the overhead of searching through a partition that is sorted differently, 2) network traffic: the overhead of broadcasting the read to all partitions in another replex. For the remainder of the paper, we use failure amplification to compare recovery scenarios.

For example, suppose a user specifies two indices on a table, which would be implemented as two replexes in Replex. If a partition fails, a simple recovery protocol would redirect queries originally destined for the failed partition to the other replex. Then the failure amplification is maximal: the read must now be broadcast to every partition in the other replex, and at each partition, a read becomes a brute-force search that must iterate through the entire local storage of a partition.

On the other hand, to avoid failure amplification within a failure threshold f, one could

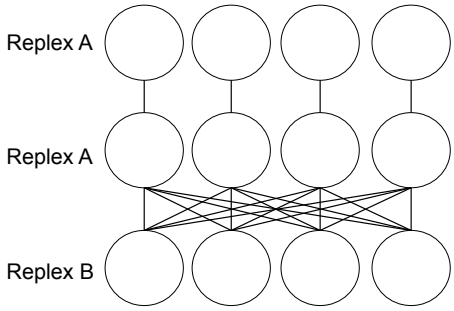


Figure 3.4: In graph depictions of replexes, nodes are partitions and edges indicate two partitions might share data. For example, because replexes A and B have independent sharding functions, it is possible for all combinations of nodes to share data. This graph shows a simple solution to reduce the failure amplification experienced by replex A, which is to replicate A again.

introduce f replexes with the same sharding function, h; these are the exact replicas of traditional replication. There is no failure amplification within the failure threshold, because sharding is identical across exact replicas; the cost is storage and network overhead in the steady-state.

The goal is to capture the possible deployments in between these two extremes. Unfortunately, without additional machinery, this space can only be explored in a discrete manner: by adding or removing exact replicas. In the next section, we introduce a construct that allows fine-grained reasoning within this tradeoff space.

3.3 Hybrid Replexes

Suppose a user schema specifies a single table with two indices, A and B, so Replex builds two replexes. As mentioned before, as soon as a partition in either replex fails, reads to that partition must now visit all partitions in the other replex, the disjoint union of which is the entire dataset.

One strategy is to add replexes that are exact replicas. For example, we can replicate replex A, as shown in Figure 3.4. Then after one failure, reads to replex A do not see any failure amplifcation. However, adding another copy of replex A does not improve failure amplification for reads to B: if a partition fails in replex B, failure amplification still becomes worst-case.

To eliminate failure amplification of a single failure on both replexes, the user must create exact replicas of both replexes, thereby doubling all storage and network overheads previously mentioned.

Instead, we present **hybrid replexes**, which is a core contribution of Replex. The basic idea behind hybrid replexes is to introduce a replex into the system that increases failure resilience of *any number* of replexes; an exact replica only increases failure resilience of a single replex. We call them hybrid replexes because they enable a middleground between adding either one or zero exact-copy replexes.

A hybrid replex is shared by replex A if h_{hybrid} is dependent on h_A. In the next few sections, we will explain how to define h_{hybrid} given the shared replexes.

Hybrid replexes are a building block for constructing a system with more complex failure amplification models per replex. To start with, we show how to construct a hybrid replex that is shared across two replexes.

3.3.1 2-Sharing

Consider replexes A and B from before. The system constructs a new, *hybrid* replex that is shared by A and B. Assume that all replexes have 4 partitions; in § 3.3.2 we will consider p partitions.

To define the hybrid replex, we must define h_{hybrid}. Assume that each partition in each replex in Figure 3.5 is numbered from left to right from 0-3. Then:

$$h_{hybrid}(r) = 2 \cdot (h_A(r) \pmod 2) + h_B(r) \pmod 2 \tag{3.1}$$

The graph in Figure 3.5 visualizes h_{hybrid}. The partition in the hybrid replex that stores row r is the partition connecting the partition in A and the partition in B that store r. Edges indicate which partitions in another replex share data with a given partition; in fact, if there exists a path between any two partitions, then those two partitions share data. Then any read that would have gone to a failed node can equally be serviced by visiting all partitions

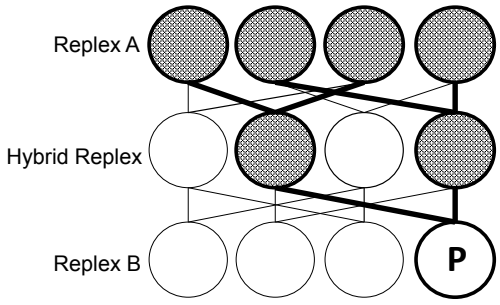


Figure 3.5: Each node is connected to exactly 2 nodes in another replex. This means that partitions in both replexes will see only 2x failure amplification after a single failure.

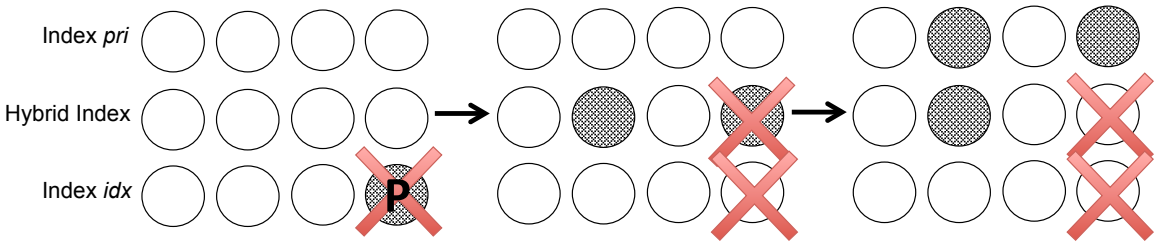


Figure 3.6: Graceful degradation. Shaded nodes indicate the nodes that must be contacted to satisfy queries that would have gone to partition P. As failures occur, Replex looks up replacement partitions for the failed node and modifies reads accordingly. Instead of contacting an entire replex after two failures, reads only need to contact a subset.

in an replex that are path-connected to the failed node.

For example, P shares data with exactly two partitions in the hybrid replex, and all four partitions in replex A. This means that when P fails, reads can either go to these two partitions in the hybrid replex or all four partitions in replex A, thereby experiencing 2x or 4x failure amplification, respectively. Then it is clear that reads should be redirected to the hybrid replex. Furthermore, because the hybrid replex overlaps attributes with replex B, any read redirected to the hybrid replex can be faster compared to a read that is redirected to replex A, which shares no attributes with replex B.

Figure 3.5 helps visualize how a partition in *any* replex will only cause failure amplification of two: each partition has an outcast of two to adjacent replexes. Hence by adding a single replex, we have reduced the failure amplification for *all* replexes after one failure. Contrast this with the extra replica approach: if we only add a single exact replica of replex

A, replex B would still experience 4x failure amplification after a single failure.

This hybrid technique might evoke erasure coding in the reader. However, as we explain in § 3.7, erasure coding solves a different problem. In erasure coding, parity bits are scattered across a cluster in *known* locations. The metric for the cost of a code is the reconstruction overhead after collecting all the parity bits. On the other hand, with replexes, there is no reconstruction cost, because replexes store full rows. Instead, hybrid replexes address the problem of *finding* data that is sharded by a different key in a different replex.

Hybrid replexes also smooth out the increase in failure amplification as failures occur. The hybrid approach introduces a recursive property that enables graceful read degradation as failures occur, as shown in Figure 3.6.

In Figure 3.6, reads to P are redirected as cascading failures happen. When P fails, the next smallest set of partitions— those in the hybrid replex— are used. If a partition in *this* replex fails, then the system replaces it in a similar manner. Then the full set of partitions that must be accessed is the three shaded nodes in the rightmost panel. Three nodes must fail concurrently before the worst set, all partitions in an replex, is used. The system is only fully unavailable for a particular read if after recursively expanding out these partition sets it cannot find a set without a failed node.

This recursion stops suddenly in the case of exact replicas. Suppose a user increases the failure resilience of A by creating an exact replica. As the first failure in A occurs, the system can simply point to the exact replica. When the second failure happens, however, reads are necessarily redirected to all partitions in B.

3.3.2 Generalizing 2-sharing

In general, we can parametrize a hybrid replex by n_1 and n_2, where $n_1 \cdot n_2 = p$ and p is the number of partitions per replex. Then:

$$h_{hybrid}(r) = n_2 \cdot (h_A(r) \pmod{n_1}) + h_B(r) \pmod{n_2} \tag{3.2}$$

Applying this to Figure 3.5, each partition in A would have an outcast of n_2 instead of two, and each partition in B would have an incast of n_1. Then when partitions in replex A fail, reads will experience n_2-factor amplification, while reads to partitions in replex B will experience n_1-factor failure amplification. The intuition is to think of each partition in the hybrid replex as a pair: (x, y), where $0 \leq x < n_1$ and $0 \leq y < n_2$. Then when a partition in replex A fails, reads must visit all hybrid partitions $(x, *)$ and when a partition in replex B fails, reads must visit all hybrid partitions $(*, y)$. The crucial observation is that $n_1 \cdot n_2 = p$, so the hybrid layer enables only $n_1, n_2 = O(\sqrt{p})$ amplification of reads during failure, as opposed to $O(p)$.

n_1 and n_2 become tuning knobs for a hybrid replex. A user can assign n_1 and n_2 to different replexes based on importance. For example, if $p = 30$, then a user might assign $n_1 = 5$ and $n_2 = 6$ to two replexes A and B that are equally important. Alternatively, if the workload mostly hits A, which means failures in A will affect a larger percentage of queries, a user might assign $n_1 = 3$ and $n_2 = 10$. Even more extreme, the user could assign $n_1 = 1$ and $n_2 = 30$, which represents the case where the hybrid replex is an exact replica of replex A.

3.3.3 More Extensions

In this section, we discuss intuition for further generalizing hybrid replexes. See § 3.6 for explicit construction, which requires defining complex h_{hybrid}.

Hybrid replexes can be shared across r replexes, not just two as presented in the previous sections. To decrease failure amplification across r replexes, we create a hybrid replex that is shared across these r replexes. To parametrize this space, we use the same notation used to generalize 2-sharing. In particular, think of each partition in the hybrid replex as an r-tuple: $(n_1, \ldots, n_r)$. Then when some partition in the qth replex fails, reads must visit all partitions $(*, \ldots, *, x_q, *, \ldots, *)$. Then failure amplification after one failure becomes $O(p^{\frac{r-1}{r}})$. As expected, if more replexes share a hybrid replex, improvement over $O(p)$

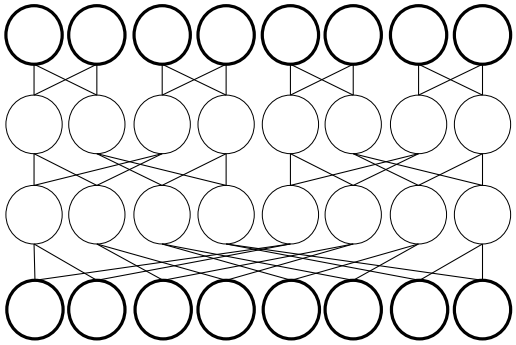


Figure 3.7: Inserting two hybrid replexes in between two replexes (in bold). Each node in the graph has outcast 2, which means after any partition fails, failure amplification will be at most 2x. After two failures, amplification will be 3x; after three, it will be 4x.

failure amplification becomes smaller.

For example, suppose a table requires 4 indices, which will be translated into 4 replexes. Then a hybrid replex is not necessary for replication, but rather can be inserted at the discretion of the user, who might want to increase read availability during recovery. Simply paying the costs of an additional 4-shared hybrid replex can greatly increase failure read availability.

We can also increase the number of hybrid replexes inserted between two replexes. For example, we can insert *two* hybrid replexes between every two desired replexes, as shown in Figure 3.7. Then two hybrid replexes enable $O(p^{1/3})$ amplification of reads during failure, at the expense of introducing yet another replex. If two replexes share k hybrid replexes, then there will be $O(p^{\frac{1}{k+1}})$ amplification of reads during failure. As expected, if two replexes share more hybrid replexes, the failure amplification becomes smaller. Furthermore, Figure 3.7 shows that adding more hybrid replexes enables better cascading failure amplification. The power of hybrid replexes lies in tuning the system to expected failure models.

3.4 Implementation

We implemented Replex on top of HyperDex, which already has a framework for supporting multi-indexed data. However, we could have implemented replexes and hybrid replexes on any system that builds indices for its data, including another NoSQL system

or an RDBMS such as MySQL Cluster. We added around 700 lines of code to HyperDex, around 500 of which were devoted to make data transfers during recovery performant.

HyperDex implements copies of the datastore as subspaces. Each subspace in HyperDex is associated with a hash function that shards data across that subspace's partitions. We replaced these subspaces with replexes, which can take an arbitrary sharding function. For example, in order to implement hybrid replexes, we initialize a generic replex and assign h to any of the h_{hybrid} discussed in § 3.3. HyperDex uses chain replication to replicate across subspaces; we modify this replication protocol with upstream ACKs, as described in § 3.2.3.

To satisfy a lookup query, Replex calculates which nodes are needed for lookup from the system configuration that is fetched from a coordinator node. A lookup is executed against any number of replexes, so Replex uses the sharding function of the respective replex to identify relevant partitions. The configuration tells Replex the current storage nodes and their status. We implemented the recursive lookup described in § 3.3.1 that uses the configuration to find the smallest set that contains all available partitions. For example, if there are no failures, then the smallest set is the original partitions. Replex implements this lookup functionality in the client-side HyperDex library. The client then sends the search query to all nodes in the final set and aggregates the responses; the client library waits to hear from all nodes before returning.

This recursive construction is used again in Replex's recovery code. In order to reconstruct a partition, Replex calculates a minimal set of partitions to contact and sends each member a reconstruction request with a predicate. The predicate can be thought of as matching on $h(r)$, where h is the sharding function of the replex to which the receiving partition belongs. When a node receives the reconstruction request, it maps the predicate across its local rows and only sends back rows that satisfy the predicate.

Finally, to run Replex, we set HyperDex's fault tolerance to $f = 0$.

3.5 Evaluation

Our evaluation is driven by the following questions:

- How does Replex's design affect steady-state index performance? (§ 3.5.1)
- How do hybrid replexes enable superior recovery performance? (§ 3.5.2)
- How can generalized 2-sharing allow a user to tune failure performance? (§ 3.5.3)
- How do hybrid replexes enable better resource tradeoffs with r-sharing? (§ 3.5.4)

Setup. All physical machines used had 8 CPUs and 16GB of RAM running Linux (3.10.0-327). All machines ran in the same rack, connected via 1Gbit links to a 1Gbit top-of-rack switch. 12 machines were designated as servers, 1 machine was a dedicated coordinator, and 4 machines were 64-thread clients. For each experiment, 1 or 2 additional machines were allocated as recovery servers.

System	Failures Tolerated	Replication Factor
Replex-2	1	2x
Replex-3	2	3x
HyperDex	2	6x

Table 3.1: Systems evaluated.

3.5.1 Steady-State Performance

To analyze the impact of replacing replicas with replexes, we report operation latencies in Replex. We specify a table in Replex with two indices: the primary index and a secondary index. We configure Replex to build a single hybrid replex, so Replex builds 3 full replexes during the benchmark; we call this system Replex-3 in Table 3.1. Because Replex-3 builds 3 replexes, data is tolerant to 2 failures. Hence, we also set HyperDex to three-way replicate data objects.

Read latency for Replex is identical to HyperDex's, because reads are simply done on the primary index of both systems; we report the read CDF in Figure 3.9. More importantly, the insert latency for Replex is consistently 2x less than the latency of a HyperDex insert,

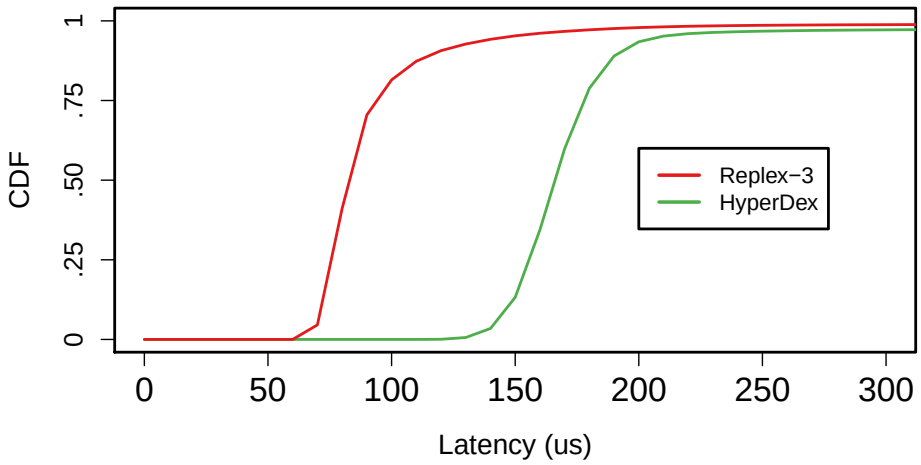


Figure 3.8: Insert latency microbenchmark CDF

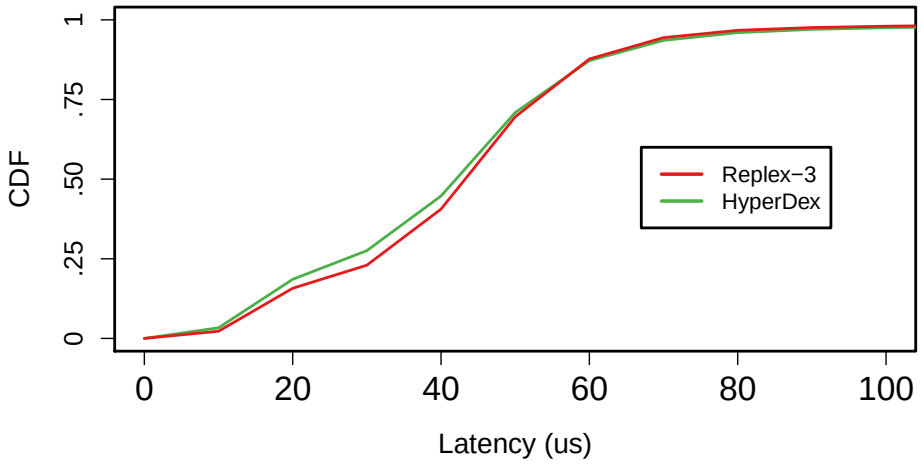


Figure 3.9: Read latency microbenchmark CDF

as in Figure 3.8. This is because one Replex insert visits 3 partitions while one HyperDex insert visits $2 \cdot 3 = 6$ partitions; these values are the replication factor denoted in Table 3.1. In fact, the more indices a user builds, the larger the factor of difference in latency inserts. This helps to demonstrate Replex's scalability compared to HyperDex.

Figure 3.10 reports results from running a full YCSB benchmark on 3 systems: Cassandra, HyperDex, and Replex-3; Yahoo Cloud-Serving Benchmark (YCSB) is a well established benchmark for comparing NoSQL stores [42]. Cassandra is a widely-used distributed key-value store that is also backed by a log-structured merge tree, similar to LevelDB [7], which backs HyperDex.

Name	Workload	Total Operations
Load	100% Insert	10 M
A	50% Read/50% Update	500 K
B	95% Read/5% Update	1 M
C	100% Read	1 M
D	95% Read/5% Insert	1 M
E	95% Scan/5% Insert	10 K
F	50% Read/50% Read-Modify	500 K

Table 3.2: YCSB workloads.

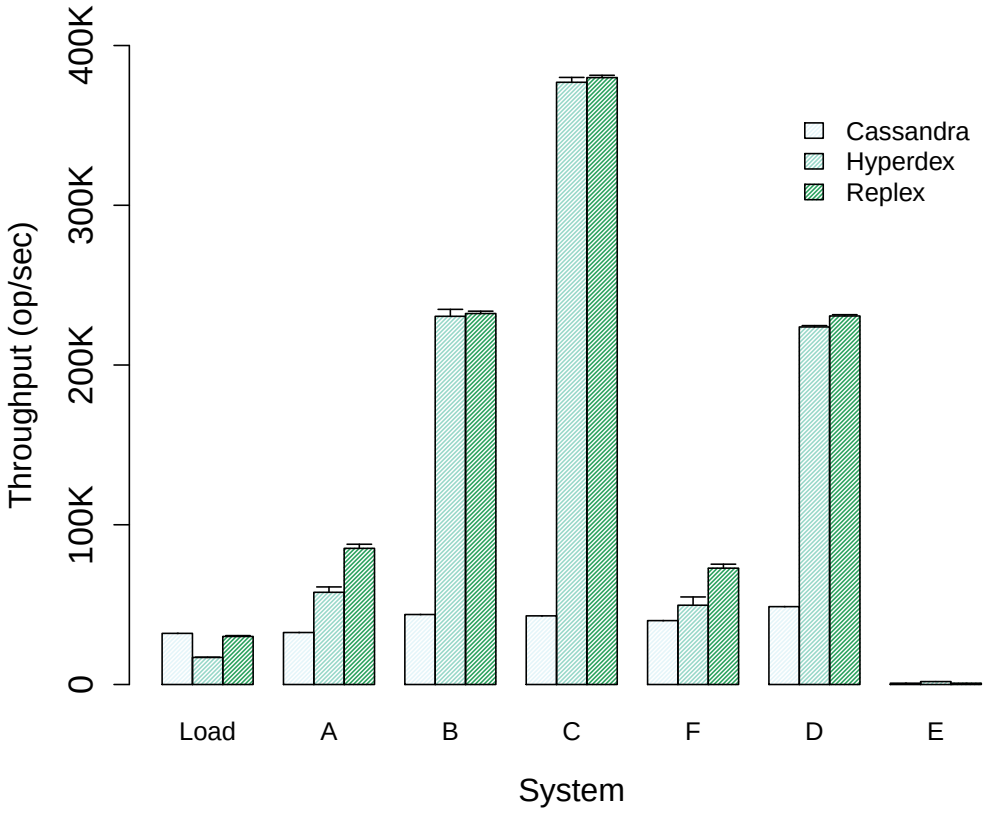


Figure 3.10: Mean throughput for full YCSB suite over 3 runs. Error bars indicate standard deviation. Results grouped by workload, in the order they are executed in the benchmark.

We present Cassandra results for baseline comparison because it is another high performing, distributed key-value store. Because Replex-3 is tolerant to 2 failures, we also set Cassandra and HyperDex to three-way replicate data objects. In the load phase, YCSB inserts 10 million 100 byte rows into the datastore. In the rest of the workloads, we measure

throughput by running enough client threads to saturate throughput.

Replex-3's lower latency insert operations enable higher throughput than HyperDex on the load portion and Workloads A, F; these are the workloads with inserts/updates. Workload C has comparable performance to HyperDex, because these reads can be performed on the index that HyperDex builds. Cassandra has comparable load throughput to Replex-3 because writes are replicated in the background; Cassandra writes return after visiting a single replica while our writes return after visiting all 3 replexes for full durabilty. However, for the rest of the workloads, Cassandra performs poorly because reads in Cassandra must visit 3 replicas in order to achieve consistency. HyperDex and Replex-3 do not pay this read penalty because they visit all 3 replicas at insert time.

3.5.2 Failure Evaluation

In this section, we examine the throughput of three systems as failures occur: 1) HyperDex with two subspaces, 2) Replex with two replexes (Replex-2), and 3) Replex with two replexes and a hybrid replex (Replex-3). Each system has 12 virtual partitions per subspace or replex. One machine is reserved for reconstructing the failed node. Each system automatically assigns the 12 virtual partitions per replex across the 12 server machines.

For each system we specify a table with a primary and secondary index. We run two experiments, one that loads 1 million rows of size 1KB bytes and one that loads 10 million rows of size 100 bytes; the second experiment demonstrates recovery behavior when CPU is the bottleneck. We then start a microbenchmark where clients read as fast as possible against both indices. Reads are split 50:50 between the two indices. We kill a server after 25 seconds. Figure 3.11 shows the read throughput in the system as a function of time, and Tables 3.3 and 3.4 report average recovery statistics.

Recovery time in each system depends on the *size* of the data loss, which depends on how much data is stored on a physical node. The number of storage nodes is a constant across all three systems, so the amount of data stored on each node is proportional to the

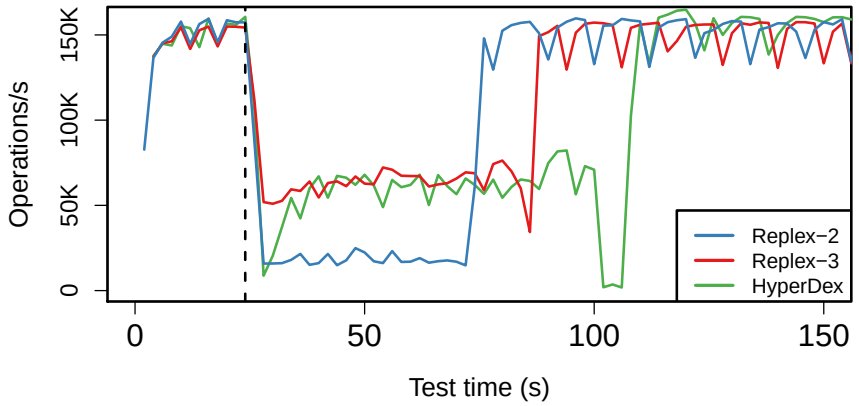


Figure 3.11: We crash a server at 25s and report read throughput for Replex-2, Replex-3, and Hyperdex. Systems are loaded with 10 million, 100 byte rows. All three systems experience a dip in throughput right before returning to full functionality due to the cost of reconfiguration synchronization, which introduces the reconstructed node back into the system configuration.

total amount of data across all replicas; recovery times in Tables 3.3 and 3.4 are approximately proportional to the Replication Factor column in Table 3.1. By replacing replicas with replexes, Replex can reduce recovery time by 2-3x, while also using a fraction of the storage resources.

Interestingly, Replex-2 recovers the fastest out of all systems, which suggests the basic Replex design has performance benefits even without adding hybrid replexes.

Recovery throughput shows one of the advantages of the hybrid replex design. In Table 3.3, Replex-2 has minimal throughput during recovery, because each read to the failed node must be sent to all 12 partitions in the other replex. These same 12 partitions are also responsible for reconstructing the failed node; each of the partitions must iterate through their local storage to find data that belongs on the failed node. Finally, these 12 partitions are still trying to respond to reads against the primary index, hence system throughput is hijacked by reconstruction throughput and the amplified reads. Replex-2 throughput is not as bad in Table 3.4, because 1 million rows does not bottleneck the CPU during recovery.

The Replex-3 alleviates the stress of recovery by introducing the hybrid replex. First, each read is only amplified 3 times, because the grid constructed by the hybrid replex has

System	Recovery Time (s)	Recovery Throughput (op/s)
Replex-2	50 ± 1	$18{,}989 \pm 1{,}883$
Replex-3	60 ± 1	$65{,}780 \pm 3{,}839$
HyperDex	105 ± 17	$34{,}697 \pm 19{,}003$

Table 3.3: Recovery statistics of one machine failure after 25 seconds. 10 million, 100 byte records. Results reported as average time $\pm$ standard deviation of 3 runs.

System	Recovery Time (s)	Recovery Throughput (op/s)
Replex-2	6.7 ± 0.57	$70{,}084 \pm 5{,}980$
Replex-3	8.7 ± 0.56	$110{,}280 \pm 11{,}232$
HyperDex	20.0 ± 2.65	$127{,}232 \pm 85{,}932$

Table 3.4: Recovery statistics of one machine failure after 25 seconds. 1 million, 1KB records. Results reported as average time $\pm$ standard deviation of 3 runs.

dimensions $n_1 = 3, n_2 = 4$. Second, only 3 partitions are responsible for reconstructing the failed node. In fact, in both experiments, Replex-3 achieves recovery throughput comparable to that of HyperDex, which has no failure amplification, whilst adding little recovery time.

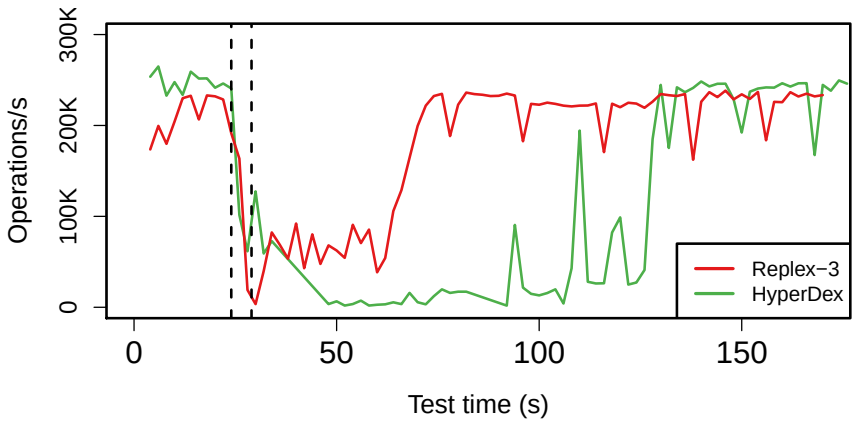


Figure 3.12: Read throughput after two failures. We crash one server at 25s and then a second at 30s. Request pile-up because throughput is used for recovery is responsible for the jumps in HyperDex throughput.

Finally, we highlight the hybrid replex design by running an experiment that causes two cascading failures. Replex-2 only tolerates two failures, so we do not include it in this

System	Recovery Time (s)	Recovery Throughput (op/s)
Replex-3	37.6 ± 1.2	$60{,}844 \pm 27{,}492$
HyperDex	98.0 ± 11	$30{,}220 \pm 8{,}104$

Table 3.5: Recovery statistics of two machine failures at 25s and 30s. Results reported as average time $\pm$ standard deviation of 3 runs. Recovery time is measured from the first failure.

experiment. Figure 3.15 shows the results when we run the same 50:50 read microbenchmark and crash a node at 25s and 30s. We reserve an additional 2 machines as spares for reconstruction. We run the experiment where each system is loaded with 1 million, 1K rows.

Figure 3.15 stresses the advantages of graceful degradation, enabled by the hybrid replex. We observe that experiencing two failures more than quadruples the recovery time in HyperDex. This is because the two reconstructions occur sequentially and independently. In Replex-3, failing a second partition causes reduced recovery throughput, because the second failed partition must rebuild from partitions that are actively serving reads. However, recovery time is bounded because reconstruction of the failed nodes occurs in parallel. When the second failed partition recovers, throughput nearly returns to normal.

3.5.3 Parametrization of the Hybrid Replex

As discussed in § 3.3.2, any hybrid replex $\mathcal{H}$ can be parametrized as (n_1, n_2). Consider the Replex-3 setup, which replicates operations to replexes in the order $A \rightarrow \mathcal{H} \rightarrow B$. If $\mathcal{H}$ is parametrized by (n_1, n_2), then failure of a partition in B will result in n_1-factor read amplification, and a failure in A will result in n_2-factor read amplification. In this section we investigate the effect of hybrid replex parameterization on throughput under failure.

We load each parametrization of Replex-3 with 1 million 1KB entries and fail a machine at 25s. Four separate client machines run an $a : b$ read benchmark, where a percent of reads go to replex A and b percent of reads go to replex B. Figure 3.13 shows the throughput results when a machine in B is killed at 25s. We report the throughput results for all three workloads to indicate that parametrization trends are independent of workload.

As expected, parametrizing $\mathcal{H}$ with $(1, 12)$ causes the least failure amplification, hence throughput is relatively unaffected by the failure at 25s. As n_1 grows larger, throughput grows steadily worse during the failure, because failure amplification becomes greater.

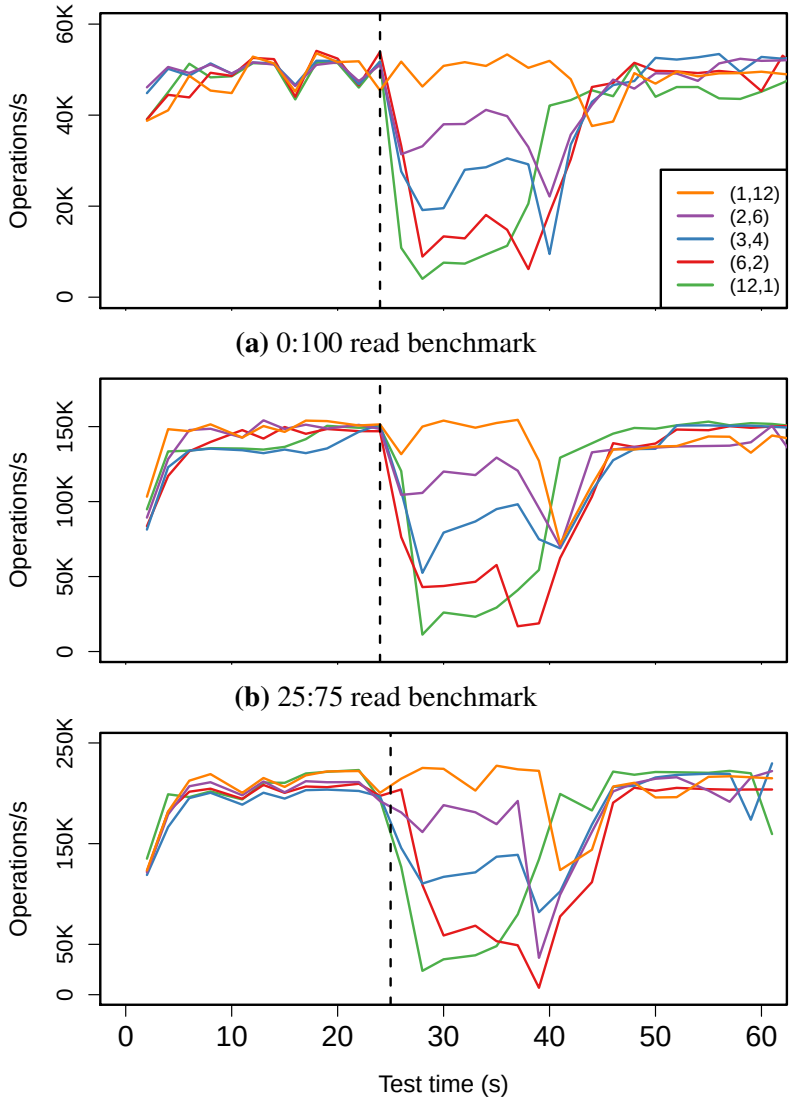
We also point out that as the benchmark contains a larger percentage of reads in replex A, steady-state throughput increases (note the different Y-axis scales in Figure 3.13). This is because of the underlying LevelDB implementation of HyperDex. LevelDB is a simple key-value store with no secondary index support; reads on replex A are simple LevelDB gets, while reads to replex B become LevelDB scans. To achieve throughput as close to native gets as possible, we optimized point scans to act as gets to replex B, but the difference is still apparent in the throughput. Fortunately, this absolute difference in throughput does not affect the relative trends of parametrization.

The tradeoff from one parametrization to the next is throughput during failures in A. As an example, Figure 14 shows the throughput results when a machine in replex A is killed after 25s, with a 25:75 read workload. The performance of the parametrizations is effectively reversed. For example, even though $(1, 12)$ performed best during a failure in B, it performs worst during a failure in A, in which failure amplification is 12x. Hence a user would select a parametrization based on which replex's failure performance is more valued.

3.5.4 Evaluating 3-Sharing

In the previous sections, all systems evaluated assumed 3-way replication. In particular, in Replex, if the number of indices i specified by a table is less than 3, then Replex can build $3 - i$ hybrid replexes for free, by which we mean those resources must be used anyway to achieve 3-way replication.

When $i \geq 3$, resource consumption from additional hybrid replexes becomes more interesting. No longer is a hybrid replex inserted to achieve a replication threshold; rather, a hybrid replex is inserted to increase recovery throughput, at the expense of an additional



(a) 0:100 read benchmark

(b) 25:75 read benchmark

(c) 50:50 read benchmark

Figure 3.13: We crash a machine at 25s. Each graph shows read throughput for Replex-3 with five different hybrid parametrizations and the labelled workload. Although $(12, 1)$ has the worst throughput during failure, it recovers faster than the other parametrizations because recovery is spread across more partitions.

storage replica. Consider $i = 3$ and suppose a user only wishes to add a single hybrid replex, because of resource constraints. One way to maximize the utility of this hybrid replex is through 3-sharing, as described in § 3.3.3. Of course, depending on the importance of the three original indices, 2-sharing is also an option, but this is already explored in the previous sections. For sake of evaluation, we consider 3-sharing in this section.

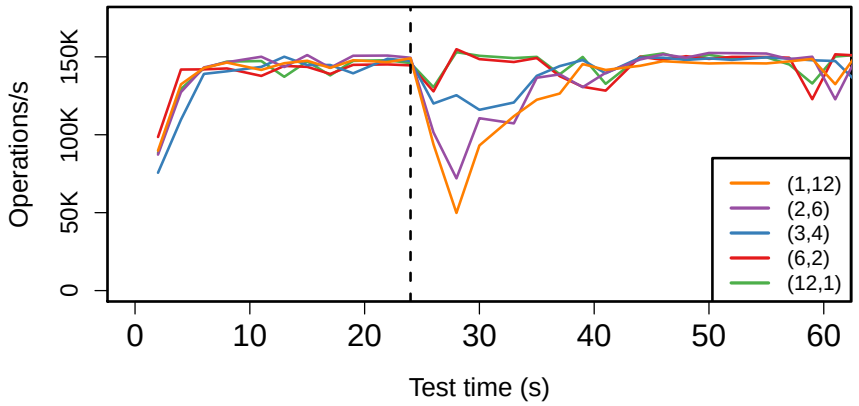


Figure 3.14: Replex-3 throughput with a 25:75 read benchmark. We crash a machine in replex A at 25s.

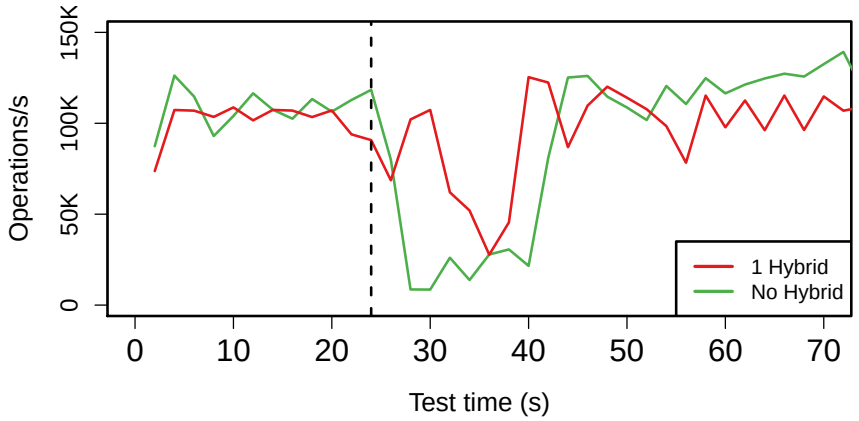


Figure 3.15: Read throughput after a failure at 25s with a 33:33:33 benchmark.

The system under evaluation has 3 replexes, A, B, C, and 1 hybrid replex that is 3-shared across the original replexes. The hybrid replex is parametrized by $n_1 = 3$, $n_2 = 2$, $n_3 = 2$. Again, we load 1 million 1KB entries and fail a node at 25s. Four seperate client machines run a read benchmark spread equally across the indices. Figure 15 shows the throughput results, compared to a Replex system without a hybrid replex.

Again, if there is no hybrid index, then recovery throughput suffers because of failure amplification. As long as a single hybrid index is added, the recovery throughput is more than doubled, with little change to recovery time. This experiment shows in the power of hybrid replexes in tables with more indices: as the number of indices grows, the fractional cost of adding a hybrid replex decreases, but the hybrid replex can still provide enormous

# Hybrids	Recovery Time (s)	Recovery Throughput (op/s)
0	14.7 ± 0.58	$5{,}831 \pm \quad 678$
1	13.0 ± 0	$14{,}569 \pm 6{,}087$

Table 3.6: Recovery statistics for Replex systems with 3 replexes and different numbers of hybrid replexes. One machine is failed after 25 seconds. Results reported as average time $\pm$ standard deviation of 3 runs.

gains during recovery.

3.6 Discussion and Extensions

In this section, we discuss how to construct a variety of hybrid replexes explicitly by specifying h_{hybrid} functions. We also show how these constructions can be used to design systems with particular requirements.

Recall that in § 3.3.2 we generalize 2-sharing by showing how to parametrize h_{hybrid} based on factors of p, where p is the number of partitions per replex. Then we can summarize the parametrization of 2-sharing (between replexes A and B) with the following table:

$h_B(r) \pmod{n_2}$ \ $h_A(r) \pmod{n_1}$	0	1	2	…	$a-1$
0	0	1	2	…	$n_1 - 1$
1	a	$a+1$	$a+2$	…	$2a-1$
⋮	⋮	⋮	⋮	⋱	
$n_2 - 1$	$n_1(n_2-1)$	$n_1(n_2-1)+1$			$n_1 n_2 - 1$

Table 3.7: Let $n_1 \cdot n_2 = p$. The table cells specify $h_{hybrid}(r)$, given $h_B(r) \pmod{n_2}$ and $h_A(r) \pmod{n_1}$. Recall that $h_{hybrid}(r)$ should define a partition number, out of p, that replicates some data object.

Table 3.7 helps visualize how, given h_{hybrid}, which partitions will be needed for the recovery of any partition in the original replexes or in the hybrid replex. In particular, column i in the grid is precisely the set of hybrid partitions that are needed to recover a partition in replex A with label $i \pmod{n_1}$. Similarly, row j in the grid is precisely the set of hybrid partitions needed to recover a partition in replex B with label $j \pmod{n_2}$.

Each row has n_1 entries and each column has n_2 entries, hence when partitions in A fail, reads will experience n_2-factor amplification, while reads to partitions in A will experience n_1-factor amplification on failure. The crucial observation is that $n_1 \cdot n_2 = p$, so the hybrid layer enables only $n_1, n_2 = O(p^{1/2})$ amplification of reads during failure, as opposed to p in the strawman case.

3.6.1 m-sharing

Thus far, we have only discussed hybrid replexes that are shared across two replexes, which leads to two-dimensional tables. Presenting h_{hybrid} as a table shows that we can further parametrize hybrid functions by extending the number of dimensions in the table. Namely, we can construct a single hybrid index that combines m regular replexes, which would result in a m-dimensional table. Because it's not possible to depict a m-dimensional table, here we just present the explicit characterization of h_{hybrid}.

Suppose $p = \prod_{i=1}^{m} n_i$, where i denotes one of the m replexes used to create the hybrid replex. Then define:

$$h_{hybrid}(r) = \sum_{i=1}^{m} \left((h_i(r) \pmod{n_i}) \cdot \prod_{j=1}^{i-1} n_j \right) \tag{3.3}$$

The reader can verify that letting $m = 2$ yields Equation 3.2. Inspection of Equation 3.3 reveals that recovery of any partition in replex i requires visiting $\frac{p}{n_1}$ partitions. Hence, read amplification of the m original replexes during failure becomes $O(p^{\frac{m-1}{m}})$.

3.6.2 Multi-dimensional Indexing

In order for an m-shared hybrid to achieve fast local access, we also observe that the local indices built at a partition of an m-shared hybrid replex should be a multi-dimensional index. There are a variety of known multi-dimension indexing structures, such as Quadtrees [115], R-trees [56, 101], and K-d trees [32, 51]. Briefly, all of these structures can query data sets by an n-tuple, rather than by a single key, which is a limitation of the BTree.

3.6.3 Constructing replexes given constraints

Hybrid replexes constructed from m regular indices are particularly useful for the following scenario. In theory, Replex can support an arbitrary number of secondary indices. However, suppose a system operator wants to support i secondary indices but wants only r replicas of each data object, where $i > r$. Keep in mind that each additional replex introduces both a storage and update overhead. If $i = r$, then supporting indices does not impose overhead beyond what traditional replication would have. If $i < r$, then the Replex system even has additional resources to work with. In particular, these resources can be used to construct any of the hybrid schemes discussed.

If $i > r$, when we need a way to collapse multiple indices into a single replex. This is precisely the mechanism provided by the m-sharing replex. For example, suppose $i = 5$ and $r = 3$. Then we can support all 5 indices on only 3 replexes by collapsing 3 indices into a single 3-shared hybrid replex. Then the final system would have two replexes, each of which corresponds to a unique index, and a third replex that is a 3-shared hybrid replex.

We can combine these hybrid schemes in any way in order to build the final system. For example, another way to construct the final system is to have one replex which corresponds to a unique index, and one replex each that corresponds to a 2-shared hybrid replex. Yet another construction is to have one replex which corresponds to a unique index (for example, the index that the system builder expects to be queried most often out of the indices), one replex correspond to a 4-shared hybrid replex, and the last replex correspond to a hybrid construction between the first two replexes.

Each of these examples, which we summarize in Table 3.8, shows how flexible the hybrid scheme is: depending on the steady-state and failure-state performance requirements of the system, the system builder can construct replexes accordingly.

	Construction	Explanation
replex 1	index A	Index A is a popular index that needs its own replex
replex 2	index B	Index B is a popular index
replex 3	3-shared hybrid replex across C, D, E	Indexes C, D, E are less frequently used, so they are coalesced into a single hybrid replex

(a)

	Construction	Explanation
replex 1	index A	Index A is a popular index that needs its own replex
replex 2	2-shared hybrid replex across indices B and C	Indexes B and C are less frequently accessed, so they can share access via a hybrid replex
replex 3	2-shared hybrid replex across indices D and E	Indexes D and E are less frequently accessed

(b)

	Construction	Explanation
replex 1	index A	Index A is a popular index that needs its own replex
replex 2	4-shared hybrid replex across indices B, C, D, E	Indexes B, C, D, E are less frequently used, so they are coalesced into a single hybrid replex
replex 3	a hybrid replex built from replex 1 and 2	Suppose the recovery performance of index A is also important. Then this hybrid replex will enable better recovery performance.

(c)

Table 3.8: Here we summarize three of the many ways we can construct a system where there are $i = 5$ desired indices but only $r = 3$ replicas of each data object are requested.

3.7 Related Work

3.7.1 Erasure Coding

Erasure coding is a field of information theory which examines the tradeoffs of transforming a short message to a longer message in order to tolerate a partial erasure (loss) of the message. LDPC [113], LT [88], Online [84], Raptor [102], Reed-Solomon [96], and Tor-

nado [81] are examples of well-known erasure codes which are used today. Hybrid replexes also explore the tradeoff between adding storage and network overheads and recovery performance. Recently, specific failure models have been applied to erasure coding to produce even more compact erasure codes [58]. Similarly, hybrid replex construction allows fine tuning given a workload and failure model.

3.7.2 Multi-Index Datastores

Several multi-index datastores have emerged as a response to the limitations of the NOSQL model. These datastores can be broadly divided into two categories: those which must contact every partition to query by secondary index, and those which support true, global secondary indices. Cassandra [111], CouchDB [28], Hypertable [63], MongoDB [41], Riak [65] and SimpleDB [40] are examples of of the former approach. While these NOSQL stores are easy to scale since they only partition by a single "sharding" key, querying by secondary index can be particularly expensive if there is a large number of partitions. Some of these systems alleviate this overhead through the use of caching, but at the expense of consistency and overhead of maintaining the cache.

Unlike the previous NOSQL stores, Hyperdex [49] builds a global secondary index for each index, enabling efficent query of secondary indices. However, each index is also replicated to maintain fault tolerance, which comes with a significant storage overhead. As we saw in § 3.5, this leads to slower inserts and significant rebuild times on failure.

3.7.3 Relational (SQL) Databases

Traditional relational databases build multiple indices and auxillary data structures, which are difficult to partition and scale. Sharded MySQL clusters [42, 97] are an example of an attempt to scale a relational database. While it supports fully relational queries, it is also plagued by performance and consistency issues [42, 114]. For example, a query which involves a secondary index must contact each shard, just as with a multi-index datastore.

Yesquel [22] provides the features of SQL with the scalability of a NOSQL system. Like

Hyperdex, however, Yesquel separately replicates every index.

3.7.4 Other Data stores

Corfu [30], Tango [31], and Hyder [33] are examples of data stores which use state machine replication on top of a distributed shared log. While writes may be written to different partitions, queries are made to in-memory state, which allows efficient strongly consistent queries on multiple indices without contacting any partitions. However, such an approach is limited to state which can fit in the memory of a single node. When state cannot fit in memory, it must be partitioned, resulting in a query which must contact each partition.

3.8 Conclusion

Programmers need to be able to query data by more than just a single key. For many NoSQL systems, supporting multiple indices is more of an afterthought: a reaction to programmer frustration with the weakness of the NoSQL model. As a result, these systems pay unnecessary penalties in order to support querying by other indices.

Replex reconsiders multi-index data stores from the bottom-up, showing that implementing secondary indices can be inexpensive if treated as a first-order concern. Central to achieving negligible overhead is a novel replication scheme which considers fault-tolerance, availability, and indexing simultaneously. We have described this scheme and its parameters and have shown through our experimental results that we outperform HyperDex and Cassandra, state-of-the-art NoSQL systems, by as much as $10\times$. We have also carefully considered several failure scenarios that show Replex achieves considerable improvement on the rebuild time during failure, and consequently availability of the system. In short, we have demonstrated not only that a multi-index, scalable, high-availability NoSQL datastore is possible, it is the better choice.

Chapter 4

DIRECT

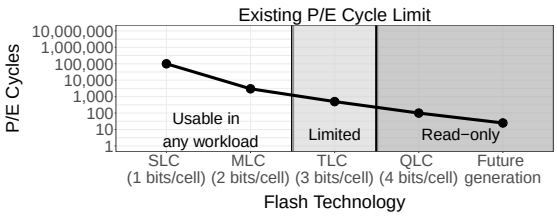
4.1 Introduction

Flash has become the dominant storage medium for hot data in datacenters [86, 99], since it offers significantly lower latency and higher throughput than hard disks. Many storage systems are built atop flash, including databases [7, 11, 17, 57], caches [6, 47, 74, 75, 108], and file systems [61, 92].

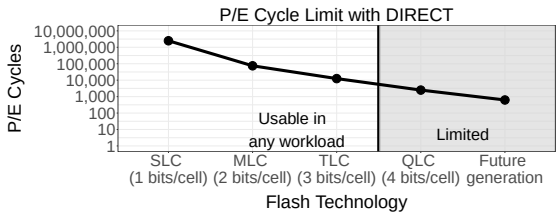
However, a perennial problem of flash is its limited endurance, or how long it can reliably correct raw bit errors. As device writes are the main contributor to flash wear, its lifetime is measured in the number of writes or program-erase (P/E) cycles the device can tolerate before exceeding an uncorrectable bit error threshold. Uncorrectable bit errors are errors that are exposed externally and occur when there are too many raw bit errors for the device to correct.

In hyper-scale datacenters, operators constantly seek to reduce flash wear by limiting flash writes [25, 86]. At Facebook,[1] for example, a dedicated team monitors application writes to ensure they do not prematurely exceed manufacturer-defined device lifetimes. To make matters worse, each subsequent flash generation tolerates a smaller number of writes before reaching end-of-life (see Figure 4.1a) [54]. Further, given the scaling challenges of

[1] Facebook is a major web company. Name anonymized for submission.



(a) Existing hardware-based error correction.



(b) Augmenting existing error correction with DIRECT.

Figure 4.1: For each generation of flash bit density, the average number of P/E cycles after which the uncorrectable bit error rate falls below the manufacturer specified level (10^{-15}). Beyond MLC, the number of flash writes the application can issue is limited [37]. With current hardware-based error correction, with QLC technology and beyond, flash can only be used for applications that are effectively read-only [26, 85, 89]. DIRECT enables the adoption of denser flash technologies because errors can be handled by the distributed storage application. The uncorrectable bit error rate that can be tolerated by DIRECT was computed using the model from § 4.3.1, while the uncorrectable bit error rate to P/E conversion was computed using data from a Google study [99].

DRAM [62, 73] and the increasing cost gap between DRAM and flash [2, 48, 50], many operators are migrating services from DRAM to flash [8, 48], increasing the pressure on flash lifetime.

There is a variety of work that attempts to extend flash lifetime by delaying the onset of bit errors [7, 13, 38, 47, 60, 76, 78, 80, 82, 107, 120, 121]. This paper takes a contrarian approach. We observe that flash endurance can be extended by *allowing* devices to go beyond their advertised uncorrectable bit error rate (UBER) and embracing the use of flash disks that exhibit much higher error rates; Google recently released a whitepaper suggesting a similar approach [36]. We can do so without sacrificing durability because (1) datacenter storage systems replicate data on remote servers, and (2) this redundancy can correct bit error rates orders of magnitude beyond the hardware error correction mechanisms implemented on the device. However, the challenge with higher flash error rates is maintaining availability and correctness.

We introduce Distributed error Isolation and RECovery Techniques (DIRECT), which is a set of three simple general-purpose techniques that, when implemented, enable distributed storage systems to achieve high availability and correctness in the face of uncorrectable bit errors:

1. **Minimize error amplification.** DIRECT detects errors using existing error detection mechanisms (e.g., checksums) and recovers data from remote servers at the smallest possible granularity.
2. **Local metadata protection.** A corruption in local metadata (e.g., database index), often requires a large amount of data to be re-replicated. DIRECT avoids this by adding redundancy locally to local metadata.
3. **Safe recovery semantics.** Any recovery operations on corrupted data must be serialized against concurrent read and write operations with respect to the system's consistency guarantees.

The difficulty of implementing DIRECT techniques depends on two properties of the underlying storage system. The first property affects the difficulty of minimizing error amplification: whether the distributed storage system is physically-replicated or logically-replicated. Physically-replicated systems replicate *data blocks* or objects between servers, while logically-replicated systems replicate the *commands* issued concurrently to the storage systems (e.g., write, update, delete). In physically-replicated systems, a certain object is stored in the same block or file on another server, which makes it straightforward to minimize the amount of data needed to recover: just rereplicate the identical data block. This assumption does not hold for most logically-replicated systems, however, where there is no guarantee that physical blocks will be identical across different replicas.

The second property affects the difficulty of maintaining safe recovery: whether the data store supports versioning. In systems without versioning, a corrupt data object can simply be rewritten. In contrast, in systems with versioning, we need to guarantee the recovered object does not override a more up-to-date version.

We demonstrate how to generalize DIRECT techniques by implementing them in two popular systems that are representative of these different classes of storage systems: (1) the Hadoop Distributed File System (HDFS), which is a physically-replicated storage sys-

tems without versioning, and (2) ZippyDB,[2] a distributed system that implements logical replication on top of RocksDB, a popular key-value store that supports key versioning.

Objects in HDFS are stored in the same physical blocks across replicas, so it is relatively straightforward for DIRECT to find the corrupt object in another replica and recovery it at a high granularity (§ 4.4.1). On the other hand, recovery is challenging in ZippyDB since it is difficult not only to find the corrupted region in one replica in another replica (different servers store the same key-value pairs in different files), but also to ensure that the recovered key-value pairs have consistent versions ZippyDB (§ 4.4.2).

Applying DIRECT results in significant end-to-end availability improvements: it enables HDFS to tolerate bit error rates that are 10,000×-100,000× greater, reduces application-visible error rates in ZippyDB by more than 100×, and speeds up recovery time in ZippyDB by 10,000× over the existing system.

DIRECT leads to significant increases in device lifetime, because these performance improvements not only maintain the same probability of application-visible errors (durability)— for the computation, see § 4.3.1— but also minimize overhead of fixing corruption errors (availability) at much higher device UBERs. An estimate of lifetime increase is shown in Figure 4.1b; we estimate the number of P/E cycles gained by running at higher UBERs from a Google study [99]. Depending on workload parameters and hardware specifications, DIRECT can increase the lifetime of devices by 10-100×. This allows datacenter operators to replace flash devices less often and adopt lower cost-per-bit flash technologies that have lower endurance. DIRECT also provides the opportunity to rethink the design of existing flash-based storage systems, by removing the assumption that the device fixes all corruption errors. Furthermore, while this paper focuses on flash, DIRECT's principles also apply in other storage mediums, including NVM and hard disks.

In summary, this paper makes several contributions:

1. We observe that flash lifetime can be extended by allowing devices to expose higher

[2]ZippyDB has been used at large-scale production at Facebook for 5 years.

bit error rates.

2. We propose DIRECT, general-purpose software techniques that enable storage systems to maintain performance and high availability in the face of high hardware bit error rates.
3. We design and implement DIRECT in two storage systems, HDFS and ZippyDB, that are representative of physical and logical replication.
4. We demonstrate that DIRECT significantly speeds up recovery time due to flash corruptions and significantly lowers application-observable errors in the resulting systems, allowing them to tolerate much higher hardware bit error rates.

4.2 Motivation

What Limits Flash Endurance? Flash chips are composed of memory cells, each of which stores an analog voltage value. The flash controller reads the value stored in a certain memory cell by sensing the voltage level of the cell and applying quantization to determine the discrete value in bits. The more bits stored in a cell, the narrower the voltage range that maps to each discrete bit, so more precise voltage sensing is required to get a correct read. Unfortunately, one of the primary ways to reduce cost per bit is to increase the number of bits per cell, which means that even small voltage perturbations can result in a misread.

Multiple factors cause voltage drift in a flash cell. The dominant source, especially in datacenter settings where most data is "hot," is the program-erase (P/E) cycle, which involves applying a large high voltage to the cell in order to drain its stored charge, thus wearing the insulating layer in the flash cell [38]. This increases the voltage drift in subsequent values in the cell, which gradually leads to bit errors.

3D NAND is a recent technology that has been adopted for further increasing flash density by stacking cells vertically. While 3D NAND relaxes physical limitations of 2D NAND (traditional flash) by enabling vertical stacking, 3D NAND inherits the reliability problems of 2D NAND and further exacerbates them, since a cell in 3D NAND has more

adjacent (vertical) neighbors. For example, voltage retention is worse, because voltage can now leak in three dimensions [71, 82, 87]. Similarly, disturb errors that occur when adjacent cells are read or programmed are also exacerbated [64, 105].

Existing Hardware Reliability Mechanisms and Limitations. To correct bit errors, flash devices use error correcting codes (ECC), which are implemented in hardware. After the ECC pass, there could still be incorrect bits on the page. To address this, SSDs also employ internal RAID across the dies of a flash device [18, 21]. After applying coding and RAID within the device, there will remain a certain rate of *uncorrectable bit errors* (UBER). Together, ECC and internal RAID mechanisms can drive the error rates of SSDs from the raw bit error rate of around 10^{-6} down to the 10^{-17} to 10^{-20} UBER range typical of enterprise SSDs [15]. "Commodity" SSD devices typically guarantee an UBER of 10^{-15}.

While it is possible to create stronger ECC engines, the higher the corrective power of the ECC, the more costly the device due to the complexity of the ECC circuit [5, 9]. Furthermore, the level of internal RAID striping is constant across generations, because the number of dies inside a flash device remains constant. This means that the corrective power of RAID is fixed.

Similarly, while RAID across devices [68, 94, 104] can add redundancy, a main design goal of DIRECT is to avoid adding extra, unnecessary overhead. We avoid turning to RAID because it imposes storage overheads, adds write overhead, and is inflexible because its recovery power is fixed at deployment time. Instead, we seek to correct error using existing redundancy in the storage stack.

Implications of Limited Flash Endurance. Flash technology has already reached the point where its endurance is inhibiting its adoption and operation in various datacenter use cases. For example, QLC was recently introduced as the next generation flash cell technology. However, it can only tolerate 100-200 P/E cycles [26, 85, 89], so it can only be used for read-heavy use cases, e.g., a 1 TB QLC drive with a lifetime of 100 P/E cycles can only write at a rate of 1 MB/s or less in order to preserve its advertised lifetime of 3 years.

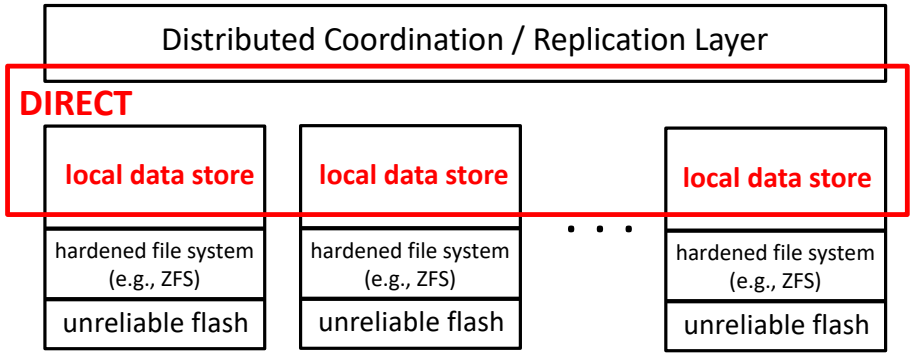
But as datacenter applications like databases and analytics that deal with hot data typically need to update objects frequently, the adoption of QLC has been more limited (and is the reason that Facebook has avoided QLC flash). Subsequent cell technology generations will suffer from even greater problems.

Operational issues also often dictate a device's usage lifetime. Flash is typically only used for its advertised lifetime to simplify operational complexity [99]. Further, in a hyperscale datacenter where it is common to source devices from multiple vendors, the most conservative estimate of device lifetime across vendors is typically chosen as the lifetime for a fleet of flash devices, so that the entire fleet can be installed and removed together. If the distributed storage layer could tolerate much higher device error rates, then datacenter operators would no longer have to make conservative and wasteful estimates about entire fleets of flash devices.

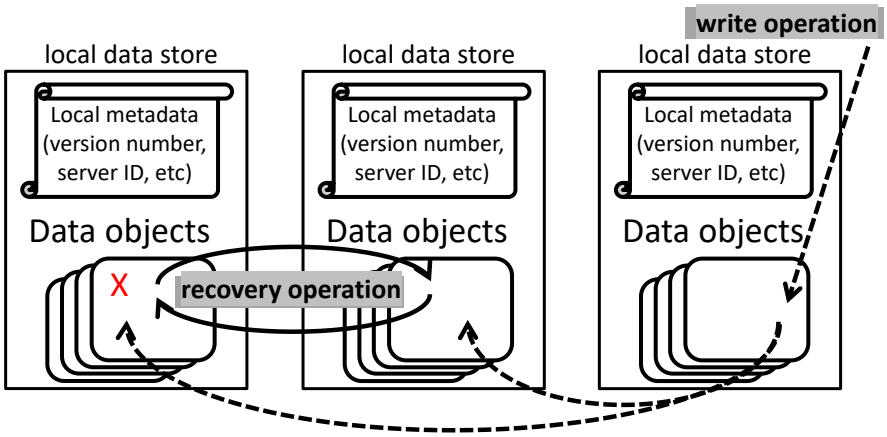
Finally, because of the increase in DRAM prices due to its scaling challenges and tight supply [2, 50, 62, 73], datacenter operators are migrating services from DRAM to flash [8, 48]. This means flash will be responsible for many more workloads, further exacerbating its endurance problem.

4.3 DIRECT Design

DIRECT is a set of techniques that enables a distributed storage system to maintain high availability and correctness in the face of high UBER. We define a distributed storage system as a set of many local stores coupled with a distributed protocol layer that replicates data and coordinates between the local stores. Figure 4.2a shows an ideal storage stack that runs on unreliable flash (flash that exposes high UBERs). Note that there is existing work on how to make local file systems tolerate corruption errors (we survey some of these systems in § 4.6), so our efforts in this paper focus on hardening the application-level storage system, which is the layer above the file system in the storage stack.



(a)



(b)

Figure 4.2: (a) DIRECT instruments cooperation between the local data stores and the distributed coordination layer to fix errors in the local data store. (b) Within the local data store, bit errors can affect either data objects or metadata. There must be precise semantics that define how recovery operations fixing data objects interact with write operations.

4.3.1 High Availability

Within the local data store, bit errors affect either application data or application metadata, as shown in Figure 4.2b. Maintaining multiple copies of each piece of data is the easiest way for a system to recover from bit errors. Our observation is that this redundancy already exists for application data.

Distributed Redundancy. Distributed storage systems typically use replication [35] or erasure coding [58, 98, 117] to store redundant copies of data. Hot data, which is stored

on flash storage, is typically replicated to avoid the higher bandwidth and CPU consumption associated with reconstructing erasure coded blocks [58]. In addition, erasure coding is not used for storage applications requiring fine-grained data access such as RocksDB. Since distributed storage systems assume storage devices correct device-level errors, they use replication primarily to correct entire server failures, not for correcting individual bit errors [52], even though this redundancy can significantly boost resilience to bit errors.

Consider the following example of a physically replicated storage system, such as HDFS. Suppose the minimum unit of recovery is a data block [3], which is replicated in each of the three data stores shown in Figure 4.2b. If the block has size B, and the uncorrectable bit error rate (UBER) is E, then the expected number of errors in the block will be $B \cdot E$. Since the block is replicated across R different servers, the storage application can recover the block from a remote server when an error occurs in at most $R - 1$ of its replicas. In this case, the only way that the storage system would encounter an application-observable read error is when at least one error exists in *each* of the copies of the block. Therefore, the probability of an application-level read error can be expressed as:

$$\mathbb{P}[error] = (1 - (1 - E)^B)^R \approx (E \cdot B)^R$$

where we assume $E \cdot B << 1$ and use a Taylor series approximation.

Then, for an UBER of $E = 10^{-15}$, a block size of $B = 128\ MB$ (typical of distributed file systems), and a replication factor of $R = 3$, the probability of error is 10^{-18} (files are measured in bytes, while UBER is in bits). This effectively is three orders of magnitude lower than the UBER of each local device.

However, with relatively large blocks, the probability of encountering at least one error in all block replicas quickly increases as UBER increases. For example, for an UBER of $E = 10^{-10}$, the expected number of errors in a single block will be $B \cdot E = 0.1$ for 128 MB

[3] Note that in HDFS while errors can be detected using checksums at a smaller granularity than the block size, actual recovery and replication is conducted at the granularity of a block.

	Probability of Application-Observable Error	
UBER	**Block Recovery**	**Chunk Recovery**
10^{-10}	$1 \cdot 10^{-3}$	$3 \cdot 10^{-10}$
10^{-15}	$1 \cdot 10^{-18}$	$1 \cdot 10^{-28}$

Table 4.1: Probability of application-observable error comparing block-by-block recovery to chunk-by-chunk recovery, with an UBER of 10^{-10}, and 10^{-15}. Finer granularity recovery provides significantly higher protection against corruptions.

blocks (Table 4.1). Then in this case $\mathbb{P}[error] \approx 0.001$. We make the observation that reducing $E \cdot B$, by reducing B, will dramatically reduce the probability of error.

Minimizing Error Amplification. DIRECT captures this intuition with error amplification (B in the previous example), or the number of bytes required to recover a bit error. DIRECT observes that *the lower the error amplification, the lower the probability of error and the faster recovery can occur*. This similarly implies a shorter period of time spent in degraded durability and thus higher availability.

In the example above, suppose the system can recover data at a finer granularity, for example, at chunk size $C = 64$ KB. Then a read error would occur if all three replicas of the same *chunk* have at least one bit error. The revised probability of read error is:

$$\mathbb{P}[error] = 1 - (1 - (1 - (1 - E)^C)^R)^{\frac{B}{C}}$$

Assuming $E \cdot C << 1$, Taylor series approximation leads to $(1 - (1 - E)^C)^R) \approx (E \cdot C)^R$, and assuming this value is much smaller than $\frac{B}{C}$, the probability of an application-observable error when correcting chunk-by-chunk is:

$$\mathbb{P}[error] \approx (E \cdot C)^R \cdot \frac{B}{C}$$

When $C = 64$ KB and $E = 10^{-10}$, this probability is $3 \cdot 10^{-10}$, which is much lower than the probability when recovering at the block level (see Table 4.1).

We can correctly recover even when all chunks have a bit error using bit-by-bit majority

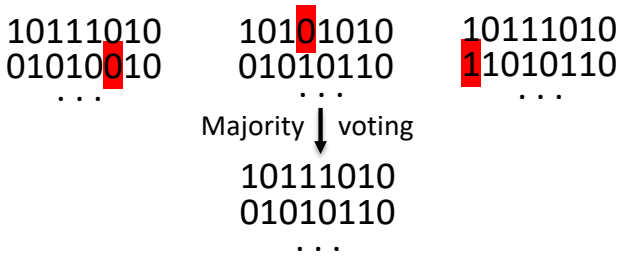


Figure 4.3: Even if the same chunk is corrupted on all replicas, bit-by-bit majority voting can reconstruct the correct chunk, by taking the majority of each bit across all chunks.

voting across all replicas: i.e., the recovered value of a bit in the chunk is the majority vote across the three chunk replicas (Figure 4.3). Bit-by-bit majority further reduces the application-observable error beyond chunk-by-chunk recovery, because the only way an application-observable error would occur is if an error occurs in the same bit across two chunks or more.

In a physically-replicated system like HDFS, minimizing error amplification is straightforward because corrupted blocks (and even bits) can be directly recovered from remote replicas. For a logically-replicated system like ZippyDB, however, blocks are not identical across replicas. This makes minimizing error amplification difficult, since DIRECT cannot simply recover from a remote physical chunk. For example, bit-by-bit majority voting is not possible in ZippyDB, because the replicas do not store the same physical bits. For such systems, DIRECT must instead first isolate the region where the error might have occurred and then retrieve objects one-by-one from the other servers (see § 4.4.2).

Metadata Error Amplification. So far, we have discussed the effect of errors on data blocks. However, error amplification can be even more severe if the error occurs in local metadata. For example, a corrupt lookup table for a distributed file system or a corrupt index in a key-value store can prevent a data store from starting up, which can mean re-replication of hundreds of GBs of data. Even though the likelihood of errors in metadata is statistically lower than in data blocks (metadata typically takes up less space than data), it requires stronger local protection to minimize error amplification. To address this problem, DIRECT either locally duplicates metadata or applies local software error correction.

Maintaining Correctness. Minimizing error amplification of data blocks and correcting data from remote replicas enables performant, live recovery of corrupted data blocks. However, DIRECT must also ensure recovery operations preserve the correctness of the distributed storage system, which might be dealing with concurrent write and read operations.

This is relatively straightforward in systems that do not support versioning or updates, such as HDFS, since if an object is recovered from a remote replica it is up-to-date. Systems like RocksDB which support versioning are more challenging, however, because if the system re-writes an object from a remote replica, it might overwrite a newer version with a stale version. In particular, the versions of the corrupted key-value pairs *are not known*, because (a) the corruption prevents the data from being read, yet (b) due to logical replication, the data's location does not provide information on its version. Hence to correctly recover corrupted key-value pairs, the system must locate some consistent (up-to-date) version of each pair. To do this, DIRECT forces recovery operations to go through a fault-tolerant log (for ZippyDB we use its existing Paxos log), which can provide correct ordering (§ 4.4.2). After a recovery operation, the corrupted data block should be fixed and "correct" with respect to consistency guarantees of the system.

4.3.2 DIRECT Techniques

To summarize, DIRECT includes the following techniques.

1. Systems must reduce error amplification of data objects and fix corruptions from remote replicas.
2. Systems must perform local metadata duplication to avoid high recovery costs from metadata corruption.
3. Systems must ensure safe recovery semantics.

Note that the first and second techniques apply exclusively to the local data store and affect *performance*, while the third technique may require that the local data store interact with the distributed coordination layer to ensure *correctness* during recovery.

4.4 Implementing DIRECT

To demonstrate the use of the DIRECT approach, we integrate it into two systems: HDFS, a popular distributed file system that represents physically-replicated systems, and ZippyDB, a distributed key-value store backed by RocksDB, which represents logically-replicated and versioned systems.

The techniques used to implement DIRECT in HDFS can be applied to other physically replicated systems, such as GFS [53], Windows Azure Storage [39], and RAMCloud [91], which write objects into large immutable blocks that are replicated across several servers. A centralized server contains cluster information and maps the block to the server that stores it. Similarly, the techniques used to implement DIRECT in ZippyDB and RocksDB can be applied to other logically replicated systems, such as Cassandra [111], MongoDB [41], CockroachDB [1], and HA PostgreSQL [10]. In these systems a distributed coordination layer manages the replication of objects across different servers and uses versioning to execute transactions.

4.4.1 HDFS-DIRECT

HDFS Overview.

HDFS is a distributed file system that is designed for storing large files that are sequentially written and read. Files are divided into 128MB blocks, and HDFS replicates and reads at the block level.

There are three types of HDFS servers: NameNode, JournalNode, and DataNode. The NameNode and JournalNodes store cluster metadata such as the cluster directory structure and mappings from block to DataNode. Together, the JournalNode and NameNode run a protocol similar to MultiPaxos, because there is no need for leader election– the leader node is the NameNode, and HA HDFS deployments run a ZooKeeper service that ensure there is always one live NameNode [4]. They store two types of files: `editLog`s, which

are update logs, and `fsImage`s, which are periodic snapshots that prevent the logs from growing indefinitely; most Paxos-like protocols implement snapshots [69, 90].

The `fsImage` is duplicated at the standby NameNode, which is part of the "high-availability" (HA) HDFS deployment and acts as a hot backup, downloading `fsImage`s periodically from the active NameNode [4]. By default, HDFS computes a single md5 hash across an entire `fsImage` file, which can be on the order of 10s of gigabytes for large enough clusters. To prevent both the case where both `fsImage` copies have a corrupted md5 hash, we calculate and store a CRC32 checksum for every 512 byte chunk of the `fsImage` (just like with data blocks). Then when loading the `fsImage`, the NameNode will fetch the corresponding chunk from the standby if it encounters a checksum error.

The `editLog`s are replicated across the JournalNodes, which implement the log replication part of a Paxos protocol. We recover the `editLog`s with a technique borrowed from PAR [24], which enables Paxos protocols to recover from corruptions. In particular, we write a special record with every update to indicate that the update was written to disk; this record enables the recovery process to distinguish machine crashes from corruptions (for more details see the PAR paper [24]). Finally, each edit log entry is individually checksummed, which means we can fix editLogs at the entry granularity.

In the steady-state, all cluster metadata resides in memory at the NameNode; corruptions on the NameNode or JournalNode do not affect steady-state performance and only affect correctness during the recovery/startup process.

DataNodes (the local data stores in Figure 4.2) store HDFS data blocks, and they respond to client requests to read blocks. Our efforts are focused on making sure DataNodes can efficiently correct corruption errors on the read path, because this is how corruptions actively affect steady-state performance. If a client encounters errors while reading a block, it will continue trying other DataNodes from the offset of the error until it can read the entire block. Once it encounters an error on a DataNode, the client will not try that node again. If there are no more DataNodes and the block is not fully read, the read fails and

that block is considered missing.

Additionally, HDFS has a configurable background "block scanner" that periodically scans data blocks and reports corrupted blocks for re-replication. But the default scan interval is three weeks, and even if the periodic scan does catch bit errors before the next read of a block, the NameNode can only recover at the 128 MB block granularity. If there is a bit error in every replica of a block, then HDFS cannot recover the block.

Implementing DIRECT

Reducing Error Amplification of Data Blocks We leverage the observation that HDFS checksums every 512 bytes in each 128 MB data block. Corruptions thus can be narrowed down to a 512 byte chunk; verifying checksums adds no overhead, because by default HDFS will verify checksums during every block read. For streaming performance, the smallest-size buffer that is streamed during a data block read is 64 KB, so we actually repair 64 KB everytime there is a corruption. To mask corruption errors from clients, we repair a data block synchronously during a read. Under DIRECT, the full read (and recovery) protocol is the following.

Each 128 MB block in HDFS is replicated on three DataNodes, call them A, B, C. An HDFS read of a 128 MB block is routed to one of these DataNodes, say A. A will stream the block to the client in 64 KB chunks, verifying checksums before it sends a chunk. If there is a checksum error in a 64 KB chunk, then A will attempt to repair the chunk by requesting the 64 KB chunk from B. If the chunk sent by B also contains a corruption, then the checksum will be incorrect, and A will request the chunk from C.

If C *also* sends a corrupted chunk, then A will attempt to construct a correct version of the chunk through bit-by-bit majority voting: the value of the bit in the chunk is the majority vote across the three versions provided by A, B, and C. The idea behind majority voting is that the probability that the corruptions on A, B, and C affect the same byte is very low, which means a majority vote across the three versions of the byte should end up with the correct data. After reconstructing the chunk via majority voting (Figure 4.3),

A will verify the checksums again; if the checksums fail, then the read fails. Majority voting allows HDFS-DIRECT to tolerate on the order of $10^4 - 10^5$ times more bit errors than HDFS. In fact, as we show in § 4.5.1, UBERs can be as high as 10^{-5} before majority voting failures are detectable in our experimental framework

Note that bit-by-bit majority voting is possible only if the device can return pages with uncorrectable errors (see § 4.6); otherwise, our HDFS implementation simply uses chunk-by-chunk recovery. Furthermore, for majority voting to add significant recovery power over chunk-by-chunk recovery, the number of corrupt bits returned by the device should be relatively small compared to the page size; the number of corrupt bits on a device page after running hardware ECC is dependent on the ECC function and its implementation.

Safe Recovery Semantics. Safety is straightforward in HDFS because data blocks are immutable once written, so there are never in-place updates or versions that will conflict with chunk recovery. Before a client does a block read, it first contacts the NameNode to get the DataNode IDs of all the DataNodes on which the block is replicated. When a client sends a block read request to a DataNode, it also sends this set of IDs. Because blocks are immutable and do not contain versions, these IDs are guaranteed to be correct replicas of the block, *if they exist*. It could be that a concurrent operation has deleted the block. In this case, if chunk recovery cannot find the block on another DataNode because it has been deleted, then it cannot perform recovery, so it will return the original checksum error to the client. This is correct, because there is no guarantee in HDFS that concurrent read operations should see the instantaneous deletion of a block.

Local Metadata Duplication. Each server in HDFS has local metadata files that must be correct, otherwise it cannot start. These files include a VERSION file, as well as special files on the NameNode and JournalNode. Metadata files are not protected in HDFS, thus a single corruption will prevent the server from starting. DIRECT adds a standard CRC32 checksum at the beginning of each file and replicates the file twice so that there are three copies of the file on disk. If there is a checksum error when the file is read, the recovery

protocol will check the copies to find one with a correct checksum.

4.4.2 ZippyDB-DIRECT

ZippyDB Overview

We also implemented DIRECT on a logically replicated system, ZippyDB, a distributed key-value store used within Facebook that is backed by RocksDB (i.e., RocksDB is the local data store in Figure 4.2a), which is a versioned key-value store.

ZippyDB runs on tens of thousands of flash provisioned servers at Facebook, which makes it an ideal target for DIRECT. ZippyDB provides a replication layer on top of RocksDB. ZippyDB is logically separated into shards, and each shard is fully replicated at least three ways. Each shard has a primary replica as well as a number of secondary replicas, wherein each replica is backed by a separate RocksDB instance residing on some server. Each ZippyDB server contains hundreds of shards, including both primary and secondary replicas. Hence, each ZippyDB server actually contains a large number of separate RocksDB instances.

ZippyDB runs a Paxos-based protocol for shard operations to ensure consistency. The primary shard acts as the leader for the Paxos entry, and each shard also has a Paxos log to persist each Paxos entry. Writes are considered durable when they are committed by a quorum of shards, and write operations are applied to the local RocksDB store in the order that they are committed. A separate service is responsible for monitoring the primary and triggering Paxos role changes.

RocksDB Overview

RocksDB is a local key-value store based on a log-structured merge (LSM) tree [93]. RocksDB batches writes in-memory—each write receives a sequence number that enables key versioning—and flushes them into immutable files of sorted key-value pairs called sorted string table (SST) files. SST files are composed of individually checksummed blocks, each of which can be a data block or a metadata block. The metadata blocks include

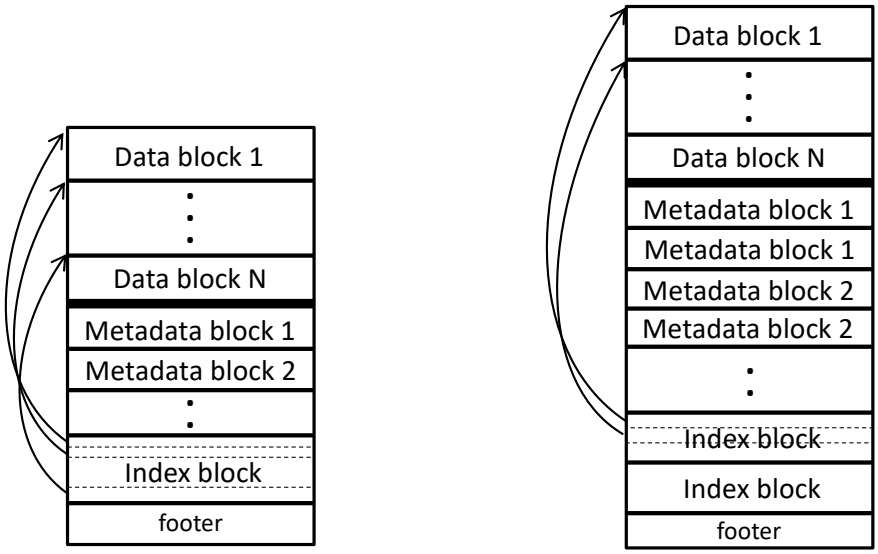


Figure 4.4: RocksDB SST file format. Index blocks point to keys within data blocks. Therefore, consecutive index blocks form a key range containing all keys in the sandwiched data block. DIRECT writes each metadata block at least twice in-line or uses an error correction code.

index blocks that point to the keys at the start of each data block (see Figure 4.4) [14].

SST files are organized into levels. A key feature of RocksDB and other LSM tree-backed stores is background compaction, which periodically scans SST files and compacts them into lower levels, as well as performs garbage collection on deleted and updated keys.

Implementing DIRECT

In ZippyDB, if a compaction encounters a corruption, an entire server, which typically has hundreds of gigabytes to terabytes of data, will shutdown and attempt to drain its RocksDB shards to another machine. Meanwhile, this sudden crash causes spikes in error rates and increases the load on other replicas while the server is recovering. To make matters worse, the new server could reside in a separate region, further delaying time to recovery. This leads to unnecessarily high error amplification: a single bit error can cause the migration of terabytes of data.

Reducing Error Amplification of Data Blocks. We observe that checksums in RocksDB are applied at the data block level, so a data block is the highest recovery granularity.

Data blocks are lists of key-value pairs, and key-value pairs are replicated at the ZippyDB layer. So if the metadata on an SST file is correct (see below how we protect metadata), a corrupted data block can be recovered by fetching the pairs in the data block from another replica. However, this is challenging for two reasons.

First, compactions are non-deterministic in RocksDB and depend on a variety of factors, such as available disk space and how compaction threads are scheduled. Hence, *two replicas of the same RocksDB instance will have different SST files*, making it impossible to find an exact replica of the corrupted SST file and the corrupted data block. Constrast this with HDFS where it is straightforward to find a replica of a corrupted block. Second, because the block is corrupted, it is impossible to know the exact key-value pairs that were stored in that block. Therefore, not only do we not know what data to look for on the other replica, we also don't know where to find it.

Instead of repairing the exact keys that are lost, we re-write a larger key range that covers the keys in the corrupted block. The key range is determined from index blocks, which are a type of metadata block that exist at the end of SST files and record a key in the range between consecutive data blocks, as shown in Figure 4.4. Hence, consecutive index block entries form a key range which is guaranteed to contain the lost keys.

Unfortunately, just knowing the key range is not enough: the existence of key versions in RocksDB and quorum replication in ZippyDB compounds the problem. In particular, a key must be recovered to a version greater than or equal to the lost key version, which could mean deleting it as key versions in RocksDB can create deletion markers. Additionally, if we naïvely fetch key versions from another replica, we may violate consistency.

Safe Recovery Semantics. To guide our recovery design, we introduce the following correctness requirement. Suppose we learn from the index blocks that we must re-replicate key range $[a, b]$. This key range is requested from another replica, which assembles a set of fresh key-value pairs in $[a, b]$, which we call a patch.

Safety Requirement: *Immediately after patch insertion, the database must be in a*

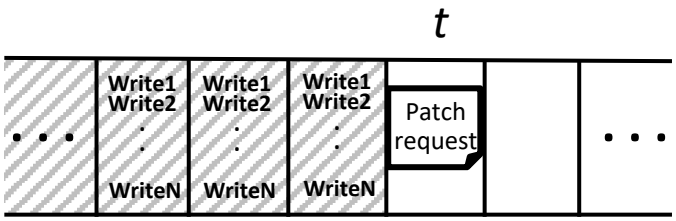


Figure 4.5: To serialize a patch properly, we add it as a request in the Paxos log. If the patch request is serialized at point t, then it must reflect all entries $t' < t$ (shaded). Furthermore, the patch request is not batched with any writes to ensure atomicity.

state that reflects some prefix of the Paxos log. Furthermore, this prefix must include the Paxos entries that originally updated the corrupted data block.

In other words, patch insertion must bring ZippyDB to some consistent state *after* the versions of the corrupted keys; otherwise, if the patch inserts prior versions of the keys, then the database will appear to go backwards.

Because the Paxos log serializes updates to ZippyDB, the cleanest way to find a prefix to recover up to is to serialize the patch insertion via the Paxos log. Then if patch insertion gets serialized as entry t in the log, the log prefix the patch must reflect is all Paxos entries $t' < t$, as shown in Figure 4.5. Serializing a patch at index t tells us exactly how to populate the patch. In particular, each key in the patch must be recovered to the largest $s < t$ such that s is the index of a Paxos entry that updates that key.

Furthermore, patch insertion must be atomic. Otherwise, it could be interleaved with updates to keys in the patch, which would violate the safety requirement, because then the version of the key in the patch would not reflect a prefix of t. This is actually a subtle point because ZippyDB batches many writes into a single Paxos entry, as shown in Figure 4.5. If patch insertion is batched with other writes, then the patch will not reflect the writes that are in front of it in the batch. Hence, we force the patch insertion to be its own Paxos entry.

Local Metadata Duplication. There are two flavors of metadata in RocksDB: metadata files and metadata blocks in SST files. Metadata files are only read during startup and then cached in memory. We can easily protect them with local replication, which adds a

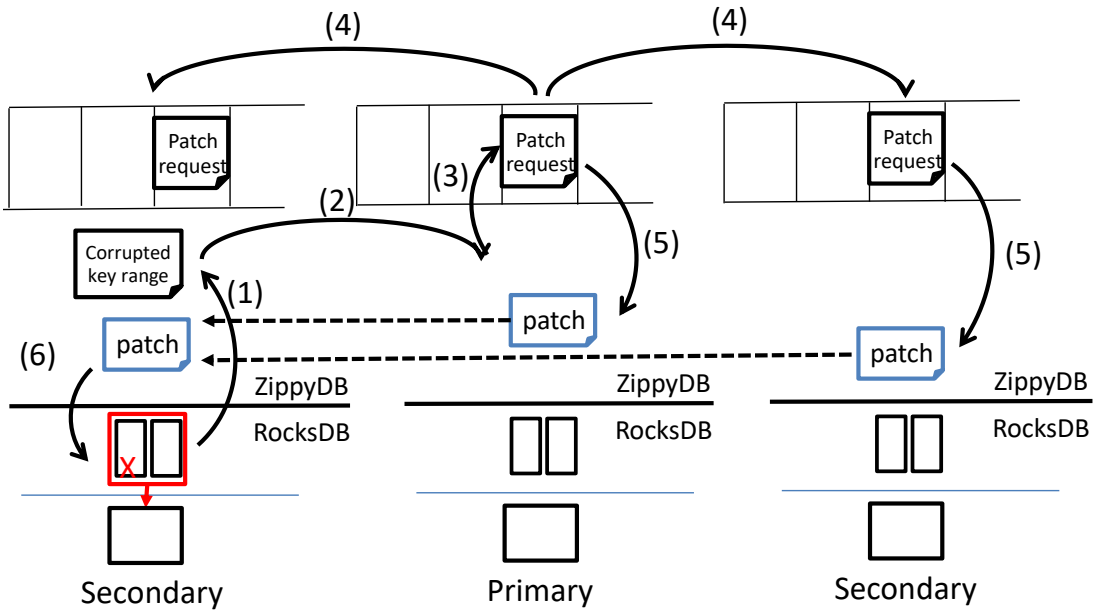


Figure 4.6: The process of recovering a corrupted RocksDB data block: (1) RocksDB compaction iterator determines the corrupted key range based on the index blocks of the SST files and reports it to ZippyDB. (2) ZippyDB reports this error to the primary of that replica. (3) Primary shard adds patch request to Paxos log. (4) Paxos engine replicates the request to all replicas. (5) Each replica tries to process the patch request. If the processing shard is *not* the corrupted shard, then it prepares a patch from its local RocksDB state and sends it to the corrupted shard. If the processing shard *is* the corrupted shard, then it waits for a patch from any replica. (6) Corrupted shard applies the fresh patch to its local RocksDB store.

minimal space overhead (on the order of kilobytes per server). We protect metadata blocks by writing them several times in-line in the same SST file. In our implementation, we write each metadata block twice[4]. Protecting metadata enables us to isolate errors to a single data block, rather than invalidating an entire SST file.

As with the HDFS JournalNode, we can protect against errors in the ZippyDB Paxos log with an additional entry [24].

DIRECT Recovery in ZippyDB

ZippyDB-DIRECT triggers a recovery procedure when RocksDB encounters a corruption error during compaction. Hence ZippyDB does not synchronously recover corrupted blocks encountered during user reads, unlike in HDFS. Instead, it returns the error to the

[4]For increased protection, metadata blocks can be locally replicated more than twice or protected with software error correction.

client, which will retry on a different replica, and ZippyDB will then trigger a manual compaction involving the file of the corrupted data block.

Figure 4.6 depicts this process. Importantly, we do not release a compaction's output files until the recovery procedure finishes; otherwise, stale key versions may reappear in the key ranges still undergoing recovery. Fortunately, because compaction is a background process, we can wait for recovery without affecting client operations.

Step (1) is implemented entirely within RocksDB. A RocksDB compaction iterator will record a corrupted key range when it's encountered, and then skip it to continue scanning. At the end of the iterator's lifetime, ZippyDB is notified about the corrupted key range. Multiple corrupt key ranges are batched into a single patch request.

In step (2), the patch is reported to the primary. Step (3) must go through the primary because the primary is the only shard that can propose entries to the Paxos log. Note this does not mean primaries cannot recover from corrupted data blocks. The patch request in the Paxos log is simply a no-op that reserves a point of reference for the recovery procedure and includes information necessary for recovery, such as the corrupted key ranges and the ID of the corrupted shard. Any replica that encounters the patch request in the log is by definition up-to-date to that point in the Paxos log, which means any replica that isn't the corrupted replica can send a patch to the corrupted replica.

In step (4), ZippyDB waits for the Paxos log to replicate the Paxos entry as well as for other replicas to consume the log until they encounter the patch request.

In step (5), an uncorrupted replica assembles a patch on the specified key range with a RocksDB iterator. Note that this replica might encounter a bit corruption while assembling the patch. In practice the probability of this is very small because the number of keys covered by the patch is on the order of kilobytes (§ 4.5.2). However, if a replica does encounter a corruption while assembling a patch, it simply does not send a patch. Therefore, for the patch request to fail, *both* (or more, if the replication factor is more than 3) uncorrupted replicas will have to encounter a bit corruption, and this probability is low (see Table 4.1).

Step (6) is also implemented at the RocksDB level. When a replica applies a patch, simply inserting all the key-value pairs present in the patch is insufficient because of deleted keys. In particular, any key present in the requested key range and *not* present in the patch is an implicit delete. Therefore, to apply a patch, the corrupted shard must also delete any keys that it can see that aren't present in the patch. This case is possible because RocksDB deletes keys by inserting a tombstone value inline in SST files. Hence the corrupted data block may contain tombstone operators that delete a key, and these must be preserved.

Invalidating Snapshots

In RocksDB, users can request snapshots, which are represented by a sequence number. Then, for as long as the snapshot with sequence number s is active, RocksDB will not delete any version, s', of a key where s' is the greatest version of the key such that $s' < s$. ZippyDB uses RocksDB snapshots to execute transactions. If RocksDB invalidates a snapshot, then the transaction using that snapshot will abort and retry.

A subtle side-effect of a corrupted data block is snapshot corruption. For example, suppose the RocksDB store has a snapshot at sequence number 100 and the corrupted data block contains a key with sequence number 90. For safety, we need to invalidate any snapshots that could have been affected by the corrupted key range. Because the data block is corrupted, it cannot be read, so we do not know whether the corruption affects snapshot 100. For now, we take the obviously correct approach and invalidate all local snapshots of the RocksDB shard affected by the corruption. In practice, this is reasonable because most RocksDB snapshots have short lifetimes.

4.5 Evaluation

This section evaluates how successfully DIRECT maintains system availability in the face of high error rates by answering the following questions:

1. By how much does DIRECT decrease application-level errors in both HDFS and ZippyDB? In HDFS, how far can DIRECT drive UBER while avoiding application-

level errors?

2. How much does DIRECT decrease time to recovery from compaction corruption errors in ZippyDB?

Note we do not measure recovery time in HDFS because DIRECT handles bit errors *synchronously*, which means read errors only propagate to the application-level if DIRECT cannot fix them. On the other hand, in ZippyDB, DIRECT handles bit errors asynchronously because recovery procedures must go through the coordination layer, as described in § 4.4.2.

Experimental Setup. To evaluate ZippyDB, we set up a cluster of 60 Facebook servers that capture and duplicate live traffic from a heavily loaded service used in computing user feeds. To evaluate HDFS, we run experiments on a cluster of 10 machines each with 8 ARMv8 cores at 2.4 GHz, 96 GB of RAM, and 120 GB of flash. In the cluster, we allocate one machine each for a NameNode, standby NameNode, and JournalNode, and three machines run the DataNode role. Four machines act as HDFS clients. HDFS experiments have a load and read phase: in the load phase, we load the cluster with 500, 128MB files with random data. In the read phase, clients randomly select files to read. After the load phase, we clear the page cache.

Error Injection. To simulate UBERs, we inject bit errors into the files of both systems. In ZippyDB, we inject errors with a custom RocksDB environment that flips bits as they are read from a file. In HDFS, we run a script in between the load and read phases that flips bits in on-disk files and flushes them. For an UBER of μ, e.g. $\mu = 10^{-11}$, we inject errors at the rate of 1 bit flip per $1/\mu$ bits read. We tested with UBERs higher than the manufacturer advertised 10^{-15} to test the system's performance under high error rates, and so that we can measure enough bit errors during an experiment time of 12 hours rather than several days (or years)[5].

[5] Note that an UBER 10^{-11} is $10,000\times$ *higher* than 10^{-15}.

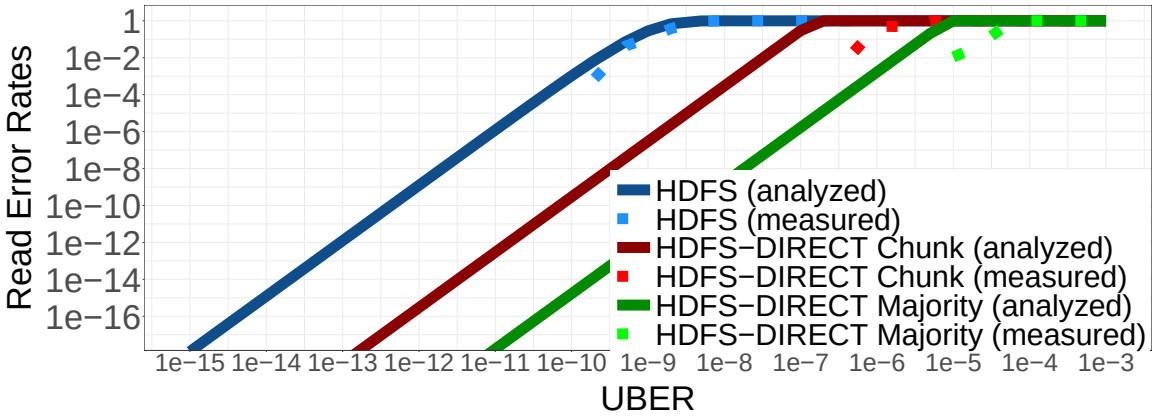


Figure 4.7: Read error rate for HDFS with varying UBER. HDFS-DIRECT Chunk is based on chunk-by-chunk recovery, while HDFS-DIRECT Majority is computed on bit-by-bit majority. The analyzed data is computed using the formulas in § 4.3.1.

Baselines. We compare against unmodified HDFS and ZippyDB, both systems used in production for many years. Although unmodified HDFS does compute checksums for chunks, it does not recover at that granularity. HDFS-DIRECT leverages these checksums during recovery, which allows it to recover blocks synchronously within client reads. In unmodified ZippyDB, when a RocksDB instance encounters a compaction error, the entire ZippyDB server crashes so that the system can reuse recovery logic for recovery from failed nodes. While this may seem like an overly aggressive baseline, it makes sense for a production system like ZippyDB, in which bit errors are assumed to be rare. On the other hand, DIRECT addresses a scenario where bit errors are common, which makes such a solution inappropriate.

4.5.1 HDFS

UBER Tolerance. The main advantage of HDFS-DIRECT over HDFS is the ability to tolerate much higher UBERs with chunk-level recovery and majority voting. Figure 4.7 reports block read error rates of HDFS with varying UBERs. In HDFS, read errors are also considered *data loss*, because the data is unreadable (and hence unrecoverable) even after trying all 3 replicas. The figure shows the measured read error on our HDFS experimental setup, within the UBER range in which we could effectively measure errors, as well as the

computed read error based on the computation presented in § 4.3.1. We compared unmodified HDFS, with chunk-by-chunk recovery and bit-by-bit majority. The experimental read error is collected by running thousands of file reads and measuring how many fail. The measured results are relatively close to the analytical results, and in fact experience even fewer errors than the analytical model. We believe the primary reason is that the Taylor's approximation used in the analytical model does not hold for high UBERs. As expected, bit-by-bit majority (green lines) reduces the read error rate due to its lower error amplification (it can recover bit-by-bit). Both our analysis and the experimental results show that HDFS-DIRECT can tolerate a $10,000\times$–$100,000\times$ higher UBER and maintain the same read error rate.

Overhead of DIRECT. Because DIRECT corrects bit errors synchronously in HDFS, error correction poses an overhead on reads that encounter bit errors.

UBER	HDFS throughput [GB/s]	HDFS-DIRECT throughput [GB/s]
10^{-7}	0.00 ± 0.00	2.09 ± 0.08
10^{-8}	0.00 ± 0.00	2.56 ± 0.09
10^{-9}	2.46 ± 0.08	2.55 ± 0.07
10^{-10}	2.89 ± 0.10	2.84 ± 0.07
No errors	2.83 ± 0.07	2.88 ± 0.07

Table 4.2: Throughput of HDFS and HDFS-DIRECT. At UBER$= 10^{-8}$, HDFS throughput collapses due to bit errors.

Table 4.2 shows the throughput of both systems, measured by saturating the DataNodes with four, 64-threaded clients that are continuously reading random files. The throughput of HDFS goes to zero at an UBER of 10^{-8}, because it cannot complete any reads due to corruption errors. Such failures do not occur in HDFS-DIRECT, although its throughput decreases modestly due to the overhead of synchronously repairing corrupt chunks during read operations.

For HDFS-DIRECT, we are also interested in latency incurred by synchronous chunk recovery. We compare the CDF of read latencies of 128 MB blocks for different UBERs in

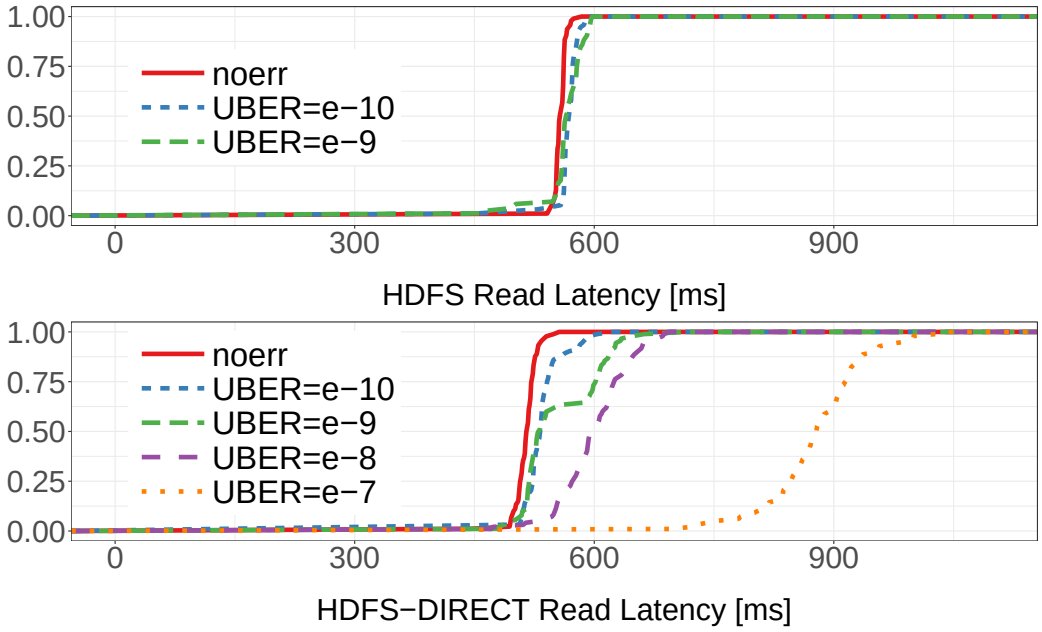


Figure 4.8: Read latencies (128 MB) of HDFS and HDFS-DIRECT. All reads fail in HDFS an UBER of 10^{-8} and higher.

Figure 4.8. The higher the UBER, the more chunk recovery requests that need to be made during a block read and the longer these requests will take. The results in Figure 4.8 (and Table 4.2) highlight the fine-grained tradeoff between performance and recoverability that is exposed by DIRECT. We also report HDFS read latencies, but there is little difference across UBERs because only latency for successful block reads are included; again, we do not report results for UBERs higher than 10^{-8}, since at those error rates HDFS cannot successfully read any blocks.

Interestingly, these overheads become minimal when we run an end-to-end benchmark. We ran the TeraSort benchmark, a canonical Hadoop benchmark. We configured TeraSort to generate and sort 20 GB of data. Table 4.3 shows the time it takes HDFS-DIRECT to complete the TeraSort benchmark. Note that at an UBER of 10^{-8}, the time it takes to complete the benchmark is similar to when there are no errors (in fact, we do no report results for UBERs lower than 10^{-8} because they are so similar to results when there are no errors). Even at an UBER of 10^{-7}, the performance overhead is relatively low, because TeraSort is dominated by sort time in the mappers and reducers, rather than the time it

UBER	Time to Complete Benchmark (s)
10^{-7}	177.4 ± 2.5
10^{-8}	169.4 ± 2.1
No errors	166.2 ± 1.8

Table 4.3: Time in seconds for HDFS-DIRECT to complete the TeraSort benchmark.

takes to read the data into memory. These results suggest that even at very high UBERs, DIRECT imposes a low recovery overhead in workloads that are not disk-bound.

4.5.2 ZippyDB

UBER Tolerance. One main difference between unmodified ZippyDB and ZippyDB–DIRECT is that ZippyDB-DIRECT avoids crashing when encountering a bit error. To characterize how many server crashes are mitigated with DIRECT, we measured the average rate of compaction errors per hour *per server*, over 12 hours. The results are shown in Table 4.4. Figure 4.9 shows the read error rate over time of both systems, and Table 4.4 also shows the number of read errors as a percentage of all reads. Note that the error rate patterns across UBERs are different because they are run during different time intervals, so each UBER experiment sees different traffic. We did try to ensure read/write QPS and query distribution remain steady throughout the experiments. Unfortunately, there is no tracing system set up for ZippyDB, so we were unable to capture and replay traces.

The error rate is much higher for ZippyDB than ZippyDB-DIRECT because not only do clients see errors from regular read operations, but also they experience the spike in errors when a server shuts down due to a compaction corruption.

Time Spent in Reduced Durability. With DIRECT, we also seek to minimize the amount of time spent in reduced durability to decrease the likelihood of simultaneous replica failures. Figure 4.10 shows a CDF of the time it takes to recover from compaction errors in ZippyDB-DIRECT. The graph shows the amount of time it takes for replicas to process the Paxos log up until the patch request, as well as the overhead of constructing and inserting the patch. With DIRECT, this recovery time is on the order of *milliseconds*. In contrast,

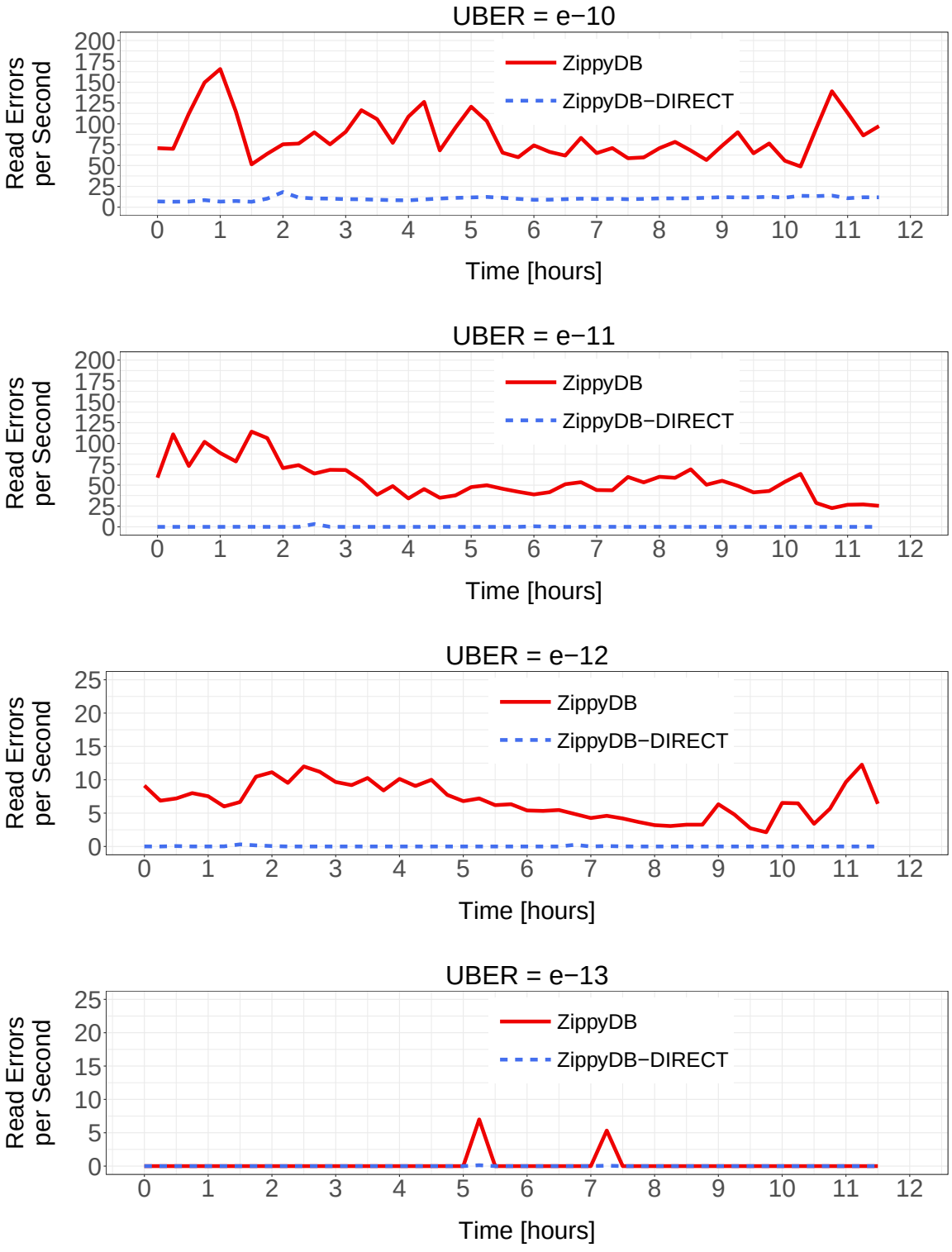


Figure 4.9: Read error rates over time in ZippyDB and ZippyDB-DIRECT, under varying UBERs.

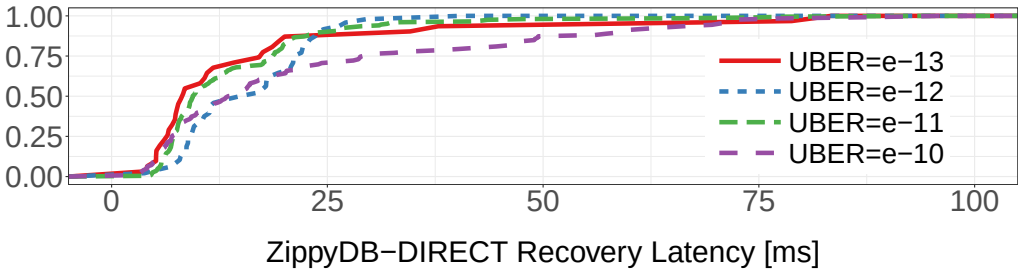


Figure 4.10: CDF of compaction recovery latencies in ZippyDB-DIRECT. ZippyDB-DIRECT takes milliseconds to recover from corruptions, while ZippyDB takes *minutes*.

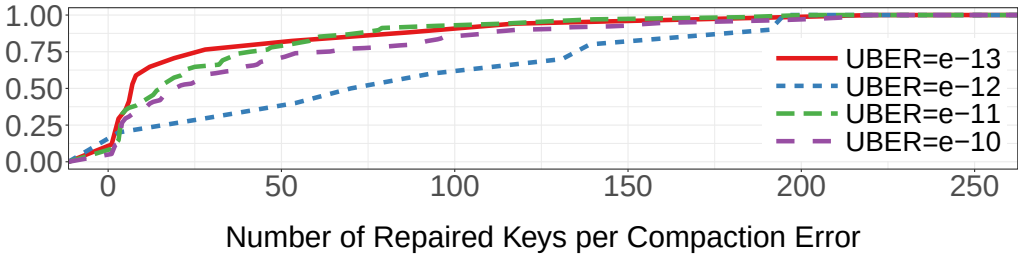


Figure 4.11: CDF of patch sizes generated during the ZippyDB-DIRECT recovery process. The patch size is small, which means low error amplification.

the period of reduced durability in unmodified ZippyDB due to a compaction error is on the order of *minutes*, depending on the amount of data stored in the crashed ZippyDB server. This is due to the high error amplification of ZippyDB, which invalidates 100s of RocksDB shards due to a single compaction bit error. With DIRECT, ZippyDB can reduce its recovery time due to a bit error by around $10,000\times$.

We also found that the recovery latency is dependent on the size of the patch required

	Read Errors		Compaction Errors per Hour per Server
UBER	ZippyDB	ZippyDB-DIRECT	ZippyDB
10^{-10}	2.7308%	0.1865%	0.1991 ± 0.1077
10^{-11}	1.9808%	0.0400%	0.0621 ± 0.0455
10^{-12}	0.2650%	0.0008%	0.0038 ± 0.0035
10^{-13}	0.0108%	0.0002%	0.0003 ± 0.0005

Table 4.4: Read and compaction errors with ZippyDB and ZippyDB-DIRECT. The read errors are a percentage of the total number of reads, and the compaction errors are the number of errors per hour per server. ZippyDB-DIRECT is able to fix all compaction errors in our experiment, while the server crashes in ZippyDB.

to correct the corrupted key range. Figure 4.11 presents a CDF of the size of the patches generated during the recovery process. Patch size is also interesting because the recovery mechanism described in § 4.4.2 recovers a *range* of keys, since the exact keys on the corrupted data block are impossible to identify. As we see in Figure 4.11, even though recovering a range can in theory increase error amplification, the number of keys required for recovery is still low. Figure 4.11 also confirms that as the UBER increases, patch sizes increase due to more key ranges getting corrupted during a single compaction operation.

4.6 Discussion

Local File System Error Tolerance. When devices exhibit higher UBERs, local file systems also experience higher UBERs. DIRECT protects application-level metadata and data, which are just data blocks at the local file system level. Protecting local file system metadata (such as inodes, the FS journal, etc.) is beyond the scope of this paper. Several existing file systems protect metadata against bit corruptions [3, 19, 20, 55, 72, 95, 119]. The general approach is to add checksums to file system metadata and locally replicate it for error correction. Another approach is to use more reliable hardware for metadata, and less reliable hardware for data blocks [72]. Alternatively, instead of directly replicating metadata, another approach is to harden local nodes using lightweight versioning [46].

Support for DIRECT. DIRECT does not require any hardware support. However, a couple of simple device-level mechanisms would help datacenter operators run devices past their manufacturer defined UBER. First, it would be beneficial if devices have a less aggressive "bad block policy", which is a firmware protocol for retiring blocks once they reach some heuristic-defined level of errors. Second, it would be beneficial if devices return the content of pages, even if they have an error. This enables distributed storage applications to minimize recovery amplification by recovering data at a granularity smaller than a device page (e.g., using majority voting). This is not a hard requirement, since as we showed in § 4.3.1 even recovering at a device page level (e.g., 4-8 KB) provides significant benefits.

In case corrupt pages cannot be read, copies of local metadata must be stored on separate physical pages. Otherwise, a page error could invalidate all copies of the metadata.

4.7 Related Work

Related work is divided into two main parts: systems that deal with device errors using software mechanisms or by applying more aggressive hardware mechanisms.

We also note that our work departs from existing work on data integrity in data storage systems [29, 34, 68, 103] because we expose bit corruptions at the distributed layer, rather than containing them completely in the storage layer. Furthermore, a key motivating factor in DIRECT is that bit corruptions will be common-place, so DIRECT does not stop at identifying corruptions but introduces a principled and performant way of fixing them to achieve high availability.

Software-level Redundancy. DIRECT is related to PAR [24] and PASC [44], which demonstrate how consensus-based protocols can be adapted to address bit-level errors. Unlike both of these works, which only address consensus protocols, our work tackles bit-level errors in general purpose storage systems. We also show how increasing the resiliency to bit-level errors can significantly reduce storage costs and improve live recovery speed in datacenter environments.

Other related work use different approaches. HARDFS [46] hardens local HDFS nodes by augmenting each node with a lightweight version that verifies its behavior. HDFS-DIRECT generalizes HARDFS, by only applying local protection to metadata and leveraging distributed replicas to recover data. FlexECC [59] and Duracache [77] are flash-based key-value caches that use less reliable disks by treating devices errors as cache misses. D-GRAID is a RAID storage system that gracefully degrades by minimizing the amount of data needed to recover from bit corruptions [104]. AHEAD and EDB-Tree apply software-level error detection and correction to address DRAM corruption in databases [66, 67].

There is a large number of distributed storage systems that use inexpensive, unreliable

hardware, while providing consistency and reliability guarantees [27, 45, 53]. However, these systems treat bit corruptions similar to entire-node failures and suffer from high recovery amplification. DIRECT extends the idea of providing reliability via software on unreliable storage, and demonstrates how distributed storage systems can conduct live recovery of disk corruptions with minimal performance cost.

There is a large body of work on finding errors in the way both local file systems and distributed file systems handle disk corruptions [52]. These efforts are orthogonal to our work, because they focus on correctness flaws of software that corrects disk corruptions. Research on hardening local file systems to tolerate disk errors supports our vision of less reliable disks, because it shows it is possible to protect a local file system from disk bit errors [3, 19, 20, 55, 72, 95, 119].

Hardware-level Redundancy. Several studies explore extending SSD lifetime via more aggressive or adaptive hardware error correction. Tanakamuru *et al.* [107] propose adapting codeword size based on the device's wear level to improve SSD lifetime. Cai *et al.* [38] and Liu *et al.* [78] introduce techniques to dynamically learn and adjust the cell voltage levels based on retention age. Zhao *et al.* [121] propose using the soft information with LDPC error correction to increase lifetime. Our approach is different: instead of improving hardware-based error correction, we leverage existing software-based redundancy to address bit-level errors.

4.8 Conclusion and Future Work

This paper presents DIRECT, which shifts the responsibility of error correction from the hardware layer to the distributed application layer. In doing so, DIRECT is able to harness the inherent redundancy that exists in distributed storage applications to recover bit corruptions *in a live system*.

We can extend the approach of handling error correction in the distributed storage layer in several directions. First, distributed storage systems can control the level of error cor-

rection depending on data type. For example, some data types may be more sensitive to bit corruptions (e.g., critical metadata), while others may not. Second, distributed storage system can control hardware mechanisms that influence the performance of the device. For example, storing fewer bits per cell generally reduces the latency of the device (at the expense of its capacity). Certain applications may prefer for to use a hybrid of low latency and low capacity devices for hot data, while reserving the high capacity devices for colder data.

Chapter 5

Conclusion and Future Vision

Because the rate of data creation is growing exponentially, it is now critical to reduce the storage costs of distributed storage systems. This is particularly important in the datacenter, where the scale of data being stored is large enough where storage cost reductions are worth the software engineering overhead that they introduce.

This thesis has showed two ways that we can use existing redundancy in distributed storage systems to reduce storage costs throughout the storage stack. First, Replex shows how replication can be repurposed to provide secondary indices, eliminating the need for indexing to take up additional storage space. Second, DIRECT shows how replication for availability can be repurposed to correct bit errors in flash devices. By correcting hardware errors at the application layer, we can greatly extend the lifetime of flash devices. In the datacenter, where there can be thousands to millions of devices deployed, such an lifetime increase would alleviate not only the monetary burdens associated with disk churn but also the operational overheads of swapping out expired devices.

Ultimately, we believe that fault tolerance in the distributed storage stack should be redesigned in an end-to-end manner. This vision begins with a storage medium that is allowed to expose some level of corruption errors to the software layer. Immediately on top of that, file systems would need to be redesigned to tolerate these bit errors. As we have

summarized in this thesis, there is already some work on file systems that can tolerate bit errors in their metadata. File systems would not have to protect their data because this will be done at the application layer, which can use techniques such as DIRECT or Replex to provide enough redundancy to correct bit errors.

With both of these techniques, datacenters can reduce the storage costs of distributed storage systems and begin to consider how to redesign the storage stack, thereby making persistent data more sustainable in the age of Big Data.

Bibliography

[1] CockroachDB docs. https://www.cockroachlabs.com/docs/stable/.

[2] DRAM prices continue to climb. https://epsnews.com/2017/08/18/dram-prices-continue-climb/.

[3] Ext4 metadata checksums. https://blogs.msdn.microsoft.com/b8/2012/01/16/building-the-next-generation-file-system-for-windows-refs/.

[4] HDFS high availiability. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html.

[5] High-efficiency SSD for reliable data storage systems. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2011/20110810_T2A_Yang.pdf.

[6] Introducing Lightning: A flexible NVMe JBOF. https://code.facebook.com/posts/989638804458007/introducing-lightning-a-flexible-nvme-jbof/.

[7] LevelDB. http://leveldb.org.

[8] McDipper: A key-value cache for flash storage. https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/.

[9] Novel 4k error correcting code for QLC NAND. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170809_FE22_Kuo.pdf.

[10] PostgreSQL. https://www.postgresql.org/.

[11] Project Voldemort: A distributed key-value storage system. http://www.project-voldemort.com/voldemort.

[12] Redis. https://redis.io.

[13] RocksDB. http://rocksdb.org.

[14] RocksDB block based table format. https://github.com/facebook/rocksdb/wiki/Rocksdb-BlockBasedTable-Format.

[15] SanDisk Datasheet. https://www.sandisk.com/business/datacenter/resources/data-sheets/fusion-iomemory-sx350_datasheet.

[16] Scaling the facebook data warehouse to 300 pb. https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/.

[17] Under the hood: Building and open-sourcing RocksDB. http://www.facebook.com/notes/facebook-engineering/under-the-hood-building-and-open-sourcing-rocksdb/10151822347683920.

[18] What is R.A.I.S.E? https://www.kingston.com/us/ssd/raise.

[19] XFS reliable detection and repair of metadata corruption. http://xfs.org/index.php/Reliable_Detection_and_Repair_of_Metadata_Corruption.

[20] The Z File System (ZFS). https://www.freebsd.org/doc/handbook/zfs.html.

[21] NAND flash media management through RAIN. Technical report, Micron, 2013.

[22] M. K. Aguilera, J. B. Leners, R. Kotla, and M. Walfish. Yesquel: scalable SQL storage for Web applications. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, 2015.

[23] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[24] R. Alagappan, A. Ganesan, E. Lee, A. Albarghouthi, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Protocol-aware recovery for consensus-based storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 15–32, Oakland, CA, 2018. USENIX Association.

[25] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and E. Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *USENIX Annual Technical Conference*, pages 91–102, 2013.

[26] P. Alcorn. Facebook asks for QLC NAND, Toshiba answers with 100TB QLC SSDs with TSV. `http://www.tomshardware.com/news/qlc-nand-ssd-toshiba-facebook,32451.html`.

[27] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM.

[28] J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: the definitive guide*. O'Reilly Media, Inc., 2010.

[29] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)*, 4(3):8, 2008.

[30] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. CORFU: A distributed shared log. *ACM Transactions on Computer Systems*, 31(4), 2013.

[31] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[32] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[33] P. A. Bernstein, C. W. Reid, and S. Das. Hyder-a transactional record manager for shared flash. In *Proceedings of the Conference on Innovative Datasystems Research*, 2011.

[34] N. Borisov, S. Babu, N. Mandagere, and S. Uttamchandani. Dealing proactively with data corruption: Challenges and opportunities. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 34–39. IEEE, 2011.

[35] D. Borthakur. HDFS block replica placement in your hands now! `http://hadoopblog.blogspot.com/2009/09/hdfs-block-replica-placement-in-your.html`.

[36] E. Brewer, L. Ying, L. Greenfield, R. Cypher, and T. T'so. Disks for data centers. Technical report, Google, 2016.

[37] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 521–526, Dresden, Germany, 2012.

[38] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, and O. Mutlu. Data retention in MLC NAND flash memory: Characterization, optimization, and recovery. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, pages 551–563, San Francisco, CA, 2015.

[39] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[40] A. Calil and R. dos Santos Mello. SimpleSQL: a relational layer for SimpleDB. In *Proceedings of Advances in Databases and Information Systems*. Springer, 2012.

[41] K. Chodorow. *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2013.

[42] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, 2010.

[43] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3), 2013.

[44] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 41–41, Berkeley, CA, USA, 2012. USENIX Association.

[45] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Operating Systems Review*, 41(6):205–220, Oct. 2007.

[46] T. Do, T. Harter, Y. Liu, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. HARDFS: Hardening HDFS with selective and lightweight versioning. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 105–118, San Jose, CA, 2013. USENIX.

[47] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti. Flashield: a key-value cache that minimizes writes to flash. *CoRR*, abs/1702.02588, 2017.

[48] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. M. Hazelwood, C. Petersen, A. Cidon, and S. Katti. Reducing DRAM footprint with NVM in facebook. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 42:1–42:13, 2018.

[49] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4), 2012.

[50] D. Exchange. DRAM supply to remain tight with its annual bit growth for 2018 forecast at just 19.6. `www.dramexchange.com`.

[51] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[52] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *15th USENIX Conference on File and Storage Technologies*, pages 149–166, Santa Clara, CA, 2017. USENIX Association.

[53] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[54] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, pages 17–24, San Jose, CA, 2012.

[55] H. S. Gunawi, V. Prabhakaran, S. Krishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving file system reliability with i/o shepherding. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 293–306, New York, NY, USA, 2007. ACM.

[56] A. Guttman. R-trees: A dynamic index structure for spatial searching. volume 14. ACM, 1984.

[57] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis of HDFS under HBase: A Facebook messages case study. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 199–212, Santa Clara, CA, 2014.

[58] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in Windows Azure Storage. In *Usenix annual technical conference*, pages 15–26. Boston, MA, 2012.

[59] P. Huang, P. Subedi, X. He, S. He, and K. Zhou. FlexECC: Partially relaxing ECC of MLC SSD for better cache performance. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 489–500, Philadelphia, PA, 2014.

[60] J. Jeong, S. S. Hahn, S. Lee, and J. Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *FAST*, pages 61–74, 2014.

[61] K. Kambatla and Y. Chen. The truth about MapReduce performance on SSDs. In *Proceedings of the 28th Large Installation System Administration Conference*, pages 118–126, Seattle, WA, 2014.

[62] U. Kang, H.-s. Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi. Co-architecting controllers and DRAM to enhance DRAM process scaling. In *The memory forum*, pages 1–4, 2014.

[63] A. Khetrapal and V. Ganesh. HBase and Hypertable for large scale distributed storage systems. *Department of Computer Science, Purdue University*, 2006.

[64] H. Kim, S.-J. Ahn, Y. G. Shin, K. Lee, and E. Jung. Evolution of NAND flash memory: From 2D to 3D as a storage market leader. In *Memory Workshop (IMW), 2017 IEEE International*, pages 1–4. IEEE, 2017.

[65] R. Klophaus. Riak core: building distributed applications without shared state. In *Proceedings of ACM SIGPLAN Commercial Users of Functional Programming*, 2010.

[66] T. Kolditz, D. Habich, W. Lehner, M. Werner, and S. T. de Bruijn. AHEAD: Adaptable data hardening for on-the-fly hardware error detection during database query processing. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1619–1634, New York, NY, USA, 2018. ACM.

[67] T. Kolditz, T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. Online bit flip detection for in-memory B-trees on unreliable hardware. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, pages 5:1–5:9, New York, NY, USA, 2014. ACM.

[68] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity lost and parity regained. In *FAST*, volume 2008, page 127, 2008.

[69] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[70] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, 2009.

[71] J. Lee, J. Jang, J. Lim, Y. G. Shin, K. Lee, and E. Jung. A new ruler on the storage market: 3D-NAND flash for high-density memory and its technology evolutions and challenges on the future. In *Electron Devices Meeting (IEDM), 2016 IEEE International*, pages 11–2. IEEE, 2016.

[72] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim. FlexFS: A flexible flash file system for MLC NAND flash memory. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 9–9, Berkeley, CA, USA, 2009. USENIX Association.

[73] S.-H. Lee. Technology scaling challenges and opportunities of memory devices. In *Electron Devices Meeting (IEDM), 2016 IEEE International*, pages 1–1. IEEE, 2016.

[74] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 501–512, 2014.

[75] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th Annual Middleware Conference*, pages 50–62, Vancouver, BC, 2015.

[76] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 1–13, New York, NY, USA, 2011.

[77] R. Liu, C. Yang, C. Li, and G. Chen. DuraCache: a durable SSD cache using MLC NAND flash. In *Proceedings of the 50th Annual Design Automation Conference 2013*, pages 166–171, Austin,TX, 2013.

[78] R.-S. Liu, C.-L. Yang, and W. Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, San Jose, CA, 2012.

[79] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*, volume 13, pages 313–328, 2013.

[80] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, Feb. 2016.

[81] M. Luby. Tornado codes: Practical erasure codes based on random irregular graphs. In *Randomization and Approximation Techniques in Computer Science*. Springer, 1998.

[82] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu. Improving 3D NAND flash memory lifetime by tolerating early retention loss and process variation. In *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '18, pages 106–106, New York, NY, USA, 2018. ACM.

[83] P. Mahajan, L. Alvisi, M. Dahlin, et al. Consistency, availability, and convergence.

[84] P. Maymounkov. Online codes. Technical report, New York University, 2002.

[85] C. Mellor. Toshiba flashes 100TB QLC flash drive, may go on sale within months. really. `http://www.theregister.co.uk/2016/08/10/toshiba_100tb_qlc_ssd//`.

[86] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 177–190, Portland, Oregon, 2015.

[87] R. Micheloni et al. *3D Flash memories*. Springer, 2016.

[88] T. D. Nguyen, L.-L. Yang, and L. Hanzo. Systematic luby transform codes and their soft decoding. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, 2007.

[89] S. Ohshima and Y. Tanaka. New 3D flash technologies offer both low cost and low power solutions. `https://www.flashmemorysummit.com/English/Conference/Keynotes.html`.

[90] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, 2014.

[91] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[92] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-defined flash for web-scale Internet storage systems. *SIGARCH Computing Architecture News*, 42(1):471–484, 2014.

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

[94] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, 1988.

[95] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[96] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2), 1960.

[97] M. Ronstrom and L. Thalmann. MySQL cluster architecture overview. *MySQL Technical White Paper*, 2004.

[98] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. In *Proceedings of the 39th International Conference on Very Large Data Bases*, pages 325–336, Trento, Italy, 2013.

[99] B. Schroeder, R. Lagisetty, and A. Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, pages 67–80, Santa Clara, CA, 2016.

[100] Scylla. ScyllaDB. `https://www.scylladb.com/`.

[101] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[102] A. Shokrollahi. Raptor codes. *IEEE Transaction on Information Theory*, 52(6), 2006.

[103] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring data integrity in storage: Techniques and applications. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*, pages 26–36. ACM, 2005.

[104] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. *Trans. Storage*, 1(2):133–170, May 2005.

[105] A. S. Spinelli, C. M. Compagnoni, and A. L. Lacaita. Reliability of NAND flash memories: Planar cells and emerging issues in 3D devices. *Computers*, 6(2):16, 2017.

[106] I. Tamo and A. Barg. A family of optimal locally recoverable codes. *IEEE Transactions on Information Theory*, 60(8):4661–4676, 2014.

[107] S. Tanakamaru, M. Fukuda, K. Higuchi, A. Esumi, M. Ito, K. Li, and K. Takeuchi. Post-manufacturing, 17-times acceptable raw bit error rate enhancement, dynamic codeword transition ECC scheme for highly reliable solid-state drives, SSDs. *Solid-State Electronics*, 58(1):2–10, 2011.

[108] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. RIPQ: Advanced photo caching on flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 373–386, Santa Clara, CA, 2015.

[109] J. Terrace and M. J. Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proceedings of the USENIX Annual Technical Conference*, 2009.

[110] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 172–182. ACM, 1995.

[111] The Apache Software Foundation. Apache Cassandra. `http://cassandra.apache.org/`.

[112] A. Thomson and D. J. Abadi. CalvinFS: consistent WAN replication and scalable metadata management for distributed file systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015.

[113] J. Thorpe. Low-density parity-check (LDPC) codes constructed from protographs. *IPN progress report*, 42(154), 2003.

[114] B. G. Tudorica and C. Bucur. A comparison between several NoSQL databases with comments and notes. In *Proceedings of Roedunet International Conference*. IEEE, 2011.

[115] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees and spatio-temporal query processing. *The Computer Journal*, 43(3):325–343, 2000.

[116] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[117] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In P. Druschel, F. Kaashoek, and A. Rowstron, editors, *Peer-to-Peer Systems*, pages 328–337, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[118] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, et al. vcorfu: A cloud-scale object store on a shared log. In *NSDI*, pages 35–49, 2017.

[119] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.

[120] G. Zemor and G. D. Cohen. Error-correcting WOM-codes. *IEEE Transactions on Information Theory*, 37(3):730–734, 1991.

[121] K. Zhao, W. Zhao, H. Sun, X. Zhang, N. Zheng, and T. Zhang. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 243–256, San Jose, CA, 2013.